

Anis Koubaa *Editor*

Robot Operating System (ROS)

The Complete Reference (Volume 2)



Springer

Studies in Computational Intelligence

Volume 707

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

About this Series

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 worldwide distribution, which enable both wide and rapid dissemination of research output.

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

Anis Koubaa
Editor

Robot Operating System (ROS)

The Complete Reference (Volume 2)

Special focus on *Unmanned Aerial Vehicles (UAVs) with ROS*



Springer

Editor

Anis Koubaa
Prince Sultan University
Riyadh
Saudi Arabia

and

CISTER Research Unit
Porto
Portugal

and

Gaitech Robotics
Hong Kong
China

ISSN 1860-949X

ISSN 1860-9503 (electronic)

Studies in Computational Intelligence

ISBN 978-3-319-54926-2

ISBN 978-3-319-54927-9 (eBook)

DOI 10.1007/978-3-319-54927-9

Library of Congress Control Number: 2017933861

© Springer International Publishing AG 2017

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, express 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.

Printed on acid-free paper

This Springer imprint is published by Springer Nature

The registered company is Springer International Publishing AG

The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Acknowledgements

The Editor would like to thank the Robotics and Internet of Things (RIoT) Unit at Center of Excellence of Prince Sultan University for their support to this work.

Furthermore, the Editor thanks Gaitech Robotics in China for their support.



Acknowledgements to Reviewers

The Editor would like to thank the following reviewers for their great contributions in the review process of the book by providing a quality feedback to authors.

Anis	Koubâa	Prince Sultan University, Saudi Arabia/CISTER Research Unit, Portugal
Francisco	Grau	CATEC (Center for Advanced Aerospace Technologies)
Michael	Carroll	Robotic Paradigm Systems
Bence	Magyar	PAL Robotics
Maram	Alajlan	Al-Imam Mohamed bin Saud University
Marc	Morenza-Cinos	UPF
Andre	Oliveira	UTFPR
Marco	Wehrmeister	Federal University of Technology – Parana
Walter	Fetter Lages	Universidade Federal do Rio Grande do Sul
Péter	Fankhauser	ETH Zurich
Christoph	Rösmann	Institute of Control Theory and Systems Engineering, TU Dortmund University
Francesco	Rovida	Aalborg University of Copenhagen
Christopher-Eyk	Hrabia	Technische Universität/DAI Labor
Guilherme	Sousa Bastos	UNIFEI
Andreas	Bühlmaier	Karlsruhe Institute of Technology (KIT)
Juan	Jimeno	linorobot.org
Timo	Röhling	Fraunhofer FKIE
Zavier	Lee	Henan University of Science and Technology
Myrel	Alsayegh	RST-TU Dortmund
Junhao	Xiao	National University of Defense Technology
Huimin	Lu	National University of Defense Technology
Alfredo	Soto	Freescale Semiconductors
Dinesh	Madusanke	University of Moratuwa

(continued)

(continued)

Roberto	Guzman	Robotnik
Ingo	Lütkebohle	Robert Bosch GmbH
Brad	Bazemore	University of Georgia
Yasir	Javed	Prince Sultan University, Saudi Arabia
Mohamed-Foued	Sriti	Al-Imam Muhammad Ibn Saud Islamic University
Murilo	Martins	Centro Universitario da FEI

Contents

Part I Control of UAVs

Model Predictive Control for Trajectory Tracking of Unmanned Aerial Vehicles Using Robot Operating System	3
Mina Kamel, Thomas Stastny, Kostas Alexis and Roland Siegwart	
Designing Fuzzy Logic Controllers for ROS-Based Multirotors	41
Emanoel Koslosky, André Schneider de Oliveira, Marco Aurélio Wehrmeister and João Alberto Fabro	
Flying Multiple UAVs Using ROS	83
Wolfgang Höning and Nora Ayanian	

Part II Control of Mobile Robots

SkiROS—A Skill-Based Robot Control Platform on Top of ROS	121
Francesco Rovida, Matthew Crosby, Dirk Holz, Athanasios S. Polydoros, Bjarne Großmann, Ronald P.A. Petrick and Volker Krüger	
Control of Mobile Robots Using ActionLib	161
Higor Barbosa Santos, Marco Antônio Simões Teixeira, André Schneider de Oliveira, Lúcia Valéria Ramos de Arruda and Flávio Neves, Jr.	
Parametric Identification of the Dynamics of Mobile Robots and Its Application to the Tuning of Controllers in ROS	191
Walter Fetter Lages	
Online Trajectory Planning in ROS Under Kinodynamic Constraints with Timed-Elastic-Bands	231
Christoph Rösmann, Frank Hoffmann and Torsten Bertram	

Part III Integration of ROS with Internet and Distributed Systems**ROSLink: Bridging ROS with the Internet-of-Things for Cloud**

- Robotics** 265
Anis Koubaa, Maram Alajlan and Basit Qureshi

- ROS and Docker** 285
Ruffin White and Henrik Christensen

A ROS Package for Dynamic Bandwidth Management

- in Multi-robot Systems** 309
Ricardo Emerson Julio and Guilherme Sousa Bastos

Part IV Service Robots and Fields Experimental**An Autonomous Companion UAV for the SpaceBot**

- Cup Competition 2015** 345
Christopher-Eyk Hrabia, Martin Berger, Axel Hessler,
Stephan Wypler, Jan Brehmer, Simon Matern and Sahin Albayrak

Development of an RFID Inventory Robot (AdvanRobot) 387

- Marc Morenza-Cinos, Victor Casamayor-Pujol, Jordi Soler-Busquets,
José Luis Sanz, Roberto Guzmán and Rafael Pous

Robotnik—Professional Service Robotics Applications

- with ROS (2)** 419
Roberto Guzmán, Román Navarro, Miquel Cantero
and Jorge Ariño

**Using ROS in Multi-robot Systems: Experiences
and Lessons Learned from Real-World Field Tests** 449

- Mario Garzón, João Valente, Juan Jesús Roldán,
David Garzón-Ramos, Jorge de León, Antonio Barrientos
and Jaime del Cerro

Part V Perception and Sensing**Autonomous Navigation in a Warehouse**

- with a Cognitive Micro Aerial Vehicle** 487
Marius Beul, Nicola Krombach, Matthias Nieuwenhuisen,
David Droschel and Sven Behnke

Robots Perception Through 3D Point Cloud Sensors 525

- Marco Antonio Simões Teixeira, Higor Barbosa Santos,
André Schneider de Oliveira, Lucia Valeria Arruda and Flávio Neves, Jr.

Part VI ROS Simulation Frameworks

Environment for the Dynamic Simulation of ROS-Based UAVs	565
Alvaro Rogério Cantieri, André Schneider de Oliveira, Marco Aurélio Wehrmeister, João Alberto Fabro and Marlon de Oliveira Vaz	
Building Software System and Simulation Environment for RoboCup MSL Soccer Robots Based on ROS and Gazebo	597
Junhao Xiao, Dan Xiong, Weijia Yao, Qinghua Yu, Huimin Lu and Zhiqiang Zheng	
VIKI—More Than a GUI for ROS	633
Robin Hoogervorst, Cees Trouwborst, Alex Kamphuis and Matteo Fumagalli	

Editor and Contributors

About the Editor

Anis Koubaa is a full professor in Computer Science at Prince Sultan University and research associate in CISTER Research Unit, ISEP-IPP, Portugal, and Senior Research Consultant with Gaitech Robotics, China. He becomes a Senior Fellow of the Higher Education Academy (SFHEA) in 2015. He received his B.Sc. in Telecommunications Engineering from Higher School of Telecommunications (Tunisia), and M.Sc. degrees in Computer Science from University Henri Poincaré (France), in 2000 and 2001, respectively, and the Ph.D. degree in Computer Science from the National Polytechnic Institute of Lorraine (France), in 2004. He was a faculty member at Al-Imam University from 2006 to 2012. He has published over 120 refereed journal and conference papers. His research interest covers mobile robots, cloud robotics, robotics software engineering, Internet-of-Things, cloud computing and wireless sensor networks. Dr. Anis received the best research award from Al-Imam University in 2010, and the best paper award of the 19th Euromicro Conference in Real-Time Systems (ECRTS) in 2007. He is the head of the ACM Chapter in Prince Sultan University. His H-Index is 30.

Contributors

Maram Alajlan Center of Excellence Robotics and Internet of Things (RIOT) Research Unit, Prince Sultan University, Riyadh, Saudi Arabia; King Saud University, Riyadh, Saudi Arabia

Sahin Albayrak DAI-Labor, Technische Universität Berlin, Berlin, Germany

Kostas Alexis University of Nevada, Reno, NV, USA

Jorge Ariño Robotnik Automation, SLL, Ciutat de Barcelona, Paterna, Valencia, Spain

Lucia Valeria Arruda Federal University of Technology—Parana, Curitiba, Brazil

Nora Ayanian Department of Computer Science, University of Southern California, Los Angeles, CA, USA

Antonio Barrientos Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

Guilherme Sousa Bastos System Engineering and Information Technology Institute—IESTI, Federal University of Itajubá—UNIFEI, Pinheirinho, Itajubá, MG, Brazil

Sven Behnke Autonomous Intelligent Systems Group, University of Bonn, Bonn, Germany

Martin Berger DAI-Labor, Technische Universität Berlin, Berlin, Germany

Torsten Bertram Institute of Control Theory and Systems Engineering, TU Dortmund University, Dortmund, Germany

Marius Beul Autonomous Intelligent Systems Group, University of Bonn, Bonn, Germany

Jan Brehmer DAI-Labor, Technische Universität Berlin, Berlin, Germany

Miquel Cantero Robotnik Automation, SLL, Ciutat de Barcelona, Paterna, Valencia, Spain

Alvaro Rogério Cantieri Federal Institute of Paraná, Curitiba, Brazil

Victor Casamayor-Pujol Universitat Pompeu Fabra, Barcelona, Spain

Henrik Christensen Contextual Robotics Institute, University of California, San Diego, CA, USA

Matthew Crosby Heriot-Watt University, Edinburgh, UK

Lúcia Valéria Ramos de Arruda Federal University of Technology—Parana, Curitiba, Brazil

Jorge de León Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

André Schneider de Oliveira Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Curitiba, Brazil

Marlon de Oliveira Vaz Federal Institute of Paraná, Curitiba, Brazil

Jaime del Cerro Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

David Droeuschel Autonomous Intelligent Systems Group, University of Bonn, Bonn, Germany

João Alberto Fabro Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Curitiba, Brazil

Matteo Fumagalli Aalborg University, Copenhagen, Denmark

David Garzón-Ramos Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

- Mario Garzón** Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain
- Bjarne Großmann** Aalborg University Copenhagen, Copenhagen, Denmark
- Roberto Guzmán** Robotnik Automation S.L.L., Paterna, Valencia, Spain
- Axel Hessler** DAI-Labor, Technische Universität Berlin, Berlin, Germany
- Frank Hoffmann** Institute of Control Theory and Systems Engineering, TU Dortmund University, Dortmund, Germany
- Dirk Holz** Bonn University, Bonn, Germany
- Robin Hoogervorst** University of Twente, Enschede, Netherlands
- Christopher-Eyk Hrabia** DAI-Labor, Technische Universität Berlin, Berlin, Germany
- Wolfgang Höning** Department of Computer Science, University of Southern California, Los Angeles, CA, USA
- Ricardo Emerson Julio** System Engineering and Information Technology Institute—IESTI, Federal University of Itajubá—UNIFEI, Pinheirinho, Itajubá, MG, Brazil
- Mina Kamel** Autonomous System Lab, ETH Zurich, Zurich, Switzerland
- Alex Kamphuis** University of Twente, Enschede, Netherlands
- Emanoel Koslosky** Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Curitiba, Brazil
- Anis Koubaa** Center of Excellence Robotics and Internet of Things (RIOT) Research Unit, Prince Sultan University, Riyadh, Saudi Arabia; Gaitech Robotics, Hong Kong, China; CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal
- Nicola Krombach** Autonomous Intelligent Systems Group, University of Bonn, Bonn, Germany
- Volker Krüger** Aalborg University Copenhagen, Copenhagen, Denmark
- Walter Fetter Lages** Federal University of Rio Grande do Sul, Porto Alegre RS, Brazil
- Huimin Lu** College of Mechatronics and Automation, National University of Defense Technology, Changsha, China
- Simon Matern** Technische Universität Berlin, Berlin, Germany
- Marc Morenza-Cinos** Universitat Pompeu Fabra, Barcelona, Spain
- Román Navarro** Robotnik Automation, SLL, Ciutat de Barcelona, Paterna, Valencia, Spain

Flávio Neves Jr. Federal University of Technology—Parana, Curitiba, Brazil

Matthias Nieuwenhuisen Autonomous Intelligent Systems Group, University of Bonn, Bonn, Germany

Ronald P.A. Petrick Heriot-Watt University, Edinburgh, UK

Athanasiros S. Polydoros Aalborg University Copenhagen, Copenhagen, Denmark

Rafael Pous Universitat Pompeu Fabra, Barcelona, Spain

Basit Qureshi Prince Sultan University, Riyadh, Saudi Arabia

Juan Jesús Roldán Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

Francesco Rovida Aalborg University Copenhagen, Copenhagen, Denmark

Christoph Rösmann Institute of Control Theory and Systems Engineering, TU Dortmund University, Dortmund, Germany

Higor Barbosa Santos Federal University of Technology—Parana, Curitiba, Brazil

José Luis Sanz Keonn Technologies S.L., Barcelona, Spain

Roland Siegwart Autonomous System Lab, ETH Zurich, Zurich, Switzerland

Jordi Soler-Busquets Universitat Pompeu Fabra, Barcelona, Spain

Thomas Stastny Autonomous System Lab, ETH Zurich, Zurich, Switzerland

Marco Antonio Simões Teixeira Federal University of Technology—Parana, Curitiba, Brazil

Cees Trouwborst University of Twente, Enschede, Netherlands

João Valente Centro De Automática y Robótica, UPM-CSIC, Madrid, Spain

Marco Aurélio Wehrmeister Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Curitiba, Brazil

Ruffin White Contextual Robotics Institute, University of California, San Diego, CA, USA

Stephan Wypler Technische Universität Berlin, Berlin, Germany

Junhao Xiao College of Mechatronics and Automation, National University of Defense Technology, Changsha, China

Dan Xiong College of Mechatronics and Automation, National University of Defense Technology, Changsha, China

Weijia Yao College of Mechatronics and Automation, National University of Defense Technology, Changsha, China

Qinghua Yu College of Mechatronics and Automation, National University of Defense Technology, Changsha, China

Zhiqiang Zheng College of Mechatronics and Automation, National University of Defense Technology, Changsha, China

Part I

Control of UAVs

Model Predictive Control for Trajectory Tracking of Unmanned Aerial Vehicles Using Robot Operating System

Mina Kamel, Thomas Stastny, Kostas Alexis and Roland Siegwart

Abstract In this chapter, strategies for Model Predictive Control (MPC) design and implementation for Unmanned Aerial Vehicles (UAVs) are discussed. This chapter is divided into two main sections. In the first section, modelling, controller design and implementation of MPC for multi-rotor systems is presented. In the second section, we show modelling and controller design techniques for fixed-wing UAVs. System identification techniques are used to derive an estimate of the system model, while state of the art solvers are employed to solve the optimization problem online. By the end of this chapter, the reader should be able to implement an MPC to achieve trajectory tracking for both multi-rotor systems and fixed-wing UAVs.

1 Introduction

Aerial robots are gaining great attention recently as they have many advantages over ground robots to execute inspection, search and rescue, surveillance and goods delivery tasks. Depending on the task required to be executed, a multi-rotor system or fixed-wing aircraft might be a more suitable choice. For instance, a fixed-wing aircraft is more suitable for surveillance and large-scale mapping tasks thanks to their long endurance capability and higher speed compared to a multi-rotor system, while for an inspection task that requires flying close to structures to obtain detailed footage a multi-rotor UAV is more appropriate.

M. Kamel (✉) · T. Stastny · R. Siegwart
Autonomous System Lab, ETH Zurich, Zurich, Switzerland
e-mail: fmina@ethz.ch

T. Stastny
e-mail: tstastny@ethz.ch

R. Siegwart
e-mail: rsiegwart@ethz.ch

K. Alexis
University of Nevada, Reno, NV, USA
e-mail: kalexis@unr.edu

Precise trajectory tracking is a demanding feature for aerial robots in general in order to successfully perform required tasks, especially when operating in realistic environments where external disturbances may heavily affect the flight performance and when flying in the vicinity of structure. In this chapter, several model predictive control strategies for trajectory tracking are presented for multi-rotor systems as well as for fixed-wing aircraft. The general control structure followed by this chapter is a cascade control approach, where a reliable and system-specific low-level controller is present as inner loop, and a model-based trajectory tracking controller is running as an outer loop. This approach is motivated by the fact that many critical flight software is running on a separate navigation hardware which is typically based on micro-controllers, such as Pixhawk PX4 and Navio [1, 2] while high level tasks are running on more powerful -but less reliable- on-board computers. This introduces a separation layer to keep critical tasks running despite any failure in the more complex high-level computer.

By the end of this chapter, the reader should be able to implement and test various Model Predictive Control strategies for aerial robots trajectory tracking, and integrate these controllers into the Robot Operating System (ROS) [3]. Various implementation hints and practical suggestions are provided in this chapter and we show several experimental results to evaluate the proposed control algorithms on real systems.

In Sect. 2 the general theory behind MPC is presented, with focus on linear MPC and robust linear MPC. In Sect. 3 we present the multi-rotor system model and give hints on model identification approaches. Moreover, linear and robust MPC are presented and we show how to integrate these controller into ROS and present experimental validation results. In Sect. 4 we present a Nonlinear MPC approach for lateral-directional position control of fixed-wing aircraft with implementation hints and validation experiments.

2 Background

2.1 Concepts of Receding Horizon Control

Receding Horizon Control (RHC) corresponds to the historic evolution in control theory that aimed to attack the known challenges of fixed horizon control. Fixed horizon optimization computes a sequence of control actions $\{u_0, u_1, \dots, u_{N-1}\}$ over a horizon N and is characterized by two main drawbacks, namely: (a) when an unexpected (unknown during the control design phase) disturbance takes place or when the model employed for control synthesis behaves different than the actual system, then the controller has no way to account for that over the computed control sequence, and (b) as one approaches the final control steps (over the computer fixed horizon) the control law “gives up trying” since there is too little time left in the fixed horizon to go to achieve a significant objective function reduction. To address these limitations, RHC proposed the alternative strategy of computing the full control

sequence, applying only the first step of it and then repeating the whole process iteratively (receding horizon fashion). RHC strategies are in general applicable to nonlinear dynamics of the form (considering that state update is available):

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \quad (1)$$

where the vector field $\mathbf{f} : \mathbb{R}^n \times \mathbb{R}^m$, $\mathbf{x} \in \mathbb{R}^{n \times 1}$ represents the state vector, and $\mathbf{u} \in \mathbb{R}^{m \times 1}$ the input vector. The general state feedback-based RHC optimization problem takes the following form:

$$\begin{aligned} \min_{\mathbf{z}} \quad & F(\mathbf{x}_{t+N}) + \sum_{k=0}^{N-1} \|\mathbf{x}_{t+k} - \mathbf{x}_{t+k}^{ref}\|_\ell + \|\mathbf{u}_{t+k}\|_\ell \\ \text{s.t.} \quad & \mathbf{x}_{t+k+1} = f(\mathbf{x}_{t+k}, \mathbf{u}_{t+k}) \\ & \mathbf{u}_{t+k} \in \mathcal{U}_C \\ & \mathbf{x}_{t+k} \in \mathcal{X}_C \\ & \mathbf{x}_t = \mathbf{x}(t) \end{aligned} \quad (2)$$

where $\mathbf{z} = \{\mathbf{u}_t, \mathbf{u}_{t+1}, \dots, \mathbf{u}_{t+N-1}\}$ is the optimization variables, ℓ denotes some (penalized) metric used for per-stage weighting, $F(\mathbf{x}_{t+N})$ represents the terminal state weighting, \mathbf{x}_{t+k}^{ref} is the reference signal, the subscript $t + k$ is used to denote the sample (using a fixed sampling time T_s) of a signal at k steps ahead of the current time t , while $t + k + 1$ indicates the next evolution of that, \mathcal{U}_C represents the set of input constraints, \mathcal{X}_C the state constraints and $\mathbf{x}(t)$ is the value of the state vector at the beginning of the current RHC iteration. The solution of this optimization problem leads again to an optimal control sequence $\{\mathbf{u}_t^*, \mathbf{u}_{t+1}^*, \dots, \mathbf{u}_{t+N-1}^*\}$ but only the first step of that \mathbf{u}_t^* is applied while the whole process is then repeated iteratively.

Within this formulation, the term $F(\mathbf{x}_{t+N})$ has a critical role for the closed-loop stability. In particular, it forces the system state to take values within a particular set at the end of the prediction horizon. It is relatively easy to prove stability per local iteration using Lyapunov analysis. In its simplest case, this essentially means that considering the regulation problem ($\mathbf{x}_{t+k}^{ref} = 0$ for $k = 0, \dots, N - 1$), and a “decreasing” metric ℓ , then the solution of the above optimization problem makes the system stable at $\mathbf{x}_t = \mathbf{0}$, $\mathbf{u}_t = \mathbf{0}$ – that is that a terminal constraint $\mathbf{x}_{t+N} = 0$ is introduced (a simplified illustration is provided in Fig. 1). However, the question of global stability

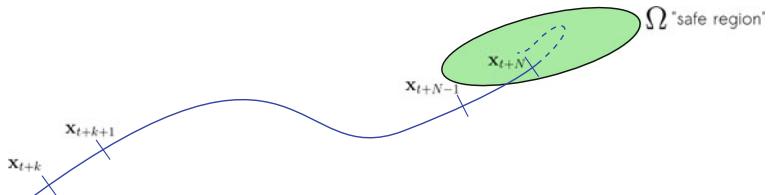


Fig. 1 Illustration of the terminal constraint set (Ω)

is in general not guaranteed. For that one has to consider the problem of introducing both a terminal cost and a terminal constraint for the states [4]. However, general constrained optimization problems can be extremely difficult to solve, and simply adding terminal constraints may not be feasible. Note that in many practical cases, the terminal constraint is not enforced during the control design procedure, but rather verified a posteriori (by increasing the prediction horizon if not satisfied).

Furthermore, one of the most challenging properties of RHC is that of recursive feasibility. Unfortunately, although absolutely recommended from a theoretical standpoint, it is not always possible to construct a RHC that has a-priori guarantee of recursive feasibility, either due to theoretical or practical implications. In general, a RHC strategy lacks recursive feasibility –and is therefore invalidated– even when it is possible to find a state which is feasible, but where the optimal control action moves the state vector to a point where the RHC optimization problem is infeasible. Although a general feasibility analysis methodology is very challenging, for specific cases powerful tools exist. In particular, for the case of linear systems then the Farkas' Lemma [5] in combination with bilevel programming can be used to search for problematic initial states which lack recursive feasibility – thus invalidating an RHC strategy.

2.2 Linear Model Predictive Control

In this subsection we briefly present the theory behind MPC for linear systems. We formulate the optimal control problem for linear systems with linear constraints in the input and state variables. Moreover, we discuss the control input properties, stability and feasibility in the case of linear and quadratic cost function. To achieve offset free tracking under model mismatch, we adopt the approach described in [6] where the system model is augmented with additional disturbances state $\mathbf{d}(t)$ to capture the model mismatch. An observer is employed to estimate disturbances in steady state. The observer design and the disturbance model will be briefly discussed in this subsection.

$$\begin{aligned} \min_{\mathbf{U}} \quad & J_0(\mathbf{x}_0, \mathbf{U}, \mathbf{X}^{ref}, \mathbf{U}^{ref}) \\ \text{subject to} \quad & \mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{B}_d\mathbf{d}_k; \\ & \mathbf{d}_{k+1} = \mathbf{d}_k, \quad k = 0, \dots, N-1 \\ & \mathbf{x}_k \in \mathcal{X}_C, \quad \mathbf{u}_k \in \mathcal{U}_C \\ & \mathbf{x}_N \in \mathcal{X}_{CN} \\ & \mathbf{x}_0 = \mathbf{x}(t_0), \quad \mathbf{d}_0 = \mathbf{d}(t_0). \end{aligned} \tag{3}$$

The optimal control problem to achieve offset-free state tracking under linear state and input constraints is shown in (3), where J_0 is the cost function, $\mathbf{X}^{ref} = \{\mathbf{x}_0^{ref}, \dots, \mathbf{x}_N^{ref}\}$ is the reference state sequence, $\mathbf{U} = \{\mathbf{u}_0, \dots, \mathbf{u}_{N-1}\}$ and $\mathbf{U}^{ref} = \{\mathbf{u}_0^{ref}, \dots, \mathbf{u}_{N-1}^{ref}\}$ are respectively the control input sequence and the steady state input sequence, \mathbf{B}_d is the disturbance model and \mathbf{d}_k is the external disturbances, \mathcal{X}_C ,

\mathcal{U}_C and \mathcal{X}_{CN} are polyhedra. The choice of the disturbance model is not a trivial task, and depends on the system under consideration and the type of disturbances expected. The optimization problem is defined as

$$J_0(\mathbf{x}_0, \mathbf{U}, \mathbf{X}^{ref}, \mathbf{U}^{ref}) = \sum_{k=0}^{N-1} \left((\mathbf{x}_k - \mathbf{x}_k^{ref})^T \mathbf{Q}_x (\mathbf{x}_k - \mathbf{x}_k^{ref}) + \right. \\ (\mathbf{u}_k - \mathbf{u}_k^{ref})^T \mathbf{R}_u (\mathbf{u}_k - \mathbf{u}_k^{ref}) + \\ (\mathbf{u}_k - \mathbf{u}_{k-1})^T \mathbf{R}_\Delta (\mathbf{u}_k - \mathbf{u}_{k-1}) \Big) + \\ (\mathbf{x}_N - \mathbf{x}_N^{ref})^T \mathbf{P} (\mathbf{x}_N - \mathbf{x}_N^{ref}), \quad (4)$$

where $\mathbf{Q}_x \succeq 0$ is the penalty on the state error, $\mathbf{R}_u \succ 0$ is the penalty on control input error, $\mathbf{R}_\Delta \succeq 0$ is a penalty on the control change rate and \mathbf{P} is the terminal state error penalty.

In general, stability and feasibility of receding horizon problems are not ensured except for particular cases such as infinite horizon control problems as in Linear Quadratic Regulator (LQR) case. When the prediction horizon is limited to N steps, the stability and feasibility guarantees are disputable. In principle, longer prediction horizon tends to improve stability and feasibility properties of the controller, but the computation effort will increase, and for aerial robot application, fast control action needs to be computed on limited computation power platforms. However, the terminal cost \mathbf{P} and terminal constraint \mathcal{X}_{CN} can be chosen such that closed-loop stability and feasibility are ensured [6]. In this chapter we focus more on the choice of terminal weight \mathbf{P} as it is easy to compute, while the terminal constraint is generally more difficult and practically stability is achieved with long enough prediction horizon.

Note that in our cost function (4), we penalize the control input rate $\Delta \mathbf{u}_k$. This ensures smooth control input and avoids undesired oscillations. In the cost function (4), \mathbf{u}_{-1} is the actual control input applied on the system in the previous time step.

As previously mentioned, offset-free reference tracking can be achieved by augmenting the system model with disturbances \mathbf{d}_k to capture the modeling error. Assuming that we want to track the system output $\mathbf{y}_k = \mathbf{Cx}_k$ and achieve steady state offset free tracking $\mathbf{y}_\infty = \mathbf{r}_\infty$. A simple observer that can estimate such disturbance can be achieved as follows

$$\begin{bmatrix} \hat{\mathbf{x}}_{k+1} \\ \hat{\mathbf{d}}_{k+1} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{B}_d \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \hat{\mathbf{x}}_k \\ \hat{\mathbf{d}}_k \end{bmatrix} + \begin{bmatrix} \mathbf{B} \\ \mathbf{0} \end{bmatrix} \mathbf{u}_k + \begin{bmatrix} \mathbf{L}_x \\ \mathbf{L}_d \end{bmatrix} (\mathbf{C}\hat{\mathbf{x}}_k - \mathbf{y}_{m,k}) \quad (5)$$

where $\hat{\mathbf{x}}$ and $\hat{\mathbf{d}}$ are the estimated state and external disturbances, \mathbf{L}_x and \mathbf{L}_d are the observer gains and $\mathbf{y}_{m,k}$ is the measured output at time k .

Under the assumption of stable observer, it is possible to compute the MPC state at steady state \mathbf{x}^{ref} and control input at steady state \mathbf{u}^{ref} by solving the following system of linear equations:

$$\begin{bmatrix} \mathbf{A} - \mathbf{I} & \mathbf{B} \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}^{ref,k} \\ \mathbf{u}^{ref,k} \end{bmatrix} = \begin{bmatrix} -\mathbf{B}_d \hat{\mathbf{d}}_k \\ \mathbf{r}_k \end{bmatrix} \quad (6)$$

2.3 Nonlinear Model Predictive Control

Aerial vehicles behavior is better described by a set of nonlinear differential equations to capture the aerodynamic and coupling effects. Therefore in this subsection we present the theory behind Nonlinear MPC that exploits the full system dynamics, and generally achieve better performance when it comes to aggressive trajectory tracking. The optimization problem for nonlinear MPC is formulated in Eq. (7).

$$\begin{aligned} \min_{\mathbf{U}} \quad & \int_{t=0}^T \|h(\mathbf{x}(t), \mathbf{u}(t)) - \mathbf{y}^{ref}(t)\|_{\mathbf{Q}}^2 dt + \|m(\mathbf{x}(T)) - \mathbf{y}^{ref}(T)\|_{\mathbf{P}}^2 \\ \text{subject to} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t)); \\ & \mathbf{u}(t) \in \mathcal{U}_C \\ & \mathbf{x}(t) \in \mathcal{X}_C \\ & \mathbf{x}(0) = \mathbf{x}(t_0). \end{aligned} \quad (7)$$

A direct multiple shooting technique is used to solve the Optimal Control Problem (OCP) (7). In this approach the system dynamics are discretized over a time grid t_0, \dots, t_N within the time intervals $[t_k, t_{k+1}]$. The inequality constraints and control action are discretized over the same time grid. A Boundary Value Problem (BVP) is solved for each interval and additional continuity constraints are imposed. Due to the nature of the system dynamics and the imposed constraints, the optimization problem becomes a Nonlinear Program (NLP). This NLP is solved using Sequential Quadratic Programming (SQP) technique where the Quadratic Programs (QPs) are solved by active set method using the qpOASES solver [7].

Note that, in case of infeasibility of the underlying QP, $\ell 1$ penalized slack variables are introduced to relax all constraints.

The controller is implemented in a receding horizon fashion where only the first computed control action is applied to the system, and the rest of the predicted state and control trajectory is used as initial guess for the OCP to solve in the next iteration.

2.4 Linear Robust Model Predictive Control

Despite the robustness properties of the nominal MPC formulation, specific robust control variations exist when further robustness guarantees are required. The problem of linear Robust Model Predictive Control (RMPC) may be formulated as a Minimax optimization problem that is solved explicitly. As an optimality metric we may

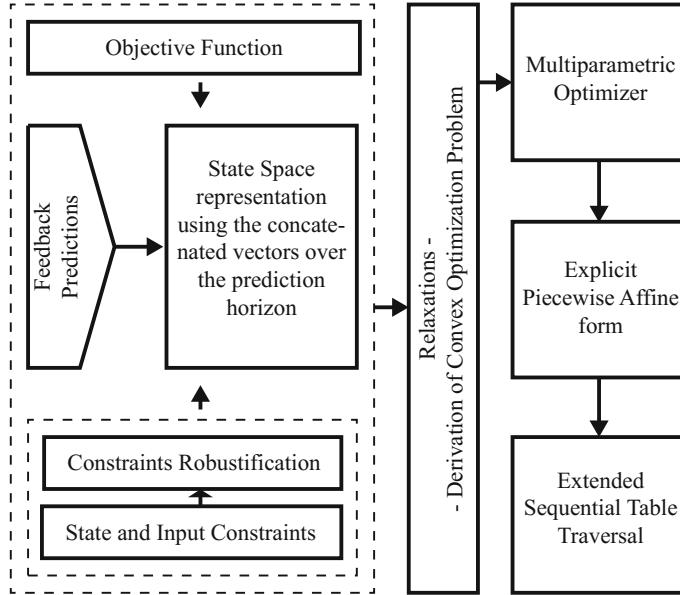


Fig. 2 Overview of the explicit RMPC optimization problem functional components

select the Minimum Peak Performance Measure (MPPM) for its known robustness properties. Figure 2 outlines the relevant building blocks [8].

Within this RMPC approach, the following linear time invariant representation of the system dynamics may be considered:

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k + \mathbf{G}\mathbf{w}_k \\ \mathbf{y}_{k+1} &= \mathbf{C}\mathbf{x}_k \end{aligned} \quad (8)$$

where $\mathbf{x}_k \in \mathbf{X}$, $\mathbf{u}_k \in \mathbf{U}$ and the disturbing signals \mathbf{w}_k are unknown but bounded ($\mathbf{w}_k \in \mathbf{W}$). Within this paper, box-constrained disturbances are considered ($\mathbf{W}_\infty = \{\mathbf{w} : \|\mathbf{w}\|_\infty \leq 1\}$). Consequently, the RMPC problem will be formulated for the system representation and additive disturbance presented above. Let the following denote the concatenated versions of the predicted output, states, inputs and disturbances, where $[k+i|k]$ marks the values profile at time $k+i$, from time k .

$$\mathcal{Y} = \left(\mathbf{y}_{k|k}^T \ \mathbf{y}_{k+1|k}^T \ \dots \ \mathbf{y}_{k+N-1|k}^T \right)^T \quad (9)$$

$$\mathcal{X} = \left(\mathbf{x}_{k|k}^T \ \mathbf{x}_{k+1|k}^T \ \dots \ \mathbf{x}_{k+N-1|k}^T \right)^T \quad (10)$$

$$\mathcal{U} = \left(\mathbf{u}_{k|k}^T \ \mathbf{u}_{k+1|k}^T \ \dots \ \mathbf{u}_{k+N-1|k}^T \right)^T \quad (11)$$

$$\mathcal{W} = \left(\mathbf{w}_{k|k}^T \ \mathbf{w}_{k+1|k}^T \ \dots \ \mathbf{w}_{k+N-1|k}^T \right)^T \quad (12)$$

where $\mathcal{X} \in \mathbb{X}^N = \mathbf{X} \times \mathbf{X} \cdots \times \mathbf{X}$, $\mathcal{U} \in \mathbb{U}^N = \mathbf{U} \times \mathbf{U} \cdots \times \mathbf{U}$, $\mathcal{W} \in \mathbb{W}^N = \mathbf{W} \times \mathbf{W} \times \cdots \times \mathbf{W}$. The predicted states and outputs present linear dependency on the current state, the future control input and the disturbance, and thus the following holds:

$$\begin{aligned}\mathcal{X} &= \mathcal{A}\mathbf{x}_{k|k} + \mathcal{B}\mathcal{U} + \mathcal{G}\mathcal{W} \\ \mathcal{Y} &= \mathcal{C}\mathcal{X}\end{aligned}\quad (13)$$

where $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{G}$ are the stacked state vector matrices as in [8]. Subsequently, the RMPC problem based on the MPPM (MPPM–RMPC) may be formulated as:

$$\begin{aligned}\min_u \max_w \quad &||\mathcal{Y}||_\infty, \quad ||\mathcal{Y}||_\infty = \max_j ||\mathbf{y}_{k+j|k}||_\infty \\ \text{s.t.} \quad &\mathbf{u}_{k+j|k} \in \mathbf{U}, \quad \forall \mathbf{w} \in \mathbf{W} \\ &\mathbf{x}_{k+j|k} \in \mathbf{X}, \quad \forall \mathbf{w} \in \mathbf{W} \\ &\mathbf{w}_{k+j|k} \in \mathbf{W}\end{aligned}\quad (14)$$

2.4.1 Feedback Predictions

Following the aforementioned formulation, the optimization problem will tend to become conservative as the optimization essentially computes an open-loop control sequence. Feedback predictions is a method to encode the knowledge that a receding horizon approach is followed. Towards their incorporation, a type of feedback control structure has to be assumed. Among the feedback parametrizations that are known to lead to a convex problem space, the following is selected [9, 10]:

$$\mathcal{U} = \mathcal{L}\mathcal{W} + \mathcal{V}, \quad \mathcal{V} = \left(v_{k|k}^T \ v_{k+1|k}^T \ \cdots \ v_{k+N-1|k}^T \right)^T \quad (15)$$

$$T\mathcal{L} = \begin{pmatrix} 0 & 0 & 0 & \cdots & 0 \\ L_{10} & 0 & 0 & \cdots & 0 \\ L_{20} & L_{21} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{(N-1)0} & L_{(N-1)1} & \cdots & L_{(N-1)(N-2)} & 0 \end{pmatrix} \quad (16)$$

Employing this feedback predictions parameterization, the control sequence is now parameterized directly in the uncertainty, and the matrix \mathcal{L} describes how the control action uses the disturbance vector. Inserting this parametrization yields the following representation, where \mathcal{V} becomes now the RMPC-manipulated action:

$$\mathcal{X} = \mathcal{A}\mathbf{x}_{k|k} + \mathcal{B}\mathcal{V} + (\mathcal{G} + \mathcal{B}\mathcal{L})\mathcal{W} \quad (17)$$

$$\mathcal{U} = \mathcal{L}\mathcal{W} + \mathcal{V} \quad (18)$$

and the mapping from \mathcal{L}, \mathcal{V} to \mathcal{X}, \mathcal{U} is now bilinear. This allows the formulation of the minimax MPC as a convex optimization problem [9]. Furthermore, let:

$$\mathcal{F}_u = (f_u^T \ f_u^T \ \cdots \ f_u^T) \quad (19)$$

$$\mathcal{F}_x = (f_x^T \ f_x^T \ \cdots \ f_x^T) \quad (20)$$

denote the concatenated –over the prediction horizon– versions of the input and state constraints f_u and f_x . More specifically, $f_u = [f_u(1)^{\max}, f_u(1)^{\min} \dots]$ and $f_x = [f_x(1)^{\max}, f_x(1)^{\min} \dots]$ where $f_u(i)^{\max}, f_u(i)^{\min}$ represent the maximum and minimum allowed input values of the i -th input, while $f_x(j)^{\max}, f_x(j)^{\min}$ represent the maximum and minimum acceptable/safe state configurations of the j -th state.

$$\begin{aligned} & \min_{\mathcal{V}, \mathcal{L}, \tau} \quad \tau \\ \text{s.t.} \quad & \|\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V} + (\mathcal{G} + \mathcal{B}\mathcal{L})\mathcal{W})\|_\infty \leq \tau, \quad \forall \mathcal{W} \in \mathbb{W}^N \\ & \mathcal{E}_u(\mathcal{V} + \mathcal{L}\mathcal{W}) \leq \mathcal{F}_u, \quad \forall \mathcal{W} \in \mathbb{W}^N \\ & \mathcal{E}_x(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V} + (\mathcal{G} + \mathcal{B}\mathcal{L})\mathcal{W}) \leq \mathcal{F}_x, \quad \forall \mathcal{W} \in \mathbb{W}^N \\ & \mathcal{E}_u = \text{diag}^N \mathbf{E}_u, \quad \mathcal{E}_x = \text{diag}^N \mathbf{E}_x, \quad \tau > 0 \end{aligned} \quad (21)$$

within which: (a) \mathbf{E}_x , \mathbf{E}_u are matrices that allow the formulation of the state and input constraints in Linear Matrix Inequality (LMI) form, (b) $\text{diag}^N \Lambda_i$ is a block diagonal matrix with Λ_i being the matrix that fills each diagonal block and allows the incorporation of the state and input constraints. Within this formulation τ is defined as a positive scalar value that bounds (from above) the objective function. The peak constraint may be equivalently reformulated as:

$$\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V}) + \mathcal{C}(\mathcal{G} + \mathcal{B}\mathcal{L})\mathcal{W} \leq \tau \mathbf{1}, \quad \forall \mathcal{W} \in \mathbb{W}^N \quad (22)$$

$$-\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V}) - \mathcal{C}(\mathcal{G} + \mathcal{B}\mathcal{L})\mathcal{W} \leq \tau \mathbf{1}, \quad \forall \mathcal{W} \in \mathbb{W}^N \quad (23)$$

where $\mathbf{1}$ is a vector of ones $(1 \ 1 \ \cdots \ 1)^T$ with suitable dimensions. Satisfaction of these uncertain inequalities is based on robust optimization methods.

2.4.2 Robust Uncertain Inequalities Satisfaction

Since box-constrained disturbances ($\mathbf{w} \in \mathbf{W}_\infty$) are assumed, the following holds:

$$\max_{|x| \leq 1} c^T x = \|c\|_1 = |c^T \mathbf{1}| \quad (24)$$

This equation holds as $\max_{|x| \leq 1} c^T x = \max_{|x| \leq 1} \sum c_i x_i = \sum c_i \text{sign}(c_i) = \|c\|_1$. Consequently, the uncertain constraints with $w \in \mathbb{W}_\infty$ are satisfied as long as [9]:

$$\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V}) + |\mathcal{C}(\mathcal{G} + \mathcal{B}\mathcal{L})| \mathbf{1} \leq \tau \mathbf{1} \quad (25)$$

$$-\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{B}\mathcal{V}) + |\mathcal{C}(\mathcal{G} + \mathcal{B}\mathcal{L})| \mathbf{1} \leq \tau \mathbf{1} \quad (26)$$

To handle these constraints in a linear programming fashion [5], the term $|\mathcal{C}(\mathcal{G} + \mathcal{BL})|$ is bounded from above by introducing a matrix variable $\Gamma \succ 0$:

$$\mathcal{C}(\mathcal{G} + \mathcal{BL}) \leq \Gamma \quad (27)$$

$$-\mathcal{C}(\mathcal{G} + \mathcal{BL}) \leq \Gamma \quad (28)$$

and the peak constraint is guaranteed as long as:

$$\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{BV}) + \Gamma \mathbf{1} \leq \tau \mathbf{1} \quad (29)$$

$$-\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{BV}) + \Gamma \mathbf{1} \leq \tau \mathbf{1} \quad (30)$$

2.4.3 Robust State and Input Constraints

To robustly satisfy hard constraints on the input and the states along the prediction horizon, a new matrix $\Omega \succ 0$ is introduced and the constraints are reformulated as:

$$\begin{pmatrix} \mathcal{E}_x(\mathcal{A}x_{k|k} + \mathcal{BV}) \\ \mathcal{E}_u \mathcal{V} \end{pmatrix} + \Omega \mathbf{1} \leq \begin{pmatrix} \mathcal{F}_x \\ \mathcal{F}_u \end{pmatrix} \quad (31)$$

$$\begin{pmatrix} \mathcal{E}_x(\mathcal{G} + \mathcal{BL}) \\ \mathcal{E}_u \mathcal{L} \end{pmatrix} \leq \Omega \quad (32)$$

$$-\begin{pmatrix} \mathcal{E}_x(\mathcal{G} + \mathcal{BL}) \\ \mathcal{E}_u \mathcal{L} \end{pmatrix} \leq \Omega \quad (33)$$

Optimizing the control sequence, while robustly satisfying the state and input constraints is of essential importance for the flight control of micro aerial vehicles.

2.4.4 Minimum Peak Performance Robust MPC Formulation

Based on the aforementioned derivations, the total MPPM-RMPC formulation is solved subject to element-wise bounded disturbances and feedback predictions through the following linear programming problem:

$$\min_{\mathcal{V}, \mathcal{L}, \tau, \Omega, \Gamma} \tau \quad (34)$$

$$s.t. \quad \mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{BV}) + \Gamma \mathbf{1} \leq \tau \mathbf{1}$$

$$-\mathcal{C}(\mathcal{A}x_{k|k} + \mathcal{BV}) + \Gamma \mathbf{1} \leq \tau \mathbf{1}$$

$$\mathcal{C}(\mathcal{G} + \mathcal{BL}) \leq \Gamma$$

$$-\mathcal{C}(\mathcal{G} + \mathcal{BL}) \leq \Gamma$$

$$\mathcal{E}_x(\mathcal{A}x_{k|k} + \mathcal{BV})$$

$$\begin{pmatrix} \mathcal{E}_x(\mathcal{A}x_{k|k} + \mathcal{BV}) \\ \mathcal{E}_u \mathcal{V} \end{pmatrix} + \Omega \mathbf{1} \leq \begin{pmatrix} \mathcal{F}_x \\ \mathcal{F}_u \end{pmatrix}$$

$$\begin{pmatrix} \mathcal{E}_x(\mathcal{G} + \mathcal{BL}) \\ \mathcal{E}_u \mathcal{L} \end{pmatrix} \leq \Omega$$

$$-\left(\frac{\mathcal{E}_x(\mathcal{G} + \mathcal{BL})}{\mathcal{E}_u \mathcal{L}}\right) \leq \Omega$$

2.4.5 Multiparametric Explicit Solution

The presented RMPC strategy requires the solution of a linear programming problem. However, a multiparametric-explicit solution is possible due to the fact that the control action takes the general form [6]:

$$\mathbf{u}_k = \mathbf{F}^r \mathbf{x}_k + \mathbf{Z}^r, \text{ if } \mathbf{x}_k \in \Pi^r \quad (36)$$

where $\Pi^i, r = 1, \dots, N^r$ are the regions of the receding horizon controller. The r -th control law is valid if the state vector \mathbf{x}_k is contained in a convex polyhedral region $\Pi^r = \{\mathbf{x}_k \mid \mathbf{H}^r \mathbf{x}_k \leq \mathbf{K}^r\}$ computed and described in h -representation [11]. Such a fact enables fast real-time execution even in microcontrollers with very limited computing power. In this framework, the real-time code described in [8] is employed.

3 Model-Based Trajectory Tracking Controller for Multi-rotor System

In this section, we present a simplified model of multi-rotor system that can be used for model-based control to achieve trajectory tracking, and we present a linear and nonlinear model predictive controller for trajectory tracking.

3.1 Multirotor System Model

The 6DoF pose of the multi-rotor system can be defined by assigning a fixed inertial frame \mathbf{W} and body frame \mathbf{B} attached to the vehicle as shown in Fig. 3. We denote by \mathbf{p} the position of the origin of frame \mathbf{B} in frame \mathbf{W} expressed in frame \mathbf{W} , by \mathbf{R} the rotation matrix of frame \mathbf{B} in frame \mathbf{W} expressed in frame \mathbf{W} . Moreover, we denote by ϕ, θ and ψ the roll, pitch and yaw angles of the vehicle. In this model we assume a low level attitude controller that is able to track desired roll and pitch ϕ_d, θ_d angles with a first order behavior. The first order inner-loop approximation provides sufficient information to the MPC to take into account the low level controller behavior. The inner-loop first order parameters can be identified through classic system identification techniques. The non-linear model used for trajectory tracking of multi-rotor system is shown in Eq. (37).

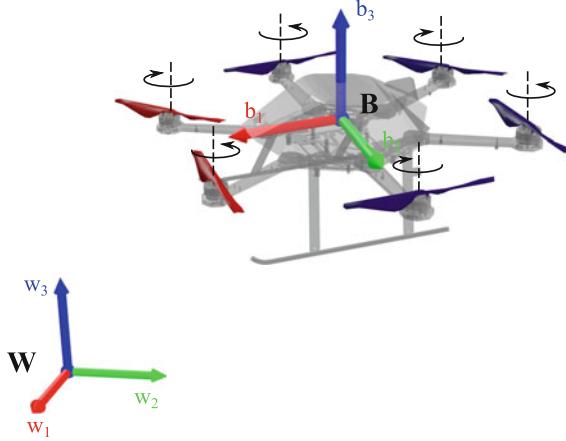


Fig. 3 Illustration of the Firefly hexacopter from Ascending Technologies with attached body fixed frame **B** and inertial frame **W**

$$\begin{aligned} \dot{\mathbf{p}}(t) &= \mathbf{v}(t) \\ \dot{\mathbf{v}}(t) &= \mathbf{R}(\psi, \theta, \phi) \begin{pmatrix} 0 \\ 0 \\ T \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ -g \end{pmatrix} - \begin{pmatrix} A_x & 0 & 0 \\ 0 & A_y & 0 \\ 0 & 0 & A_z \end{pmatrix} \mathbf{v}(t) + \mathbf{d}(t) \\ \dot{\phi}(t) &= \frac{1}{\tau_\phi} (K_\phi \phi_d(t) - \phi(t)) \\ \dot{\theta}(t) &= \frac{1}{\tau_\theta} (K_\theta \theta_d(t) - \theta(t)) \end{aligned} \quad (37)$$

where \mathbf{v} indicates the vehicle velocity, g is the gravitational acceleration, T is the mass normalized thrust, A_x , A_y , A_z indicate the mass normalized drag coefficients, \mathbf{d} is external disturbance. τ_ϕ , K_ϕ and τ_θ , K_θ are the time constant and gain of inner-loop behavior for roll angle and pitch angle respectively.

The cascade controller structure assumed in this chapter is shown in Fig. 4.

3.2 Linear MPC

In this subsection we show how to formulate a linear MPC to achieve trajectory tracking for multi-rotor system and integrate it into ROS. The optimization problem presented in Eq. (3) is solved by generating a C-code solver using the CVXGEN framework [12]. CVXGEN generates a high speed solver for convex optimization problems by exploiting the problem structure. For clarity purposes, we rewrite the optimization problem here and show how to generate a custom solver using CVXGEN. The optimization problem is given by

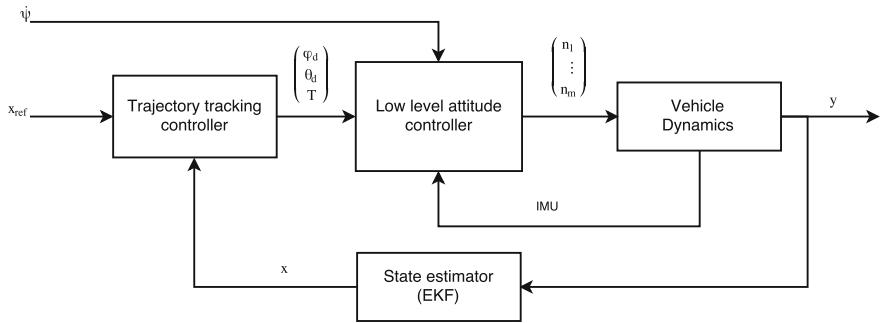


Fig. 4 Controller scheme for multi-rotor system. $n_1 \dots n_m$ are the $i - t h$ rotor speed and y is the measured vehicle state

$$\begin{aligned}
 & \min_{\mathbf{U}} \sum_{k=0}^{N-1} (\mathbf{x}_k - \mathbf{x}_k^{ref})^T \mathbf{Q}_x (\mathbf{x}_k - \mathbf{x}_k^{ref}) + (\mathbf{u}_k - \mathbf{u}_k^{ref})^T \mathbf{R}_u (\mathbf{u}_k - \mathbf{u}_k^{ref}) + (\mathbf{u}_k - \mathbf{u}_{k-1})^T \mathbf{R}_\Delta (\mathbf{u}_k - \mathbf{u}_{k-1}) \\
 & + (\mathbf{x}_N - \mathbf{x}_N^{ref})^T \mathbf{P} (\mathbf{x}_N - \mathbf{x}_N^{ref}) \\
 \text{subject to } & \mathbf{x}_{k+1} = \mathbf{A} \mathbf{x}_k + \mathbf{B} \mathbf{u}_k + \mathbf{B}_d \mathbf{d}_k; \\
 & \mathbf{d}_{k+1} = \mathbf{d}_k, \quad k = 0, \dots, N-1 \\
 & \mathbf{u}_k \in \mathcal{U}_C \\
 & \mathbf{x}_0 = \mathbf{x}(t_0), \quad \mathbf{d}_0 = \mathbf{d}(t_0).
 \end{aligned} \tag{38}$$

To generate a solver for the aforementioned optimization problem, the following problem description is used in CVXGEN.

```

1 dimensions
2   m = 3      # dimension of inputs.
3   nd = 3     # dimension of disturbances.
4   nx = 8      # dimension of state vector.
5   T = 18      # horizon - 1.
6 end
7
8 parameters
9   A (nx,nx)      # dynamics matrix.
10  B (nx,m)        # transfer matrix.
11  Bd (nx,nd)      # disturbance transfer matrix
12  Q_x (nx,nx) psd # state cost, positive semidefined.
13  P (nx,nx) psd # final state penalty, positive semidefined.
14  R_u (m,m) psd # input penalty, positive semidefined.
15  R_delta (m,m) psd # delta input penalty, positive semidefined.
16  x[0] (nx)       # initial state.
17  d (nd)          # disturbances.
18  u_prev (m)      # previous input applied to the system.
19  u_max (m)        # input amplitude limit.
20  u_min (m)        # input amplitude limit.
21  x_ss[t] (nx), t=0..T+1    # reference state.
22  u_ss[t] (m), t=0..T           # reference input.
23 end

```

```

24
25 variables
26   x[ t ] (nx) , t =1..T+1      # state .
27   u[ t ] (m) , t =0..T          # input .
28 end
29
30 minimize
31   quad(x[0]-x_ss[0], Q_x) + quad(u[0]-u_ss[0], R_u) + quad(u[0] -
32     u_prev, R_delta) + sum[ t =1..T](quad(x[ t ]-x_ss[ t ], Q_x) + quad(
33     u[ t ]-u_ss[ t ], R_u) + quad(u[ t ] - u[ t -1], R_delta))+quad(x[T+1]-
34     x_ss[T+1], P)
32 subject to
33   x[ t +1 ] == A*x[ t ] + B*u[ t ] + Bd*d , t =0..T # dynamics
34   u_min <= u[ t ] <= u_max , t =0..T      # input constraint .
35 end

```

Before we show the integration of the generated solver into ROS, we discuss how to estimate the attitude loop parameters τ_ϕ , K_ϕ , τ_θ , K_θ and the derivation of the discrete system model \mathbf{A} , \mathbf{B} , \mathbf{B}_d .

3.2.1 Attitude Loop Parameters Identification

The attitude loop identification is a recommended process when no knowledge is available about the attitude controller used onboard of the vehicle. This is the case for many commercially available platforms. To perform this system identification, typically a pilot excites the vehicles axes in free flight. Attitude command and actual vehicle attitude (estimated using motion capture system if available or Inertial Measurement Unit (IMU)) are logged with accurate time stamp. Typically two datasets are collected, one is used for parameters estimation and the other dataset is used for validation purpose. We will not go into details of system identification, interested readers can find more details in [13]. A set of scripts for attitude dynamics identification will be made open source upon acceptance of the chapter.

To perform parameters identification using the available scripts please follow the following steps:

1. Prepare the system and make sure you are able to log time stamped attitude commands and actual vehicle attitude.
2. Perform a flight logging the commands and vehicle attitude in a bag file. During the flight excite as much as possible each axis of the vehicle.
3. Repeat the flight test to collect validation dataset.
4. Set the correct bag files name and topics name in the provided script and run it.
5. The controller parameters will be displayed on screen with validation percentage to confirm the validity of the identification.

3.2.2 Linearization, Decoupling and Discretization

For controller design, the vehicle dynamics can be approximated around hovering condition where small attitude angles are assumed and vehicle heading aligned with inertial frame \mathbf{x} axis (i.e. $\psi = 0$). The linearized system around hovering condition can be written as

$$\dot{\mathbf{x}}(t) = \underbrace{\begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -A_x & 0 & 0 & g & 0 & 0 \\ 0 & 0 & 0 & 0 & -A_y & 0 & 0 & -g & 0 \\ 0 & 0 & 0 & 0 & 0 & -A_z & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\tau_\phi} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{\tau_\theta} & 0 \end{pmatrix}}_{\mathbf{A}_c} \mathbf{x}(t) + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \\ \frac{K_\phi}{\tau_\phi} & 0 & 0 \\ 0 & \frac{K_\theta}{\tau_\theta} & 0 \end{pmatrix}}_{\mathbf{B}_c} \mathbf{u}(t) + \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{B}_{d,c}} \mathbf{d}(t), \quad (39)$$

where the state vector is $\mathbf{x} = (\mathbf{p}^T, \mathbf{v}^T, {}^W\phi, {}^W\theta)^T$, the input vector $\mathbf{u} = ({}^W\phi_d, {}^W\theta_d, T)^T$ and the disturbance vector $\mathbf{d} = (d_x, d_y, d_z)^T$. The c subscription indicates that this is a continuous time model. Note that in this linearization we marked the attitude to be in inertial frame \mathbf{W} to get rid of the yaw angle ψ from the model. The attitude control action ${}^W\phi_d, {}^W\theta_d$ is computed in inertial frame and then converted to body frame by performing a rotation around z axis. The control action in the vehicle body frame \mathbf{B} is given by

$$\begin{pmatrix} \phi_d \\ \theta_d \end{pmatrix} = \begin{pmatrix} \cos \psi & \sin \psi \\ -\sin \psi & \cos \psi \end{pmatrix} \begin{pmatrix} {}^W\phi_d \\ {}^W\theta_d \end{pmatrix}. \quad (40)$$

After the computation of the control input, it is recommended to add a feed-forward term to compensate for coupling effects and to achieve better tracking performance. To compensate for vehicle non-zero attitude effects on thrust, and to achieve better tracking of dynamic trajectory, the following compensation scheme is employed.

$$\begin{aligned} \tilde{T} &= \frac{T + g}{\cos \phi \cos \theta} + {}^B \ddot{z}_d \\ \tilde{\phi}_d &= \frac{g \phi_d - {}^B \ddot{y}_d}{\tilde{T}} \\ \tilde{\theta}_d &= \frac{g \theta_d + {}^B \ddot{x}_d}{\tilde{T}} \end{aligned} \quad (41)$$

where ${}^B \ddot{x}_d, {}^B \ddot{y}_d, {}^B \ddot{z}_d$ are the desired trajectory acceleration expressed in vehicle body frame and quantities with tilde sign are the actual applied control input.

Given that the controller is implemented in discrete time, it is necessary to discretize the system dynamics (39). This can be done as follows

$$\begin{aligned}\mathbf{A} &= e^{\mathbf{A}_c T_s} \\ \mathbf{B} &= \int_0^{T_s} e^{\mathbf{A}_c \tau} d\tau \mathbf{B}_c \\ \mathbf{B}_d &= \int_0^{T_s} e^{\mathbf{A}_c \tau} d\tau \mathbf{B}_{d,c}\end{aligned}\quad (42)$$

where T_s is the sampling time. The computation of the terminal cost \mathbf{P} matrix is done by solving the Algebraic Riccati Equation iteratively.

3.2.3 ROS Integration

The strategy followed for the ROS integration of the solver is to create a ROS node to interface the controller to ROS environment, while the controller, estimator and other components are implemented as C++ shared libraries that get linked to the node at compilation time. The controller node expects the vehicle state as a nav odometry message and publishes a custom message of type RollPitchYawRateThrust command message. The desired trajectory can be sent to the controller over a topic of type MultiDOFJointTrajectory or as single desired point over a topic of type PoseStamped. The advantage of passing the whole desired trajectory over single point is that the MPC can take into consideration the future desired trajectory and react accordingly. Figure 5 gives an overview of nodes and topics communication through a ros graph diagram.

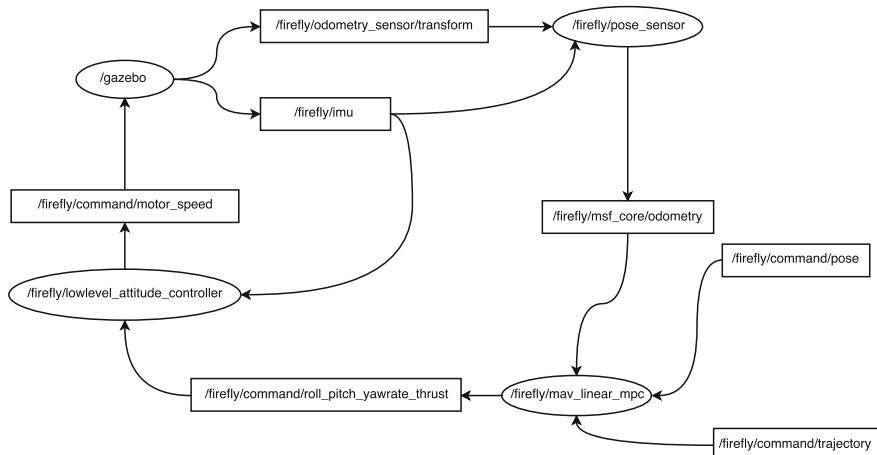


Fig. 5 ROS graph diagram showing various nodes and topics to control multi-rotor system

Each time the controller receives a new odometry message, a control action is computed and published. Therefore it is important to guarantee that the state estimator is publishing an odometry message at the desired rate of control. We recommend to use this controller with the Modular Framework for MultiSensor Fusion [14]. An important point to consider when implementing such a controller is to reduce as much as possible the delays in the loop. A hint to minimize communication delay is to use the ROS transportation hints by passing `ros::TransportHints().tcpNoDelay()` flag to the odometry subscriber.

The controller node publishes the following information:

1. Current desired reference as `t_f`.

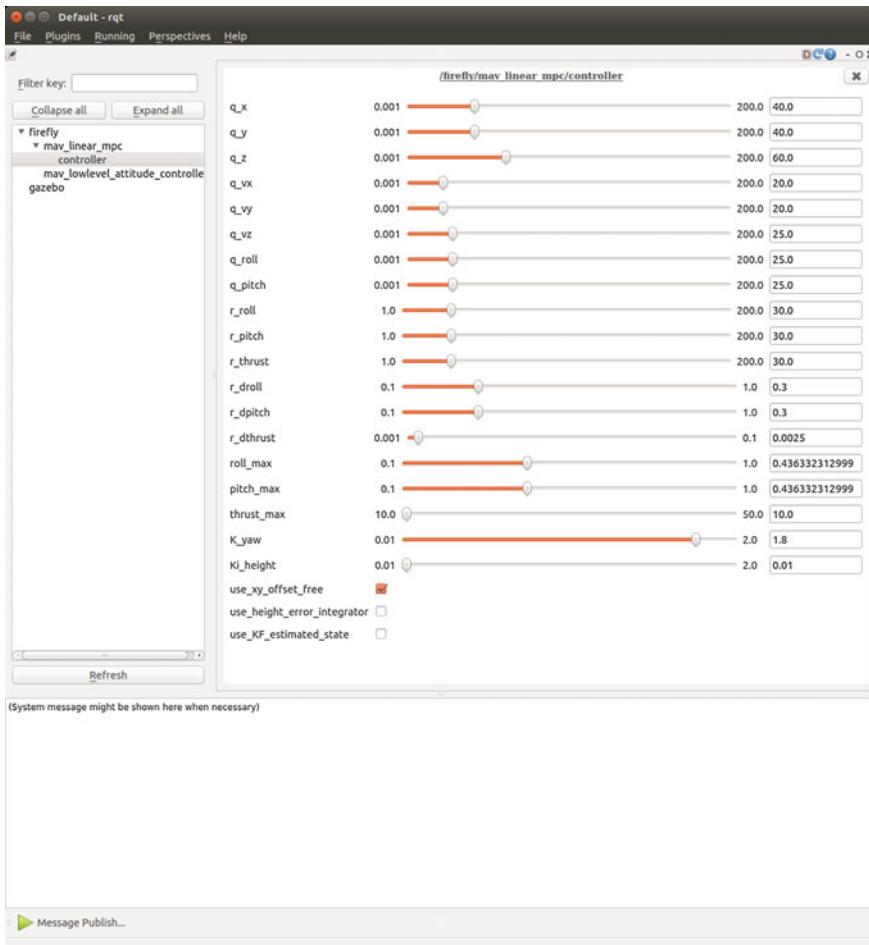


Fig. 6 Through dynamic reconfiguration it is possible to change the controller parameters online for tuning purposes

2. Desired trajectory as rviz marker.
3. Predicted vehicle state as rviz marker.

The controller parameters can be easily changed in run time from the dynamic reconfigure as shown in Fig. 6.

An open source implementation of the presented controller can be found in https://github.com/ethz-asl/mav_control_rw

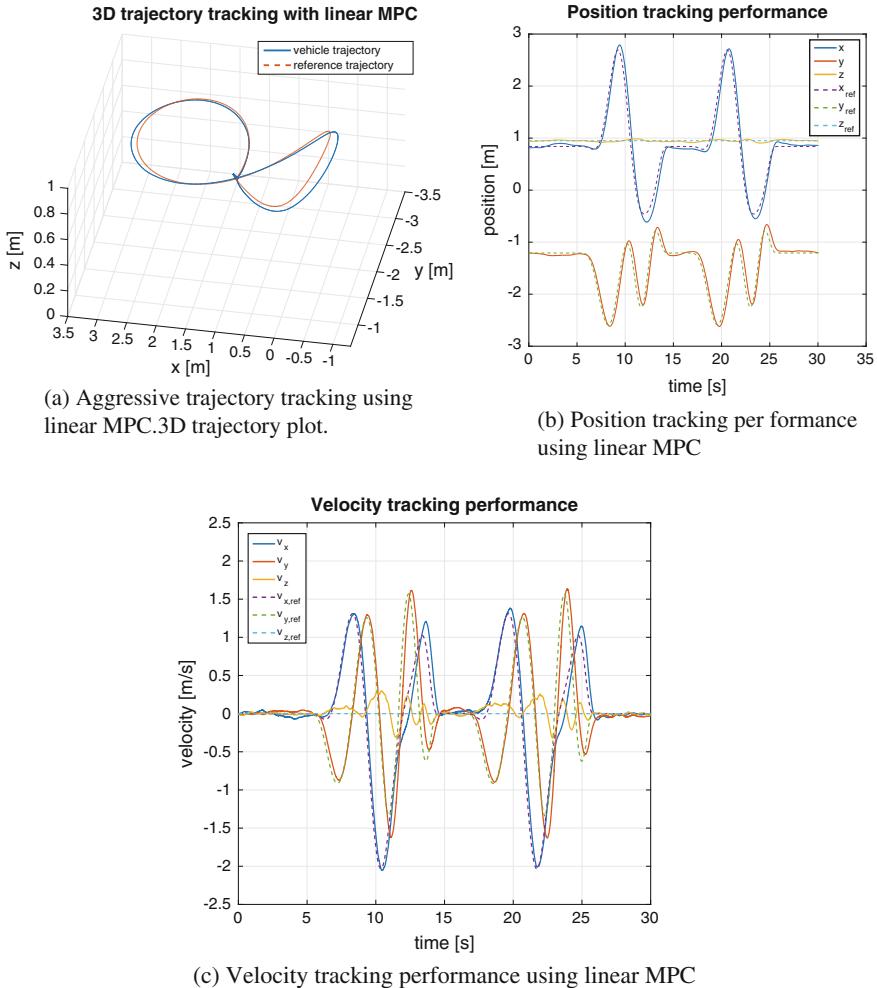


Fig. 7 Aggressive trajectory tracking performance using linear MPC controller running onboard Firefly hexacopter from Ascending Technologies

3.2.4 Experimental Results

To validate the controller performance, we track an aggressive trajectory. The controller is running onboard a Firefly hexacopter from Ascending Technologies (AscTec) with a NUC i7 computer onboard. An external motion capture system Vicon [15] is used as pose sensor and fused with onboard IMU using the Multisensor Fusion Framework (MSF). The controller is running at 100 Hz and the prediction horizon is chosen to be 20 steps.

Figure 7 shows the tracking performance of the controller.

3.3 Nonlinear MPC

In this subsection we use the full vehicle nonlinear model to design a continuous time nonlinear model predictive controller. The toolkit employed to generate the nonlinear solver is ACADO [16] which is able to generate very fast custom C code solvers for general optimal control problems OCP. The optimization problem can be written as

$$\begin{aligned} \min_{\mathbf{U}} \quad & \int_{t=0}^T (\mathbf{x}(t) - \mathbf{x}^{ref}(t))^T Q_x (\mathbf{x}(t) - \mathbf{x}^{ref}(t)) + (\mathbf{u}(t) - \mathbf{u}^{ref}(t))^T \mathbf{R}_u (\mathbf{u}(t) - \mathbf{u}^{ref}(t)) dt \\ & + (\mathbf{x}(T) - \mathbf{x}^{ref}(T))^T \mathbf{P} (\mathbf{x}(T) - \mathbf{x}^{ref}(T)) \\ \text{subject to} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}); \\ & \mathbf{u}(t) \in \mathcal{U}_C \\ & \mathbf{x}(0) = \mathbf{x}(t_0). \end{aligned} \tag{43}$$

To generate a solver for the aforementioned optimization problem, the following problem description is used in ACADO's Matlab interface.

```

1 clc;
2 clear all;
3 close all;
4
5 Ts = 0.1; %sampling time
6 EXPORT = 1;
7
8 DifferentialState position(3) velocity(3) roll pitch yaw;
9 Control roll_ref pitch_ref thrust;
10
11 %online data represent data that can be passed to the solver online.
12 OnlineData roll_tau;
13 OnlineData roll_gain;
14 OnlineData pitch_tau;
15 OnlineData pitch_gain;
16 OnlineData yaw_rate_command;
17 OnlineData drag_coefficient(3);

```

```

18 OnlineData external_disturbances(3);
19
20 n_XD = length(diffStates);
21 n_U = length(controls);
22
23 g = [0;0;9.8066];
24
25 %% Differential Equation
26
27 %define vehicle body z-axis expressed in inertial frame.
28 z_axis = [(cos(yaw)*sin(pitch)*cos(roll)+sin(yaw)*sin(roll));...
29             (sin(yaw)*sin(pitch)*cos(roll)-cos(yaw)*sin(roll));...
30             cos(pitch)*cos(roll)];
31
32 droll = (1/roll_tau)*(roll_gain*roll_ref - roll);
33 dpitch = (1/pitch_tau)*(pitch_gain*pitch_ref - pitch);
34
35 f = dot([position, velocity; roll; pitch; yaw]) == ...
36     [velocity ...
37      z_axis*thrust - g - diag(drag_coefficient)*velocity +
38      external_disturbances;...
39      droll;...
40      dpitch;...
41      yaw_rate_command];
42
43 h = [position; velocity; roll; pitch; roll_ref; pitch_ref; z_axis(3)*
44 thrust-g(3)];
45
46
47 %% NMPCexport
48 acadoSet('problemname','nmpc_trajectory_tracking');
49
50 N = 20;
51 ocp = acado.OCP( 0.0, N*Ts, N );
52
53 W_mat = eye(length(h));
54 WN_mat = eye(length(hN));
55 W = acado.BMatrix(W_mat);
56 WN = acado.BMatrix(WN_mat);
57
58 ocp.minimizeLSQ( W, h );
59 ocp.minimizeLSQEndTerm( WN, hN );
60 ocp.subjectTo(-deg2rad(45) <= [roll_ref; pitch_ref] <= deg2rad(45));
61 ocp.subjectTo( g(3)/2.0 <= thrust <= g(3)*1.5);
62 ocp.setModel(f);
63
64 mpc = acado.OCPexport( ocp );
65 mpc.set('HESSIAN_APPROXIMATION', 'GAUSS_NEWTON' );
66 mpc.set('DISCRETIZATION_TYPE', 'MULTIPLE_SHOOTING' );
67 mpc.set('SPARSE_QP SOLUTION', 'FULL_CONDENSING_N2' );

```

```

68 mpc.set('INTEGRATOR_TYPE',          'INT_IRK_GL4',      );
69 mpc.set('NUM_INTEGRATOR_STEPS',      N,                  );
70 mpc.set('QP_SOLVER',                'QP_QPOASES',     );
71 mpc.set('HOTSTART_QP',              'NO',               );
72 mpc.set('LEVENBERG_MARQUARDT',     1e-10,             );
73 mpc.set('LINEAR_ALGEBRA_SOLVER',   'GAUSS_LU',        );
74 mpc.set('IMPLICIT_INTEGRATOR_NUM_ITS', 4,               );
75 mpc.set('CG_USE_OPENMP',           'YES',              );
76 mpc.set('CG_HARDCODE_CONSTRAINT_VALUES', 'NO',         );
77 mpc.set('CG_USE_VARIABLE_WEIGHTING_MATRIX', 'NO',       );

78
79
80
81 if EXPORT
82   mpc.exportCode('mav_NMPC_trajectory_tracking');
83
84 cd mav_NMPC_trajectory_tracking
85 make_acado_solver('../mav_NMPC_trajectory_tracking')
86 cd ..
87 end

```

3.3.1 ROS Integration

The ROS integration of the nonlinear controller follows the same guidelines of the linear MPC previously presented. An open source implementation of the previously presented controller can be found here. https://github.com/ethz-asl/mav_control_rw.

3.3.2 Experimental Results

The same setup used for the linear MPC is used to evaluate the nonlinear MPC. The only difference is that we are currently evaluating the controller on a more aggressive trajectory as shown in Fig. 8.

3.3.3 Robust Linear Model Predictive Control for Multirotor System

In order to evaluate the proposed RMPC, a set of test-cases were conducted using the structurally modified AscTec Hummingbird quadrotor (ASLquad) shown in Fig. 9. For the implementation of the RMPC, a software framework around ROS was developed. In particular, a set of low-level drivers and nodes handle the communication with the attitude and motor control on-board the aerial robot and therefore enabling us to provide attitude and thrust references through the RMPC. MATLAB was used to derive and compute the explicity formulation of the RMPC, while the complete explicit algorithm overviewed in Sect. 2.4.5 was implemented within a SIMULINK block. Auto-code generation was then employed to extract the C-code equivalent

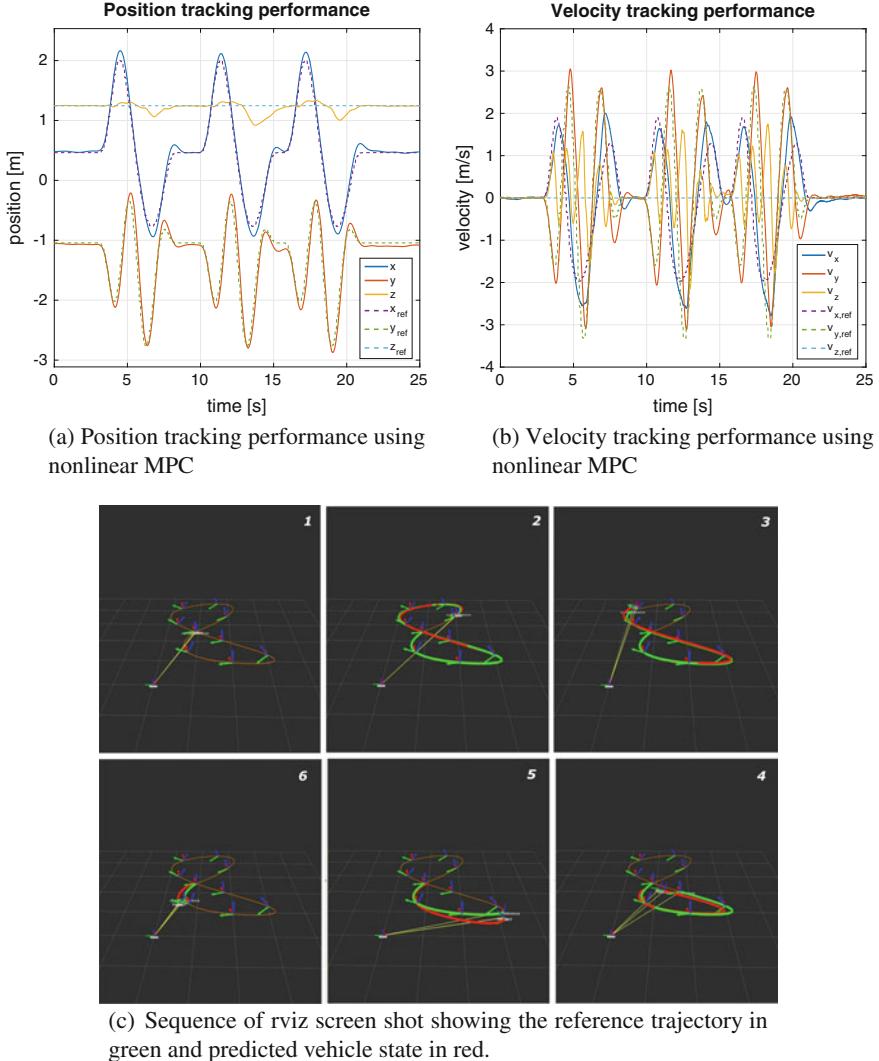


Fig. 8 Aggressive trajectory tracking performance using nonlinear MPC controller running onboard of Firefly hexacopter from Ascending Technologies

of this controller, which was then wrapped around a ROS node and integrated into the overall software framework. For the described experiments with the RMPC, full state feedback is provided through external motion capture, while an alignment step to account for relative orientation of the on-board attitude and heading estimation system also takes place.

Below we present the results on (a) trajectory tracking subject to wind disturbances, and (b) slung load operations, while further results are available at [8]. For

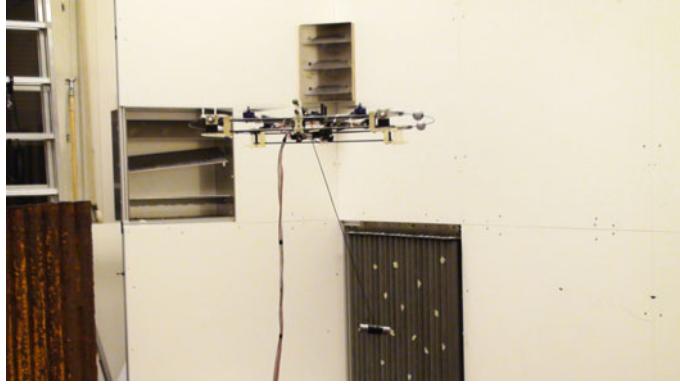


Fig. 9 Instance of an RMPC test for slung load operations

the presented experiments, the sampling time was set to $T_s = 0.08$ s, the prediction horizon was set to $N = 6$ for both of them, while the following state and input constraints were considered:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \theta \\ \phi \\ \theta^r \\ \phi^r \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \theta \\ \phi \\ \theta^r \\ \phi^r \\ \phi^r \end{bmatrix} \leq \begin{bmatrix} 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ 1/5 \text{ m/s} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \end{bmatrix}, \quad \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \dot{x} \\ \dot{y} \\ \dot{z} \\ \phi \\ \phi^r \\ \phi^r \\ \phi^r \end{bmatrix} \leq \begin{bmatrix} 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ \pi/4 \text{ rad} \\ \pi/6 \text{ rad} \\ \pi/4 \text{ rad} \\ \pi/4 \text{ rad} \\ 1.5 \text{ m/s} \\ 1.5 \text{ m/s} \\ 1/5 \text{ m/s} \\ \pi/4 \text{ rad} \\ \pi/6 \text{ rad} \\ \pi/4 \text{ rad} \end{bmatrix} \quad (44)$$

Regarding the first experimental test-case, an 80 W electric fan was pointed to the ASLquad, while the RMPC was acting in order to track a helical path despite the turbulent wind disturbance. As can be observed in Fig. 10, the tracking response remains precise and only a minor influence from the external disturbance is observed. Note that the box-constraints selection is sufficient to account –up to some extent– for the kind of disturbance as although the dynamics of the wind disturbance are unknown to the controller, its effect may be considered as bounded.

Consequently, the capabilities of performing slung load operations were considered. As shown in Fig. 11 where a forcible external disturbance is also applied onto the slung load (a hit on the load), highly precise position hold and disturbance rejection results were achieved. This is despite the fact that the slung load introduces disturbances that are phase-delayed compared to the vehicle states and the controller is not augmented regarding its state to incorporate the load's motion. For this exper-

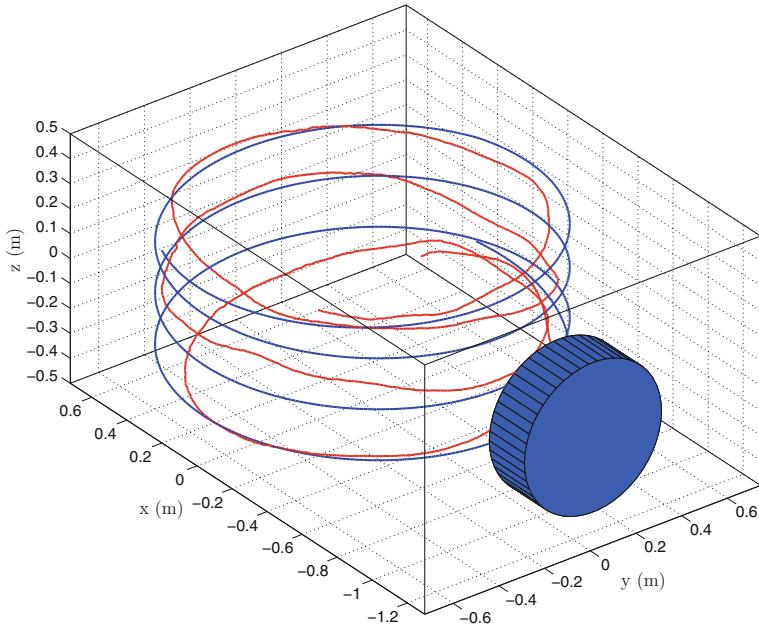


Fig. 10 Trajectory tracking subject to wind disturbances using the RMPC framework applied to the ASLquad

iment, a 0.16 kg load was utilized and was attached through a compressible string with a length of 0.65 m.

Once the capabilities of the proposed RMPC to handle slung load operations even while subjected to strong disturbances were verified, the problem of trajectory tracking during slung load operations (i.e. transportation and delivery of goods in a Search-and-Rescue scenario or as part of a product delivery mission), was examined. Figure 12 presents the results achieved when continuously tracking a square reference trajectory for 10 times while the same slung load is attached onto the quadrotor platform. As shown, efficient and robust results were derived, indicative of the capabilities of the proposed control scheme to effectively execute such operations.

In the last presented trajectory tracking experiments, a slow response was commanded. However, more agile maneuvers could also be considered even during slung load operations. To this purpose, a step reference of 1.25 m was commanded with the velocity and attitude references being set to zero. The results are shown in Fig. 13 and as depicted, a satisfactory response was derived although some overshoot is observed.

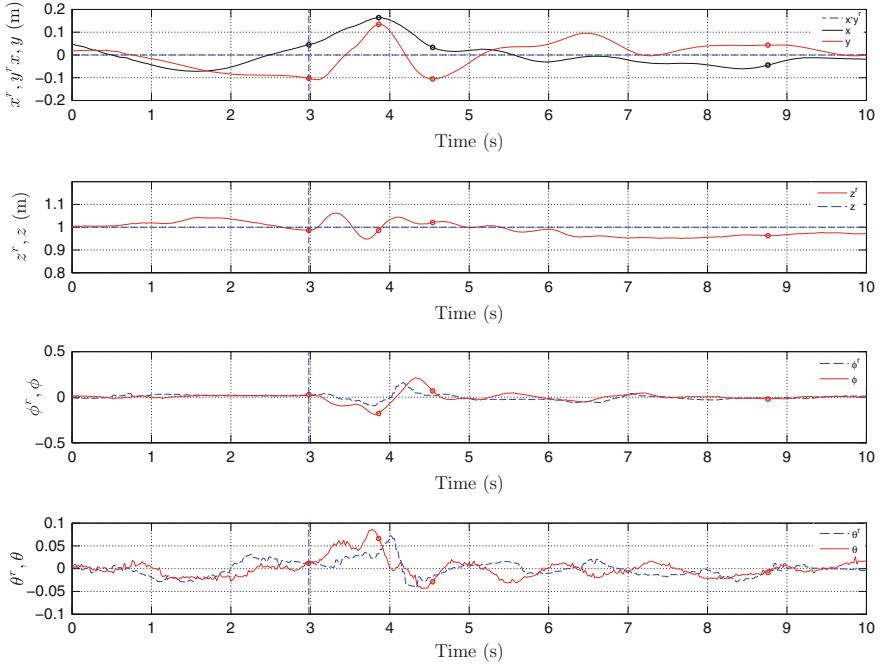


Fig. 11 RMPC performance against disturbance of the slung load which induces unpredicted disturbances on the vehicle. At time $t \approx 3$ s a forcible external disturbance (hit) is applied on the load, however the RMPC manages to quickly and accurately stabilize the vehicle

4 Model-Based Trajectory Tracking Controller for Fixed-Wing UAVs

In this section, we describe the modeling of simplified fixed-wing aircraft flight dynamics, closed-loop low-level system identification methodology, and control objective design necessary for a general-form high-level nonlinear model predictive trajectory tracking controller. As the vast majority of fixed-wing flight is conducted at fixed altitudes, we focus our presentation on planar lateral-directional position control; though, the techniques employed can easily be extended towards three-dimensional applications, assuming a suitably stabilizing longitudinal low-level controller.

4.1 Fixed-Wing Flight Dynamics and Identification

Lateral-directional dynamics for a fixed-wing system are defined in the inertial frame \mathbb{I} , in local coordinates composed of Northing \hat{n} and Easting \hat{e} components, and body

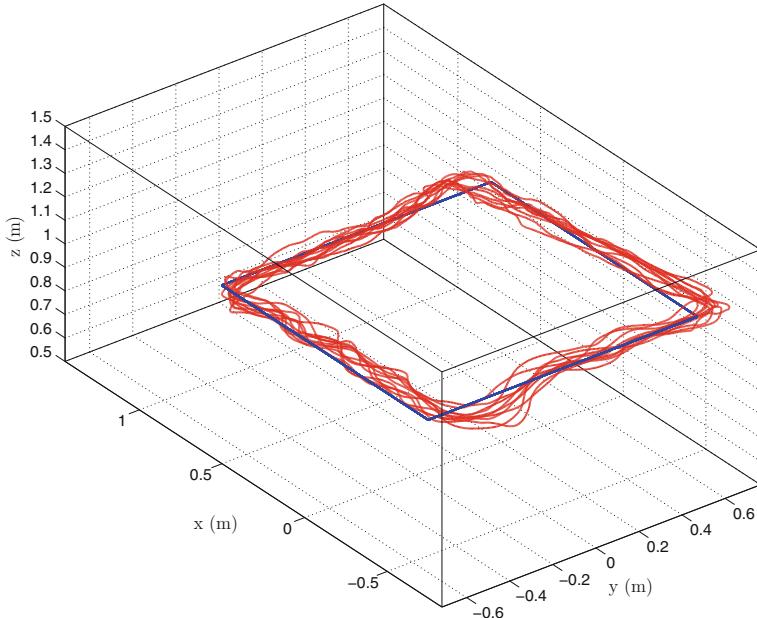


Fig. 12 Trajectory tracking during slung load operations using the ASLquad: the controller tracks the same square trajectory for 10 times with small deviations from the reference despite the significant and phase-delayed disturbances introduced from the load

frame \mathbb{B} . In reality, aerodynamic effects result in an additional “wind” frame, where the aircraft may slip, causing the airspeed vector \mathbf{v} to deviate from the body- \hat{x} -axis. In our simplified model, we will assume low-level control is designed such that slip is regulated appropriately, and we will assume wind and body frames are identical. It is worth also noting our planar assumption further entails the assumption that we fly at a fixed altitude. The dynamic equation are defined in Eq. (45).

$$\begin{aligned}\dot{n} &= V \cos \psi + w_n \\ \dot{e} &= V \sin \psi + w_e \\ \dot{\psi} &= \frac{g \tan \phi}{V} \\ \dot{\phi} &= p \\ \dot{p} &= b_0 \phi_r - a_1 p - a_0 \phi\end{aligned}\tag{45}$$

where n and e are the Northing and Easting positions, respectively, ψ is the yaw angle, V is the air-mass-relative airspeed, ϕ is the roll angle, p is the roll rate, g is the acceleration of gravity, and w_n and w_e are the Northing and Easting components of the wind vector \mathbf{w} , respectively. Note that roll ϕ and yaw ψ angles are defined about the body frame \mathbb{B} . This distinction is important when considering flight dynamics in wind, where the ground-relative flight path of the vehicle is defined as the course angle χ from North \hat{n} to the ground speed vector, \mathbf{v}_g , see Fig. 14. Additionally, note

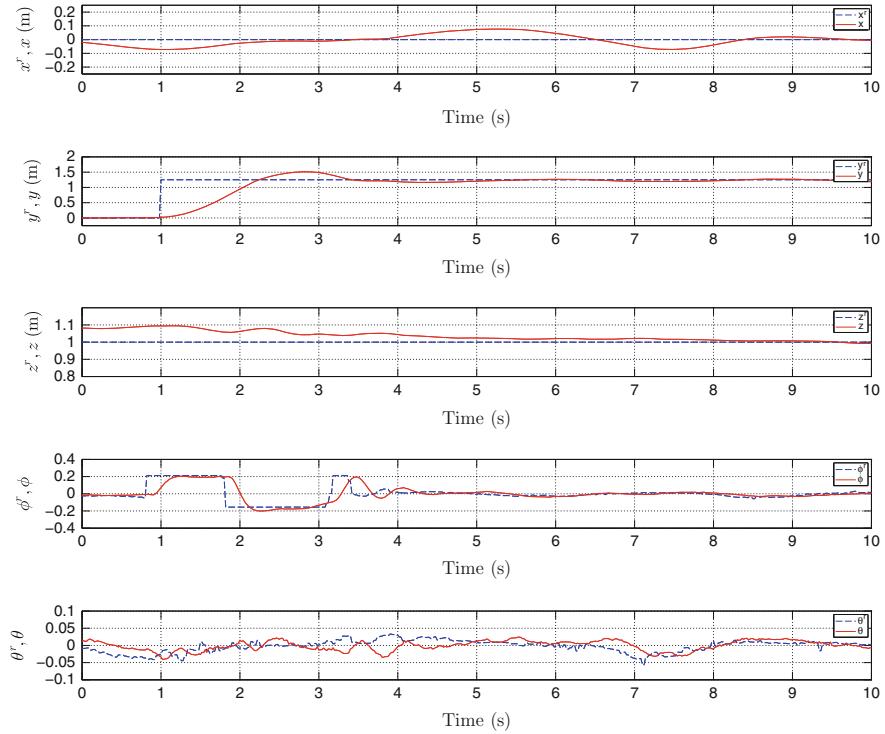
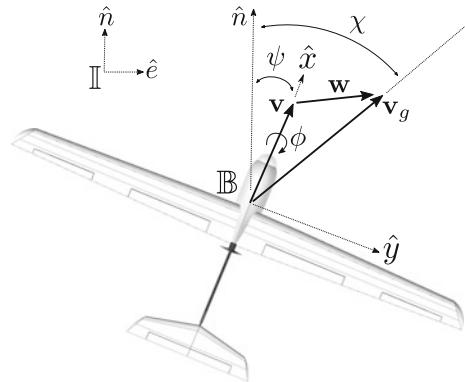


Fig. 13 Lateral step response during slung load operations: minimal overshoot is observed

Fig. 14 Fixed-wing modeling definitions



that we choose a second-order model with no zeros to describe the rolling dynamic with respect to roll references ϕ_r , where a_0 , a_1 , and b_0 are model coefficients. Higher order dynamics could be used, however we have found through the identification procedures outlined in Sect. 4.1.1, that second-order fits appropriately model the closed loop low-level autopilot attitude control response; further, each increase in order in turn increases the dimension of the control optimization problem, increasing computational expense.

4.1.1 Model Identificaiton

Here, we will outline the basic methodology for closed loop model identification on fixed-wing aircraft. Towards these ends, it is assumed a low-level autopilot with onboard state estimation and attitude, airspeed, and altitude control functionality. Such capabilities are present in commercially available autopilot hardware and software such as the Pixhawk Autopilot, an open source/open hardware project started at ETH Zurich [1]. Tuning of the low-level loops will not be discussed, though these procedures are well documented in the literature as well as in practice on the Pixhawk website.

The control architecture shown in Fig. 15 demonstrates a typical cascaded PID structure with attitude PI control and angular rate PD control. Additional compensation for slipping effects is considered for coordinated turns, i.e. a yaw damper signal, $r_r = \frac{g \sin \phi}{V}$. The TECS block indicates the use of Total Energy Control System for airspeed and altitude control, again fully implemented and documented on the pixhawk website. In three-dimensional extensions of the proposed lateral-directional nonlinear MPC, the high-level TECS block could be replaced.

The aim is to identify the closed loop low-level autopilot dynamic response when reacting to attitude commands. We will specifically discuss the roll dynamic here, however the same procedures discussed could be used for identification of pitching dynamics as well as airspeed, given that well-tuned low-level controllers are in place. Depending on the hardware used, autopilot source code could be modified for an identification option commanding repeatable excitation inputs, or in the case of

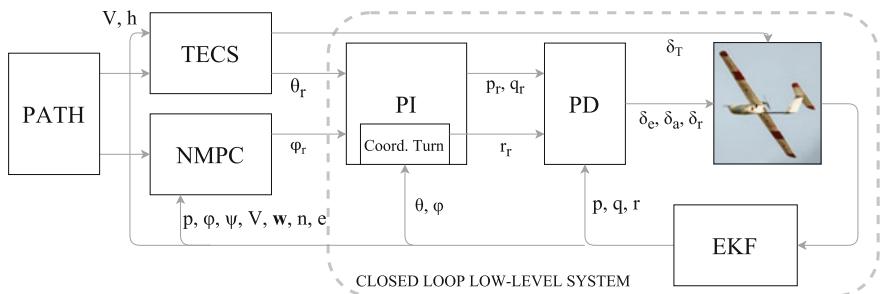
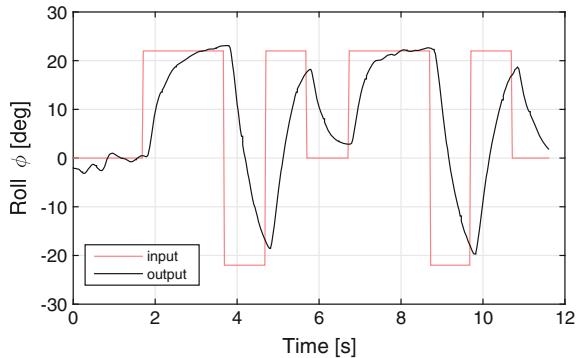


Fig. 15 Fixed-wing control architecture

Fig. 16 Two concatenated 2-1-1 maneuvers from a flight experiment with a fixed-wing UAV



utilizing a Linux operating system, a simple ROS node could be written to generate the same. For roll channel identification, pitch references to the low level controller should be held constant for holding altitude. Depending on the operational airspeed, the pitch reference may vary. 2-1-1 maneuvers, a modified doublet input consisting of alternating pulses with pulse widths in the ratio 2-1-1, are recommended, see Fig. 16. Morelli [17] demonstrated that flight time required for the 2-1-1 maneuver is approximately one-sixth of the time required for the standard frequency sweep, enabling one to gather more data in the same flight time, which is often limited by battery capacity. At the same time, concatenated 2-1-1 maneuvers make for suitable identification inputs for both frequency and time domain system identification approaches, on par with frequency sweeps.

It is good practice to perform all identification experiments on calm days with no wind and to persistently excite the control inputs. Data collection should consist of several 2-1-1 maneuvers for each of several setpoint magnitudes throughout the desired range. A similar set of data for both training and validation is required. To test the generalizability of the fit parameters, it is also worthwhile to include non-2-1-1 maneuvers in the validation set, e.g. arbitrary piloting (still within attitude control augmentation mode), to test the generalizability of the fit parameters. The authors provide a set of Matlab scripts that can perform the parameter identification after data collection and format conversion. Further literature on closed loop system identification for fixed-wing vehicles can be found in [17, 18].

4.2 Nonlinear MPC

In this section, we formulate a nonlinear MPC for general high-level fixed-wing lateral-directional trajectory tracking utilizing the model developed in the previous section. The generalized form involves augmentation of the vehicle model with trajectory information, including discretely defined sequential tracks. We use the ACADO

Toolkit [16] for generation of a fast C code based nonlinear solver and integration scheme. The optimization problem OCP takes the continuous time form

$$\begin{aligned} \min_{\mathbf{U}} \quad & \int_{t=0}^T \|h(\mathbf{x}(t), u(t)) - \mathbf{y}^{ref}(t)\|_{\mathbf{Q}}^2 dt + \|m(\mathbf{x}(T)) - \mathbf{y}^{ref}(T)\|_{\mathbf{P}}^2 \\ \text{subject to} \quad & \dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}(t), u(t)); \\ & u(t) \in \mathcal{U}_C \\ & \mathbf{x}(t) \in \mathcal{X}_C \\ & \mathbf{x}(0) = \mathbf{x}(t_0). \end{aligned} \quad (46)$$

The state vector is defined as $\mathbf{x} = (n, e, \phi, \psi, p, x_{sw})^T$, and control input $u = \phi_r$, where the augmented state x_{sw} is a switch state used within the horizon in the case that desired trajectories are piece-wise continuously, or generally discretely, defined. The switch variable has no dynamic until a switch condition is detected within the horizon, at which point an arbitrary differential α is applied for the remainder of the horizon calculations, see Eq. (47). This assumes we only consider a maximum of two track segments in a given horizon, and ensures the optimization does not revert back to a previous track after switching within the horizon.

$$\dot{x}_{sw} = \begin{cases} \alpha & \text{switch condition met } \parallel x_{sw} > \text{threshold} \\ 0 & \text{else} \end{cases} \quad (47)$$

A general tracking objective is constructed for minimizing the position error to a given track,

$$e_t = (\mathbf{d} - \mathbf{p}) \times \bar{\mathbf{T}}_d \quad (48)$$

where $\bar{\mathbf{T}}_d$ is the tangent unit tangent vector at the closest point \mathbf{d} from the UAV position \mathbf{p} to the current path \mathcal{P} , while also aligning the vehicle course with the desired trajectory direction, i.e. minimize

$$e_\chi = \chi_d - \chi \quad (49)$$

where $\chi_d = \text{atan2}(\bar{T}_{d_e}, \bar{T}_{d_n}) \in [-\pi, \pi]$. Here, we use the atan2 function from the standard C math library. See also Fig. 17. Use of this general objective formulation allows inputting any path shape, so long as the nearest point from the UAV position can be calculated and a direction of motion along the path is given for minimization throughout the horizon.

A relevant example of switching trajectories is that of Dubins Car, or Dubins Aircraft in the three-dimensional case, path following, see [19]. Dubins paths can be used to describe the majority of desired flight maneuvers in a typical fixed-wing UAV mission. Further, using continuous curves such as arcs and lines allow time-invariant

Fig. 17 Trajectory tracking objectives

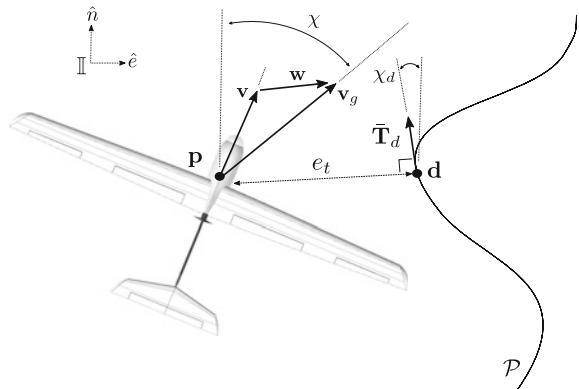
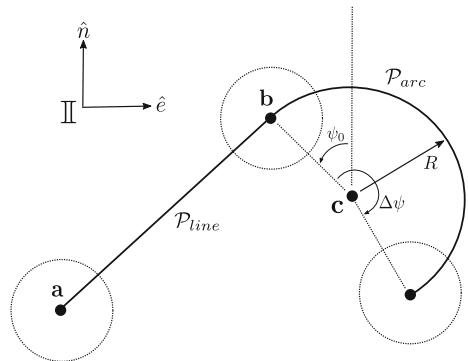


Fig. 18 Dubins path definitions



trajectory tracking, as oppose to desired positions in time, a useful quality when only spatial proximity to the track is desired and timing is less important; for instance, if energy conservation is required and a single low airspeed reference is given to be tracked. For the remainder of the section, we will consider Dubins segments as path inputs to the high-level controller, though it should be noted that the objective formulation is not limited to these.

Using the definitions in Eqs. (48) and (49), we formulate the objective vector $\mathbf{y} = (e_t, e_\chi, \phi, p, \phi_r)^T$. We assume a fixed air-mass-relative airspeed V and two-dimensional wind vector \mathbf{w} , held constant throughout the horizon. These values are estimated and input to the optimization as online data. Also input as online data, the current and next sets of Dubins path parameters \mathcal{P}_{cur} , \mathcal{P}_{next} , where line parameters include $\mathcal{P} = \{type = 0, \mathbf{a}, \mathbf{b}\}$, \mathbf{a} and \mathbf{b} are two waypoints defining a straight segment, and arc parameters include $\mathcal{P} = \{type = 1, \mathbf{c}, R, dir, \psi_0, \Delta\psi\}$, \mathbf{c} is the center point of the arc, R is the radius, dir is the loiter direction, and ψ_0 , is the heading pointing towards the entrance point on the arc, and $\Delta\psi$ is the arclength traveled. The path segments are managed and switched based on an acceptance radius and heading direction criteria, see Fig. 18.

All references are set to zero, except for the control input references. As the continuous time formulation does not allow slew rate penalties, it is possible to instead store the previous control horizon for input as weighted control references in the next nonlinear MPC iteration. This is done to avoid bang–bang control action when the nonlinear MPC iterations occur at relatively large steps, for fixed-wing we run the high-level controller at either 10 or 20Hz. As only the second step in the current control horizon is sent to the vehicle for tracking, the early horizon is penalized more heavily than the latter horizon values. This can be accomplished by weighting the horizon of controls with a decreasing quadratic function. More insights on the control formulation may be found in [20].

The following MATLAB script may be run to generate the ACADO solver in C code. Note when dealing with discontinuous functions in the model formulation, external models with accompanying external jacobians must be supplied, written in C. Further, if the objective function contains discontinuous functions, this may also be input externally. Here, we implement a numerical jacobian, though one could find individual analytic expressions for each discontinuous case and supply them to the code generation.

```

1 clc;
2 clear all;
3 close all;
4
5 Ts = 0.1; % model discretization
6 N = 40; % horizon length
7
8 % STATES -----
9 DifferentialState n; % (northing) [m]
10 DifferentialState e; % (easting) [m]
11 DifferentialState phi; % (roll angle) [rad]
12 DifferentialState psi; % (yaw angle) [rad]
13 DifferentialState p; % (roll rate) [rad/s]
14 DifferentialState x_sw; % (switching state) [~]
15
16 % CONTROLS -----
17 Control phi_r; % (roll angle reference) [rad]
18
19 % ONLINE DATA -----
20 OnlineData V; % (airspeed) [m/s]
21 OnlineData pparam1; % type=0 type=1
22 OnlineData pparam2; % a_n c_n
23 OnlineData pparam3; % a_e c_e
24 OnlineData pparam4; % b_n R
25 OnlineData pparam5; % b_e dir
26 OnlineData pparam6; % — psi0
27 OnlineData pparam7; % — dpsi
28 OnlineData pparam1_next; % type=0 type=1
29 OnlineData pparam2_next; % a_n c_n
30 OnlineData pparam3_next; % a_e c_e
31 OnlineData pparam4_next; % b_n R
32 OnlineData pparam5_next; % b_e dir
33 OnlineData pparam6_next; % — psi0

```

```

34 OnlineData pparam7_next;           % —      dpsi
35 OnlineData wn;                  % (northing wind) [m/s]
36 OnlineData we;                  % (easting wind) [m/s]
37
38 % OPTIMAL CONTROL PROBLEM -----
39
40 % lengths
41 n_X = length(diffStates);       % states
42 n_U = length(controls);         % controls
43 n_Y = 4;                       % state/outputs objectives
44 n_Z = 1;                       % control objectives
45 n_OD = 17;                     % online data
46
47 % weights
48 Q = eye(n_Y+n_Z,n_Y+n_Z);
49 Q = acado.BMatrix(Q);
50
51 QN = eye(n_Y,n_Y);
52 QN = acado.BMatrix(QN);
53
54
55
56 % optimal control problem
57 acadoSet('problemname','nmpc');
58 ocp = acado.OCP( 0.0, N*Ts, N );
59
60 % minimization
61 ocp.minimizeLSQ( Q, 'evaluateLSQ' );
62 ocp.minimizeLSQEndTerm( QN, 'evaluateLSQEndTerm' );
63
64 % external model
65 ocp.setModel('model','rhs','rhs_jac');
66 ocp.setDimensions( n_X, n_U, n_OD, 0 );
67
68 ocp.subjectTo( -35*pi/180 <= phi_r <= 35*pi/180 );
69
70 setNOD(ocp, n_OD);
71
72 % export settings
73 nmpc = acado.OCPexport( ocp );
74 nmpc.set('HESSIAN_APPROXIMATION','GAUSS_NEWTON');
75 nmpc.set('DISCRETIZATION_TYPE','MULTIPLE_SHOOTING');
76 nmpc.set('SPARSE_QP SOLUTION','FULL_CONDENSING');
77 nmpc.set('INTEGRATOR_TYPE','INT_IRK_GL4');
78 nmpc.set('NUM_INTEGRATOR_STEPS', N );
79 nmpc.set('QP_SOLVER','QP_QPOASES');
80 nmpc.set('HOTSTART_QP','YES');
81 nmpc.set('LEVENBERG_MARQUARDT', 1e-10 );
82 nmpc.set('GENERATE_MAKE_FILE','YES');
83 nmpc.set('GENERATE_TEST_FILE','YES');
84 nmpc.set('GENERATE_SIMULINK_INTERFACE','YES');
85 nmpc.set('CG_HARDCODE_CONSTRAINT_VALUES','YES');
86 nmpc.set('CG_USE_VARIABLE_WEIGHTING_MATRIX','YES');

```

```

87 % export
88 copyfile('.. / acado / external_packages / qpoases' , ...
89   'export_nmpc / qpoases')
90 nmpc.exportCode('export_nmpc');
91
92 cd export_nmpc
93 make_acado_solver('.. / acado_nmpc_step' , 'model.c')
94 cd ..

```

4.2.1 ROS Integration

As described in Sect. 3.2.3, we integrate the ACADO solver into a ROS node for generating control solutions in real-time on the aircraft. However, our approach for the fixed-wing UAV differs slightly from the MAV as all low-level control and state estimation is performed on the low-level autopilot's micro-controller (we use the Pixhawk Autopilot [1]). As processing power on the Pixhawk micro-controller is somewhat limited, an additional onboard ODROID-U3 computer with 1.7 GHz Quad-Core processor and 2 GB RAM, running Robotic Operating System (ROS) [3] is integrated into the platform for experimentation with more computationally taxing algorithms. High-level controllers can be run within ROS node wrappers which communicate with the Pixhawk via UART serial communication; average communication latency was observed $<3\mu\text{s}$. The nonlinear MPC is run within a ROS node on the ODROID-U3. MAVROS [21], an extendable communication node for ROS, is used to translate MAVLink Protocol messages containing current state estimates from the Pixhawk, and similarly send back control references from the nonlinear MPC implemented in ROS. As high-level fixed-wing dynamics are somewhat slow, we choose a fixed rate loop for control generation using a simple while loop, shown as an example in the following code excerpt.

```

1 while (ros::ok()) {
2
3   /* empty callback queues */
4   ros::spinOnce();
5
6   /* nmpc iteration step */
7   ret = nmpc.nmpcIteration();
8
9   if (ret != 0) {
10     ROS_ERROR("nmpc_iteration: error in qpOASES QP solver.");
11     return 1;
12   }
13
14   /* sleep */
15   nmpc_rate.sleep();
16 }

```

Note the nonlinear MPC iteration step is only called once per loop, and the sleep function regulates the timing of the loop. The `ros::spinOnce()` updates any

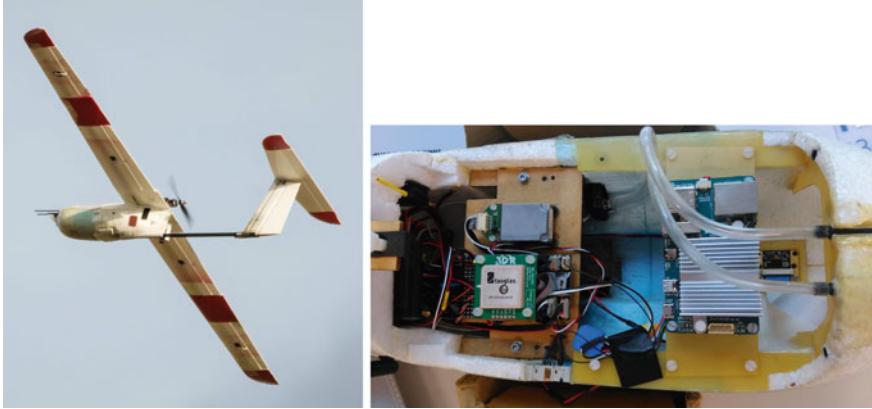


Fig. 19 Fixed-wing test platform

subscriptions with the most recent state estimates for use in the controller, and control action is subsequently published as a `geometry_msgs::TwistStamped` message for processing within the MAVROS attitude setpoint plugin. This plugin is integrated for off-board control functionality within the Pixhawk standard firmware. Messages from the pixhawk are streamed at a rate of ~ 40 Hz, and it is reasonable to assume the callback functions will contain up-to-date values in their queues at every nonlinear MPC iteration; as mentioned previously, typical high-level control rates for fixed-wing vehicles are often 10 or 20 Hz. All augmented states, i.e. not directly measured or estimated, used within the controller are also stored and kept for the next iteration.

4.2.2 Experimental Results

System identification, controller design, and flight experiments described in this section are performed on a small, 2.6 m wingspan, light-weight 2.65 kg, low-altitude, and hand-launchable fixed-wing UAV, Techpod, see Fig. 19.

Two flight experiments were conducted using the described framework. A horizon length of $N = 40$, or 4 s was used with objective weights $Q_{y_{diag}} = P_{diag} = [0.01, 10, 0.1, 0.01, 10]$, $Q_u = 100$. The discretization time step within the horizon is $T_{step} = 0.1$ s, and the nonlinear MPC is iterated every 0.05 s. The average solve time for the nonlinear MPC running on the ODROID-U3 was observed to be ~ 13 ms. Both experiments took place during very calm conditions, and the wind speed was negligible.

In Fig. 20, Techpod is commanded towards a box pattern until returning to a loiter circle. Minimal overshoot is observed, considering the set acceptance radius of 35 m, and convergence within less than 1 m of position error is observed for each

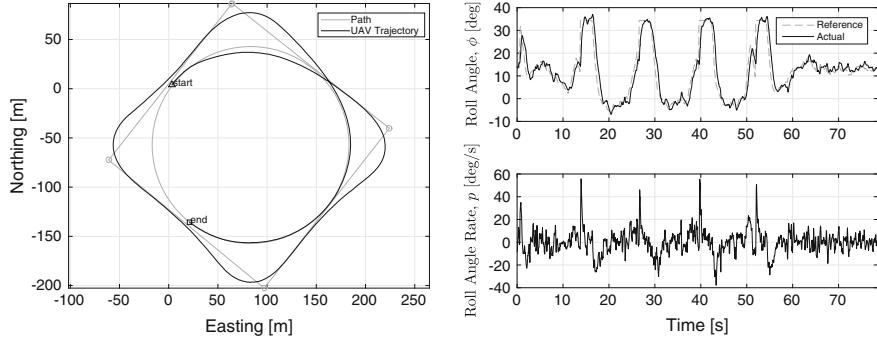


Fig. 20 Flight experiment: box tracking

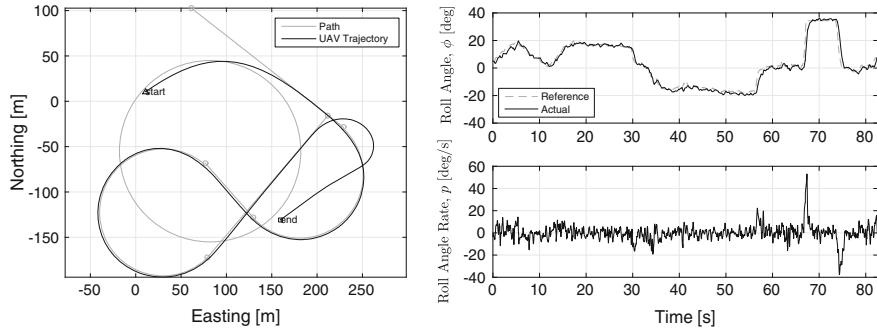


Fig. 21 Flight experiment: Dubins tracking

line segment and the final loiter circle. The commanded and actual roll angles as well as the roll rate, are both kept within acceptable bounds.

In Fig. 21, an arbitrary sequence of Dubins segments were given to the high-level nonlinear MPC. Again, good convergence to the path is seen, with acceptable state responses. Position error of less than 3 m was observed after convergence to the path. The end of the shown flight path is stopped just before converging to the final loiter due to rain fall starting during the flight experiment and manual take-over of the aircraft for landing.

5 Conclusion

In this chapter we presented an overview of many model-based control strategies for multiple classes of unmanned aerial vehicles. The strategies presented are: Robust MPC for multi-rotor system, Linear MPC for multi-rotor system and Nonlinear MPC for multi-rotor system and fixed-wing UAVs. These control strategies have

been evaluated in real experiments and performance evaluation has been shown in this chapter. The presented controllers are available as open source ROS package on https://github.com/ethz-asl/mav_control_rw for rotary wing MAVs and https://github.com/ethz-asl/mav_control_fw for fixed wing MAVs.

References

1. Px4 autopilot. <https://pixhawk.org>.
2. Navio autopilot. <https://emlid.com>.
3. Robot operating system. <http://www.ros.org>.
4. Mayne, D.Q., J.B. Rawlings, C.V. Rao, and P.O. Scokaert. 2000. Constrained model predictive control: Stability and optimality. *Automatica* 36 (6): 789–814.
5. Boyd, S., and L. Vandenberghe. 2004. *Convex Optimization*. Cambridge: Cambridge University Press.
6. Borrelli, F., A. Bemporad, and M. Morari. 2015. *Predictive Control for Linear and Hybrid Systems*. Cambridge: Cambridge University Press.
7. Ferreau, H., C. Kirches, A. Potschka, H. Bock, and M. Diehl. 2014. qpOASES: A parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation* 6 (4): 327–363.
8. Alexis, K., C. Papachristos, R. Siegwart, and A. Tzes. 2015. Robust model predictive flight control of unmanned rotorcraft. *Journal of Intelligent & Robotic Systems* 1–27.
9. Loefberg, J. 2003. Minimax approaches to robust model predictive control. Ph.D. dissertation Linkoping, Sweden: Linkoping University.
10. Cannon, M., S. Li, Q. Cheng, and B. Kouvaritakis. 2011. Efficient robust output feedback mpc. In *Proceedings of the 18th IFAC World Congress*, vol. 18, 7957–7962.
11. Kvasnica, M. 2009. *Real-Time Model Predictive Control via Multi-Parametric Programming: Theory and Tools*. Saarbrücken: VDM Verlag.
12. Mattingley, J., and S. Boyd. 2012. Cvxgen: A code generator for embedded convex optimization. *Optimization and Engineering* 13 (1): 1–27.
13. Ljung, L. 1999. *System identification - Theory for the User*. Englewood Cliffs: Prentice-Hall.
14. Lynen, S., M.W. Achtelik, S. Weiss, M. Chli, and R. Siegwart. 2013. A robust and modular multi-sensor fusion approach applied to mav navigation. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 3923–3929. IEEE.
15. vicon systems. <http://www.vicon.com>.
16. Houska, B., H. Ferreau, M. Vukov, and R. Quirynen. 2009–2013. ACADO Toolkit User’s Manual. <http://www.acadotoolkit.org>.
17. Morelli, E.A. 2003. Low-order equivalent system identification for the tu-144ll supersonic transport aircraft. *Journal of guidance, control, and dynamics* 26 (2): 354–362.
18. Luo, Y., H. Chao, L. Di, and Y. Chen. 2011. Lateral directional fractional order (π) control of a small fixed-wing unmanned aerial vehicles: controller designs and flight tests. *Control Theory & Applications, IET* 5 (18): 2156–2167.
19. Beard, R.W., and T.W. McLain. 2013. Implementing dubins airplane paths on fixed-wing uavs. In *Contributed Chapter to the Springer Handbook for Unmanned Aerial Vehicles*.
20. Stastny, T., A. Dash, and R. Siegwart. 2017. Nonlinear mpc for fixed-wing uav trajectory tracking: Implementation and flight experiments. In *AIAA Guidance, Navigation, and Control (GNC) Conference*.
21. MAVROS. 2016. <http://wiki.ros.org/mavros>.

Designing Fuzzy Logic Controllers for ROS-Based Multirotors

**Emanoel Koslosky, André Schneider de Oliveira,
Marco Aurélio Wehrmeister and João Alberto Fabro**

Abstract This chapter presents a tutorial on using an open-source ROS package for implementing control systems based on Fuzzy Logic. Such a package has been created to facilitate the development of fuzzy control systems along with ROS technology and infrastructure. A step-by-step tutorial discusses how to develop a set of distributed and interconnected fuzzy controllers using the proposed ROS package. A fuzzy control system that controls the movement of an unmanned multirotor (specifically a hexacopter) is presented as case study. The behavior of this control system is demonstrated by means of a commercial robotics simulation environment named V-REP. One scenario is used to illustrate the fuzzy control system steering the movement of a virtual hexacopter carrying an attached loose payload, i.e. such a loose payload forms a pendulum. In this case study, one can see the hexacopter flight after receiving commands to fly to distinct positions within the scenario. It is important to highlight that, in order to be able to perform this tutorial, the reader must use ROS Indigo Igloo and V-REP PRO EDU version V3.3.0 both running on Ubuntu 14.04.4 LTS.

Keywords ROS · Multirotor · Fuzzy logic · Simulation

The source code and examples discussed in this chapter are available as a catkin package published in [1]

E. Koslosky (✉) · A.S. de Oliveira · M.A. Wehrmeister · J.A. Fabro
Advanced Laboratory of Embedded Systems and Robotics (LASER), Federal University of Technology—Parana (UTFPR), Av. Sete de Setembro 3165, Curitiba 80230-901, Brazil
e-mail: ekosky@gmail.com

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

M.A. Wehrmeister
e-mail: wehrmeister@utfpr.edu.br

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

1 Introduction

Recent technology advances have led to a cost reduction in electronic and electro-mechanical components, providing new capabilities to small electromechanical aircrafts such as multirotor helicopters (also known as drones). Such devices are being applied in many distinct application fields, such as video recording, plantation inspections, search-and-rescue assistance, military and civil surveillance applications, among others. Some of these new applications demand multirotor helicopters that fly autonomously as presented in [2, 3]. For that, additional computing systems must be embedded into an autonomous multirotor helicopter, in addition to movement and stabilization control systems, in order to provide higher level capabilities to support the mission accomplishment. Unmanned Aerial Vehicles (UAV) are the preferred choice for these applications due to cost reductions obtained from eliminating the need of high-skilled and trained pilots. It is important to highlight that, in this text, the term “*multirotor*” is used as a synonym for “*multirotor helicopter*”.

The multirotor rotors can be organized in different ways, varying in the amount of rotors, as well as their positions onto the aircraft frame. The so-called quadcopter is a multirotor equipped with four rotors. It is the most common multirotor. However, its characteristic may limit some applications, e.g. a payload transportation from one point to another. Recently, other multirotor topologies have become popular such the hexacopter [4] that is equipped with six rotors.

In order to provide a stable flight for UAVs, hybrid control approaches (parallel, cascade) with multiple PID controllers are commonly used [5, 6]. Although these methods perform the system control in a proper way, they require a precise mathematical formulation as well as the identification of UAV dynamics, in order to stabilize the system while minimizing disturbances [7].

Adaptive algorithms can be applied to control multivariable systems (such as UAV flying control system) more efficiently than classical strategies. In [8], an approach based on artificial neural networks is presented to control the trajectory of UAV flight. A genetic algorithm is applied to control the flight of a hexacopter in [9], where as a fuzzy logic method has been proposed to control the position of a hexacopter in [10]. The main focus of these previous works is on the UAV stabilization in the presence of linear disturbances. However, these works do not consider nonlinear disturbances, such as the ones introduced when the UAV carries a variable or loose payload. In a previous work [11], we created a fuzzy logic controller to control the movements of a hexacopter and also to deal with nonlinear disturbances. Such a fuzzy control system was created to provide a robust and flexible controller that is able to keep the hexacopter stability when moving or hovering, even when it carries a free or loose payload that changes its center of gravity. It is important to highlight that, due to space constraints, this chapter does not provide a in-depth discussion on this fuzzy control system. Interested readers are referred to [11] in order to obtain details on the design of such a control system.

This chapter discusses a ROS-based implementation of our fuzzy control system in terms of an open-source ROS-based fuzzy logic library designed to control

multirotors. Specifically, the proposed fuzzy library has been implemented to be used within a `roscpp` Node. The main goal is to present a step-by-step tutorial on designing fuzzy based controllers for mobile robots (focusing on UAVs) using ROS features [12]. This tutorial is intended to be followed by beginner level ROS users. It discusses how ROS is used to receive signals from sensors and also to send commands to actuators by means of the publisher/subscriber mechanism. Moreover, the tutorial shows how to integrate a different robot simulator named V-REP [13] with the fuzzy control system implemented with the proposed library. Thus, the engineers may perform a round-trip engineering process¹ by integrating the developed fuzzy control system with a virtual environment or the real hardware seamlessly.

It is important to highlight that, as already mentioned, the tutorial described in this chapter is aimed to beginner level users of ROS. Thus, in order to correctly understand its content, the reader should be familiar with Linux and C/C++ programming language, as well as he/she should have some basic ROS understanding. The beginner level tutorials [12] should be sufficient to understand the presented approach (tutorial #16 presents the concepts about nodes, messages, publishers and subscribers). Although it might be a good idea to follow the V-REP BubbleRob tutorial [14], the short introduction given here would be enough for allowing the experimentation. Moreover, a tutorial about V-REP and ROS integration is presented in [15]. The versions of the software used in this tutorial are: (i) Operating System: Ubuntu 14.04.4 LTS; (ii) ROS Indigo Igloo; (iii) V-REP PRO EDU version V3.3.0.

The remainder of this chapter is organized as follows. Section 2 presents a brief overview of multirotors features and movements. Section 3 summarizes our fuzzy control system for a hexacopter. Section 4 presents the open-source ROS library that provides the services supplied within our fuzzy logic library. Section 5 discusses how to use a different robotics simulation environment along with ROS and our library to perform experiments on fuzzy control system, including the discussion of a case study that illustrates the concepts and technologies discussed in this tutorial. Finally, Sect. 6 concludes this chapter by discussing some final remarks.

2 Brief Overview of Multirotors

This section provides a description on how multirotors perform their movements. An empirical discussion is presented rather than a formal modeling of multirotor dynamics, in order to provide a practical view similar to a human pilot controlling the multirotor by means of a radio control system (i.e. RC controller). Formal models of multirotor dynamics can be obtained, for instance, in [5–7].

Initially, it is worth mentioning that a multirotor can present various distinct configurations, i.e. multirotors may present various topologies. In summary, multirotor topology varies in rotors number as well as the rotors position. Regarding the number of rotors, a multirotor may have from three, four, six or eight rotors, namely, tricopter,

¹This chapter does not intend to propose or discuss any concrete round-trip engineering process.

quadcopter, hexacopter and octocopter, respectively. The most common multirotor is the quadcopter, although the hexacopter is recently becoming popular due to its good trade-off among cost, flight robustness, fault-tolerance, and capacity of flying with heavier payloads.

On the other hand, these rotor may be positioned in distinct topologies regarding the front/rear of the multirotor.

- “*X*” topology presents two rotors on both front and rear. One rotor is positioned on the right-hand side and the other one on the left-hand side of front/rear. This topology can be used with quad-, hexa-, and octocopters.
- “*I*” or “+” topology presents one rotor positioned on the front and one rotor position on the rear. The remainder rotors are distributed evenly on the right-hand and left-hand sides of the multirotor. This topology can be used with quad-, hexa-, and octocopters.
- “*H*” topology is similar to the “*X*” topology, i.e. two rotors on both the front and rear. However, the arms of the aircraft frame form an “*H*” rather than an “*X*”. This topology can be used with quad- or hexacopters.
- “*Y*” topology presents two rotors on either front or rear, and one rotor on the opposite side. This topology can be used with tri- or hexacopters. In hexacopters, the counter rotating propellers are placed one on top of another.

It is important to highlight that both the amount of rotors and their positioning onto the aircraft frame influence how a multirotor performs its movements, and hence, how the movement control system is designed. In the remainder of this section, the hexacopter “+” topology is used to illustrate the multirotor movements.

A multirotor moves on three dimensions along X, Y and Z axes as shown in Fig. 1. In summary, the rotors create thrust that allows for pitch, roll, yaw, uplift and downfall movements. Thus, by activating the rotors accordingly, it is possible to control the hexacopter movement along X, Y and Z axes. In other words, by speeding up or slowing down some rotors, it is possible to move the multirotor towards the desired direction on each axis.

Pitch is the multirotor movement towards either forward or backward. Controlling pitch angle implies on the control of multirotor rotation² on Y-axis as shown in Fig. 2. The multirotor moves forward when the rear rotor spins faster than the front rotor; similarly, it moves backward when front rotor spins faster than the rear rotor. The difference in these rotors spinning produces an unbalanced thrust on each rotor, rotating the multirotor around its Y-axis leading to an horizontal movement along the X-axis. The multirotor slows down the movement when the rotor positioned on the movement direction spins faster than the other rotor, i.e. this decreases the pitch angle.

Roll is the multirotor movement towards right-hand or left-hand side. Controlling roll angle implies on the control of multirotor rotation (see footnote 2). on X-axis as shown in Fig. 3. The multirotor moves sideways when the rotor on one side spins faster

²Figures 2, 3, 4 and 5 depict the inertial frame at the right-bottom corner of the figures. It is important to note that this inertial frame is used in both V-REP environment and ROS representation.

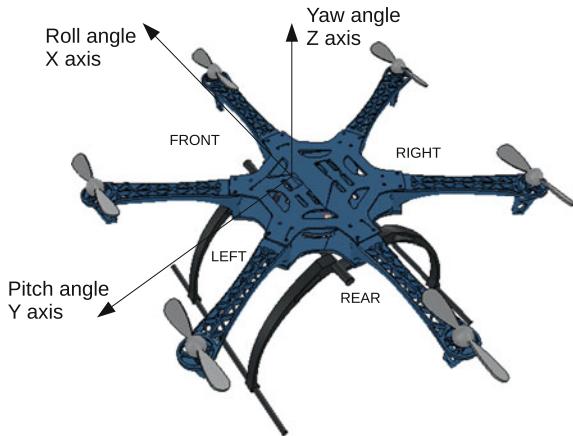


Fig. 1 Hexacopter movements: (i) roll is the rotation on X axis; (ii) pitch is the rotation on Y axis; and (iii) yaw is the rotation on Z axis. The *arrow* indicates positive direction

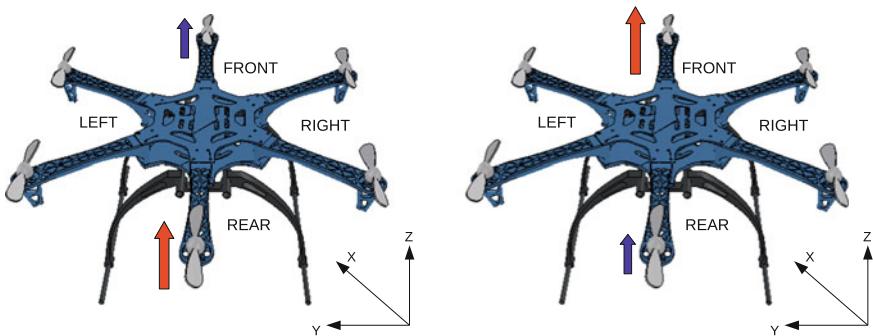


Fig. 2 Hexacopter pitch movement: rotation around Y-axis

than the ones on the other side. The difference in these rotors spinning produces an unbalanced thrust on right-hand or left-hand side, rotating the multirotor around its X-axis leading to a horizontal movement along the Y-axis. Likewise forward/backward movement, the multirotor slows down the movement when the rotors positioned on the movement direction spins faster than the other ones, i.e. this decreases the roll angle.

Yaw is the deviation of the multirotor head (i.e. its front orientation) towards either right or left. Controlling yaw angle implies on the control of multirotor rotation on Z-axis as shown in Fig. 4. For that, interleaved rotors must spin faster than the other ones leading to a gyroscopic effect on the multirotor frame. It is important to note that the propellers attached to interleaved motors rotate either clockwise or counterclockwise. Therefore, in order to turn the multirotor to right-hand side direction, the clockwise rotors must spin faster. Similarly, for turning to the left-hand side direction, the

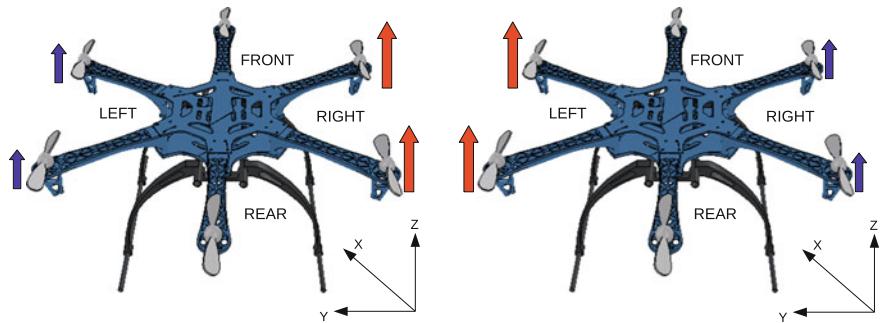


Fig. 3 Hexacopter roll movement: rotation around X-axis

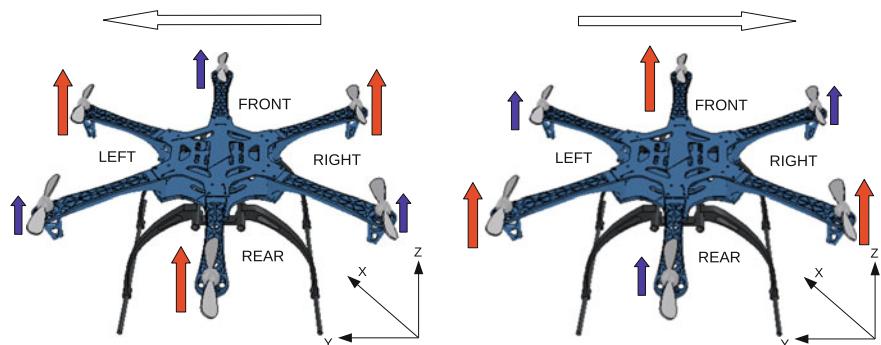


Fig. 4 Hexacopter yaw movement: rotation around Z-axis

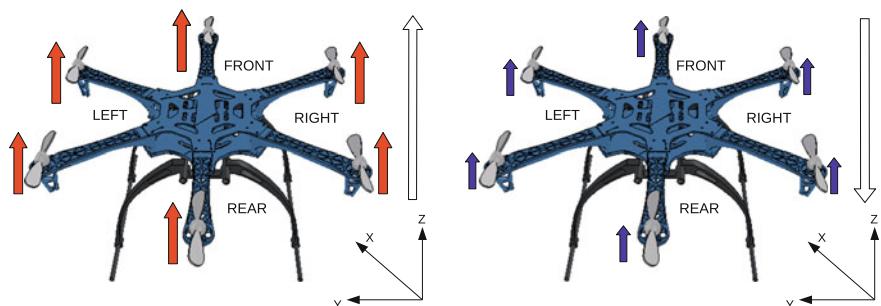


Fig. 5 Hexacopter uplift and downfall movements

counterclockwise rotors must spin faster. The multirotor does not move along X- or Y-axis; it only rotates around Z-axis.

Uplift and downfall are the multirotor movements related to the flight altitude. These movement are achieved by spinning all rotors on the same pace. The multirotor flies upwards when the lift force produced by the rotors is higher than the multicopter weight. On the other hand, the multirotor flies downwards when the lift

force produced by the rotors is lower than the multicopter weight. When lift force is equal to the weight, the multicopter hovers in the air.

Furthermore, in many application fields, a multirotor carries a payload in order to accomplish its mission. A payload attached to the multirotor body frame changes the gravity center of the whole aircraft thus affecting the way the multirotor performs its movements [16]. When the payload has a constant mass and is fixed to the body frame, the multirotor gravity center is modified but it remains on the same position. On the other hand, when the payload has a varying mass (e.g. leaking bag of sand) or it is loosely attached to the multirotor, the center of gravity changes during the flight, introducing nonlinear disturbances. A loosely attached payload forms a moving pendulum when the multirotor flies. Hence, while the pendulum is moving, center of gravity of the entire aircraft changes as well [17]. Fuzzy logic controllers have been used to deal with the moving center of gravity created by a moving pendulum [17–19].

Finally, it is worth mentioning that a multirotor is equipped with a set of sensors and actuators in order to run high- and low-level control systems. An Inertial Measurement Unit (IMU) provides a combination of gyroscope, accelerometer and compass (magnetometer) sensors. Multirotors demand three dimensional IMUs. The accelerometer detects the current acceleration rate along with X, Y and Z axes, whereas changes in rotational attributes, e.g. roll, pitch and yaw, are measured by the gyroscope. The gyroscope provides the body frame angles known as Euler angles. The magnetometer measures the magnetic field in order to assist the calibration against orientation drift. In order to obtain the absolute position of the multirotor within the environment, the Global Positioning System (GPS) sensor are used. GPS is also applied to decrease errors in position and velocity produced within an inertial navigation system. Stereo cameras or laser scanners can also be used to obtain the distance to obstacles in the environment, and hence, allowing the multirotor to avoid collisions.

3 Fuzzy Control System for Hexacopters

3.1 Brief Overview of Fuzzy Logic

This section provides an overview of the key concepts of Fuzzy Logic in order to improve the reader's understanding on our ROS-based fuzzy logic library. Interested readers must refer to [17, 20] in order to get a deeper discussion on Fuzzy Logic.

Fuzzy logic is a way to model a system using basic human interpretations, providing a method to describe both the system model and the computation of its outputs. For instance, when someone says that “I am close to a car” and “you are not close to it”, the meaning of these phrases can be diverse. How can a computer calculate how close is something? A human could answer “yes”, “not” or “almost”. However, in order to provide such an answer, it is important to consider the context to realize

the meaning of this answer. Fuzzy logic provides a way to cope with the intrinsic imprecision of these answers by means of representing imprecisions and a common reference to the meaning of concepts such as “close” and “distant”.

Let us use the altitude control of the hexacopter as an example. For instance, the operator sets the altitude to ten meters. If the hexacopter is on the ground, at altitude zero, some amount of power must be applied to all rotors until it reaches the target altitude. How much power must be applied? If the target altitude is far away from the starting position, the maximum power might be applied, and hence, the hexacopter reaches the target altitude faster but it might overshoot the desired position. On the other hand, if the applied power is minimal, the hexacopter moves slower in order to minimize the overshooting, but it takes too much time to reach the desired position. The process of mapping and adjusting the numerical values to represent human linguistic values is key to design fuzzy systems.

A Fuzzy controller comprises a set of artifacts that enables the translation from human linguistic terms into elements that are processed by computers. The following artifacts compose a fuzzy control system:

- **Linguistic Variables** are used to represent the meaning of terms that are related to input or output signals. The concept of “distance” is an example. It may have the following values: “far”, “near”, “very close” and “on”. Linguistic variables can define output values as well, e.g. the “power” to be applied on the rotors could have the absolute values, e.g. “minimum”, “middle”, “maximum”, or relative values, e.g. “much lower”, “lower”, “maintain”, “higher” and “much higher”. Therefore any system input or output can be modeled as a set of linguistic values that have meaning for an human expert.
- **Membership Function** defines the mapping linguistic variable and its linguistic values. Since linguistic variable defines a set of concepts that are understood by human experts, its linguistic values must be defined so that the computer can process them. The expert defines numeric values to represent each concept. Each membership function defines the interpretation of any numeric value, incorporating the subjective imprecision. This is done by using a geometric representation of the concept, such as a Gaussian function, or triangular/trapezoidal function. Examples of some membership functions defined in the proposed fuzzy controllers are presented in the following sections.
- **Rules** define the relationship between input and output linguistic variables. A fuzzy system comprises a set of rules that relates premises and consequences in the form: **IF** premises **THEN** consequences. Premises represent comparisons between an input linguistic variable and values of its membership function. During the inference process, the relative strength of each premise is obtained by means of a process called *fuzzyfication* and thereafter propagated to the consequences. Once all the rules are evaluated, the consequences are evaluated altogether. The quantitative output values are produced within the *defuzzyfication* process and then such values are applied to the controlled process.

It is worth pointing out that, besides defining the fuzzy set, the engineers are responsible for tuning the membership function values, in order to obtain the desired behavior for each situation.

Furthermore, the execution of fuzzy logic systems comprises the following three processes:

- **Fuzzification** is the process in which raw values obtained from input signal are compared with values of each member function, in order to find out the corresponding activation level. Usually the input signal value comes directly from a sensor reading. However, it can also be derived from some kind of calculation, such as the velocity obtained from the difference of two consecutive position readings provided by a GPS. If the raw value intercepts more than one membership function, all concepts are considered, and thus, each one presents a different activation level.
- **Rule inference** (also known as **rule evaluation**) is a process that evaluates all rules of the fuzzy system. It takes the fuzzyfied input values and evaluates the activation value of each premise, calculating the output value for each rule. Partially activated premises lead to partial activation of consequences, allowing for a “fuzzy” inference procedure.
- **Defuzzification** process obtains an exact output value (e.g. numerical value) that can be directly applied onto an actuator, e.g. the power to be applied onto the hexacopter rotors. The output strength points are used to calculate an average value. For that, the area formed by the union of each output membership function is used. There are several methods to obtain the output value. For instance, Center of Gravity (COG) defuzzification method takes into account the relative position over the horizontal axis plus the weight of the combined area.

3.2 Overview of Hexacopter Movement Control System

This section provides an overview of the fuzzy movement controller for a hexacopter, in order to explain the ROS-based open-source package presented in this chapter. Interested readers must refer to [11] in order to obtain details on the design of such a control system.

The proposed controller implements a closed loop that comprises three layers. Data produced as output of one layer is passed as input to the next layer. The proposed multi-layer fuzzy controller is based on [17] and is depicted in Fig. 6. The *Movement Fuzzy Control System box* is composed by a pre-processing phase (first layer), a set of fuzzy controllers (second layer), and post-processing phase (third layer).

As one can observe, after the post-processing phase, the control outputs are applied onto the plant by means of the hexacopter rotors that actuate on the hexacopter movement, i.e. pitch, roll, yaw. The sequence of maneuvering is depicted in Fig. 7. The sensors perceive the changes on the plant controlled variables, and hence, provide the feedback to the controller. The controller, in turn, compares these input values

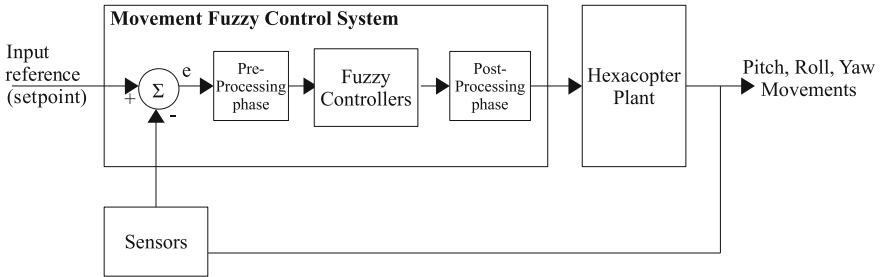


Fig. 6 Overview of the hexacopter movement fuzzy control system

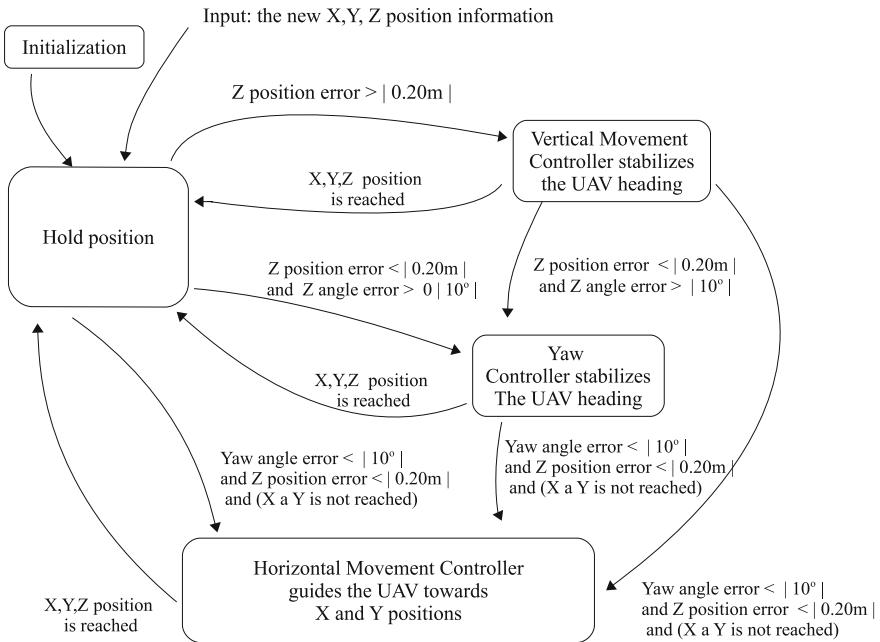


Fig. 7 FSM for sequencing the hexacopter maneuvering process. (X, Y, Z) inputs represent the new target position of the hexacopter

with the reference values established as setpoints thereby closing the control loop [17, 20].

The pre-processing phase (first layer) is responsible for acquiring data from the input sensors, processing the input movement commands, as well as calculating the controlled data used as input to the fuzzy controllers in the second layer. Before the multi-layer controller starts its execution, there is an initialization phase that is performed within the first layer. The target position is set as the current position, so that the hexacopter does not move before receiving any command. Gyroscope and accelerometer sensors are calibrated and the GPS sensor is initialized by gathering

at least four satellites. During the execution phase, the first layer is responsible to calculate the input variables to the fuzzy controllers: (i) the angular and linear distance (delta error) for X, Y, and Z axes between the current hexacopter position and the target position; (ii) the rotation and translation movement matrices in order to translate movement along X, Y and Z axes into the speed related to the ground (i.e. X and Y axis). In addition, the pre-processing phase is responsible to convert the input movement commands into setpoints for X-, Y- and Z-axis. Movements commands are composed of three values representing the positive or negative movement along X, Y and Z axes regarding the current positions, i.e. a command indicates a target relative position. Thus, when a new command is received, the first layer converts such a command to an absolute position. Then, once the control system is running, this layer uses the current GPS position to determine the error in the position of the hexacopter concerning the target position. These calculated errors in position are the inputs to the fuzzy controllers (Euler X, Euler Y and Euler Z errors).

The second layer contains five fuzzy controllers, which act on issues regarding the hexacopter movement, namely hovering, vertical and horizontal movement and heading. As mentioned, these controllers take as input the data produced in the first layer and generate output for the third layer. The generated outputs represent the actuation on the six rotors for performing pitch, roll, yaw movements for all maneuvers necessary to reach the target position. In order to provide an illustrative example, one fuzzy controller is discussed in details in the next section.

The post-processing phase (third layer) is responsible for coordinating the fuzzy controllers outputs. In order to perform a proper maneuver, the proposed multi-layer controller establishes a priority on movements needed to complete a maneuver. When a new command is received, i.e. a new target point is set, the hexacopter must firstly reach the target altitude. Then, the hexacopter must turn until its front aims the target position. Finally, the hexacopter moves horizontally towards the target position. This layers controls the position thresholds by means of output values saturation, in order to keep the hexacopter stable while flying or hovering.

3.3 Example: Design of Vertical Movement and Hovering Controller

Vertical movement and hovering fuzzy controller controls the movement on the Z-axis, i.e. it controls uplift, downfall and hovering movements. Figure 8 shows the block diagram of this controller.

This controller takes as input the vertical distance to the target position, as well as the vertical speed. The first one is the error in vertical distance (altitude error), i.e. the difference between target position and actual hexacopter position on Z-axis. The second input is the current speed calculated as a derivative information of hexacopter displacement over time.

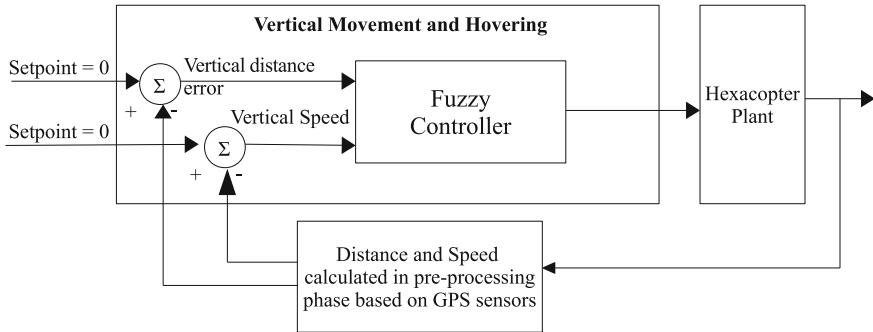


Fig. 8 Vertical movement and hovering fuzzy controller

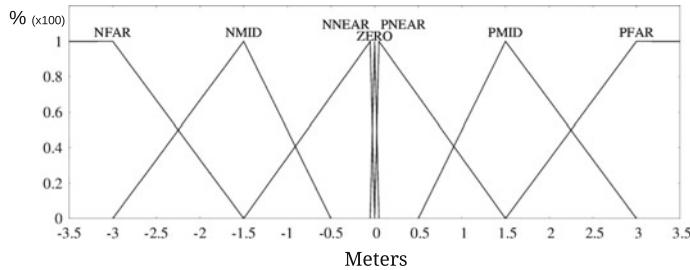


Fig. 9 Input linguistic variables and their membership functions for vertical distance

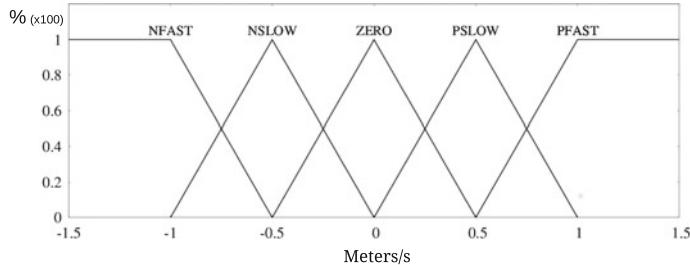


Fig. 10 Input linguistic variables and their membership functions for vertical speed

The linguistic variables for vertical distance and vertical speed are shown in Figs. 9 and 10, respectively. Letters “N” and “P” at the beginning of each membership function mean negative and positive values, respectively. In Figs. 9 and 10, values in the X-axis represent the distance from the setpoint in meters, whereas values in the Y axis indicate the activation of each membership function, varying from 0.0 (minimum) to 1.0 (maximum activation) indicating a percentage. The shape of these membership functions is the triangle, since the algorithm for calculating its area presents a low computing cost, and hence, it may be used on embedded system platform.

Table 1 Control rules of vertical navigation

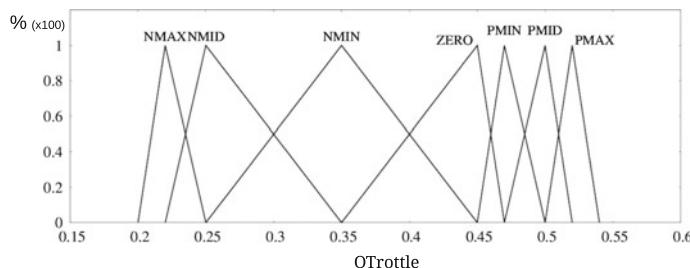
V. Dist. V. Speed	NFAR	NMID	NNEAR	ZERO	PNEAR	PMID	PFAR
NFAST	NMAX	NMID	NMIN	PMIN	PMID	PMAX	PMAX
NSLOW	NMAX	NMID	NMIN	PMIN	PMID	PMAX	PMAX
ZERO	NMAX	NMID	NMIN	ZERO	PMIN	PMID	PMAX
PSLOW	NMAX	NMAX	NMID	NMIN	PMIN	PMID	PMAX
PFAST	NMAX	NMAX	NMID	NMIN	PMIN	PMID	PMAX

Moreover, one can see the intersections of membership variables values, i.e. adjacent variables share a given range of values. This is an important characteristic of fuzzy systems and allows the modeling of smooth transitions among adjacent concepts. If there are gaps between transitions, i.e. no membership function is activated, the fuzzy control system may stop working. Another important issue is that the input membership functions must cover the complete range of input values, so that the fuzzy process can work with all possible sensor readings.

Fuzzy rules can be specified by means of clauses (e.g. if premise then consequence) or a table. The Table 1 shows the set of rules that composes the vertical movement and hovering controller. It is important to highlight that the number of rules increases as more linguistic variables and membership functions are added to the fuzzy control system.

This controller sets the output throttle variable OThrottle as result (see Fig. 11). Each value presented in Table 1 is decomposed into an amount of power applied on all rotors, increasing or decreasing the overall lift force making the hexacopter fly on higher or lower altitude. It is worth noting that the power applied on the rotors decreases along with vertical speed when the hexacopter comes closer to a target altitude.

The fuzzy control surface graphic shown in Fig. 12 provides the visualization of the input and output values of the vertical movement and hovering fuzzy controller. The altitude is maintained by means of controlling the throttle applied on all rotors

**Fig. 11** Output linguistic variables and their membership functions for OThrottle

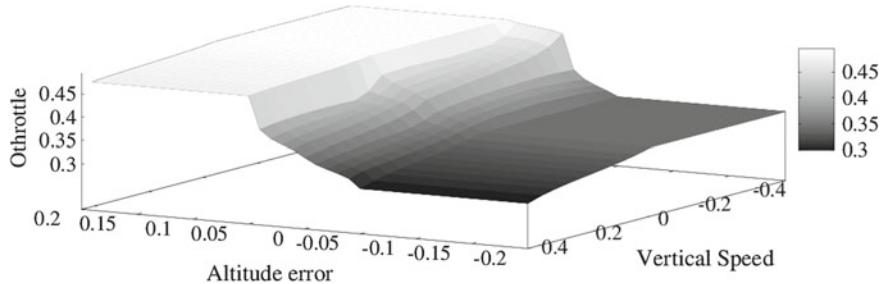


Fig. 12 Surface of fuzzy controller for vertical navigation and hovering control

simultaneously. The GPS sensor provides the current altitude information. On the other hand, the vertical speed is used to decrease the oscillation when the hexacopter reaches the desired altitude. In order to illustrate the relationship between vertical distance error (altitude error) and vertical speed, let us suppose some situations. In the first one, the altitude error is zero (i.e. the hexacopter reaches the target vertical position) and the vertical speed is positive (e.g. 0.4 or higher). This situation means that the hexacopter has reached the target altitude but it will fly beyond that position because the speed indicates the hexacopter is still flying upwards. The hexacopter vertical speed must be slowed down before reaching the target altitude, and hence, the controller must set an output value lower than ZERO. In the second situation, the altitude error is zero and the vertical speed is negative. In this situation the hexacopter is falling down, and hence, the controller must set an output value higher than ZERO in order to stop the fall. It is worth noting that the ZERO output value for OThrottle does not mean that any power is applied on the rotors; instead, it represents a minimal power value that keeps the hexacopter hovering at the current altitude.

In order to illustrate the fuzzyfication and defuzzyfication process, let us assume that the current vertical distance error (altitude error) is -0.75 m . This value is compared to the level of membership functions during the fuzzification process. After fuzzyfication, the value represents 0.28 (28%) of the NMID membership function and 0.5 (50%) of the NNEAR as shown in Fig. 13. During rules inference process, the activation of NMID and NNEAR membership function enables ten rules (see third and fourth columns of Table 1).

As one can see in Table 1, these rules combine two linguistic variables (vertical distance error and vertical speed), and hence, they define two premises that are connected with an “AND” operator (the minimum operator). “AND” operator selects a lower number of rules that have received high activation values. The resulting consequence depends on both linguistic variables. Let us suppose that the current value of speed_uavZ is ZERO. The rule with the highest value is the intersection of NNEAR column and ZERO line of Table 1.

However, it is important to highlight that NMID variable has some influence on the result of the fuzzy rules inference. Therefore, Othrottle output linguistic variable is defined as a combination of NMID and NMIN values. This situation forms points of

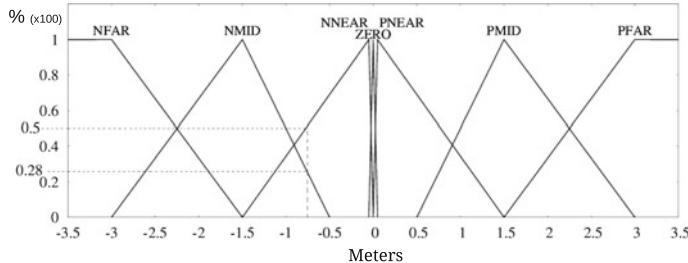


Fig. 13 An example of input at -0.75 m. After fuzzification the distance means 0.28 of negative middle, NMID, and 0.5 of NNEAR

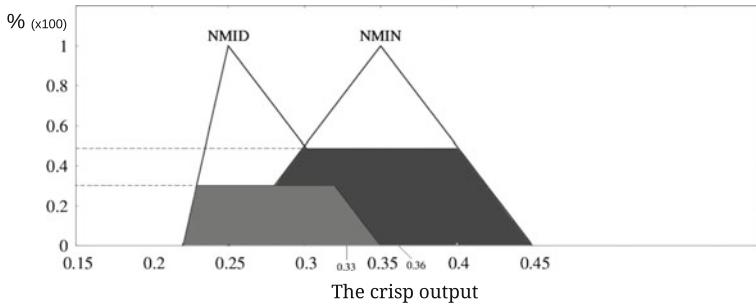


Fig. 14 Obtaining Othrottle raw value after defuzzification process

different activations between two output membership functions, creating two areas. The defuzzification process produces the raw value that represents the influence of both NMID and NMIN values. The values provided to each linguistic value depends on the activation of each rule. Figure 14 shows two possible output values. The first output value is 0.33, and it was obtained by calculating the arithmetic average of the two areas. The second output value is 0.36, and it was calculated using the center of gravity (COG) method that represents a weighted average between the two areas.

4 Open-Source Package of ROS-Based Fuzzy Logic Control Systems

4.1 Package Overview

This section describes our open-Source Package of ROS-based Fuzzy Logic Controllers [1]. This package provides the following artefacts: (i) the proposed fuzzy logic library; (ii) examples of fuzzy set files for the hexacopter movement fuzzy control system; (iii) the fuzzy control system main software implemented using ROS;

(iv) a software to send commands (i.e. the desired target position) to the virtual hexacopter; (v) a telemetry software that displays data from the hexacopter as well as the fuzzy controllers.

The package structure is composed by the following directories³:

- **fz** directory stores the text files that specify the fuzzy set. One fuzzy set is formed by three files: (i) input linguistic variables, e.g. “HexaPlus_i_stabZ.fz” defines the inputs for the vertical controller and hovering fuzzy controller; (ii) fuzzy rules, e.g. HexaPlus_r_stabZ.fz specifies the rules for the vertical controller and hovering fuzzy controller; (iii) output linguistic variables, e.g. “HexaPlus_o_stabZ.fz” defines the output linguistic variables.
- **include** directory provides the C/C++ header files. This directory presents two subdirectories: one subdirectory provides the header files for the tutorial, while the other one provides the header files for the fuzzy library, so that the library can be reused in other projects.
- **src** directory stores the source code. The code files of fuzzy logic library are stored within the subdirectory “fuzzy”, whereas the tutorial source code files are located directly in the **src** directory.
- **scenes** directory provides the V-REP scenarios that are used in this tutorial.

The next sessions discuss how to install, use and test our package. For that, the reader must use Ubuntu 14.04.4 LTS Operating System with the following software installed: ROS Indigo Igloo, catkin, cmake, V-REP PRO EDU version V3.3.0.⁴

4.2 Configuring ROS Environment and Installing the Package

First of all, a workspace is created in order to compile the shared library files so that the V-REP can be integrated with ROS. The V-REP provides a set of ROS packages to generate the share libraries. The workspace will also host the fuzzy package, as well as some libraries will be copied to the V-REP directory.

1. Create a directory under your home directory and initialize the catkin work-space using `catkin_init_workspace`.

```

1 $ cd ~
2 $ $ mkdir -p catkin_hexacopter/src
3 $ cd catkin_hexacopter/src
4 $ catkin_init_workspace
5 $ cd ..
6 $ catkin_make
7 $ source devel/setup.bash

```

³Files and subdirectories created automatically by catkin/make commands are ignored.

⁴This tutorial assumes that V-REP has been installed in /opt/V-REP/ directory.

2. Copy the ROS package from the V-REP directory and generate the libraries with `catkin_make`.

```

1 $ cd ~/catkin_hexacopter/src
2 $ cp -rp /opt/V-REP/V-REP_PRO_EDU_V3_3_0_64_Linux/programming
   /ros_packages/* .
3 $ cd ..
4 $ catkin_make

```

3. Once two shared libraries have been created, copy these libraries to the V-REP directory, enabling the V-REP to work with the `roscore`.

```

1 $ cd ~/catkin_hexacopter
2 $ cp devel/lib/libvrepExtRos.so /opt/V-REP/V-
   REP_PRO_EDU_V3_3_0_64_Linux/
3 $ cp devel/lib/libvrepExtRosSkeleton.so /opt/V-REP/V-
   REP_PRO_EDU_V3_3_0_64_Linux/

```

These libraries enable V-REP to look for an instance of `roscore` at startup. If `roscore` is not running and the simulation scenario has some call to ROS, the simulation fails and the user is warned. V-REP acts as a ROS node, and hence, `roscore` must be running before starting V-REP. The integration between ROS and V-REP succeeded whether the ROS plugins are loaded during the V-REP startup, as shown below.

```

1 $ /opt/V-REP/V-REP_PRO_EDU_V3_3_0_64_Linux/vrep.sh &
2 ...
3 Plugin 'Ros': loading...
4 Plugin 'Ros': load succeeded.
5 ...

```

4. After performing this tutorial, the reader may want to delete V-REP packages, and thus, the following commands must be executed:

```

1 $ cd ~/catkin_hexacopter
2 $ rm -fr ros_bubble_rob vrep_joy vrep_*

```

Once catkin workspace identified as `catkin_hexacopter` has been created and configured, the reader can download our ROS-based fuzzy logic package from [1]. In order to compile and run such a package, the package zip file must be uncompressed inside the the `catkin_hexacopter` workspace source directory.

```

1 $ unzip hexaplusTutorial.zip -d ~/catkin_hexacopter/src
2 $ cd ~/catkin_hexacopter
3 $ catkin_make

```

If the fuzzy package has been uncompressed into a workspace with a lot of others packages, use the option “`pkg`” to compile only the fuzzy package.

```

1 $ catkin_make --pkg hexaplusTutorial

```

These commands compile the package and generate the objects listed below.

```

1 [ 14%] [ 28%] Built target FuzzySet
2 Built target Linguistic
3 [ 28%] [ 28%] Built target FuzzyLoader
4 Built target HexaPlus

```

```

5 [ 28%] Built target LinguisticSet
6 [ 42%] [ 42%] Built target MembershipFunction
7 Built target Rule
8 [ 57%] Built target RuleSet
9 [ 71%] Built target RuleElement
10 [ 71%] [ 71%] Built target navigation

```

Thereafter, it is important to check whether the package was successfully compiled and it is working properly. For that, the roscore must be started, and then three package applications can be executed: `rosvrep_controller`, `rosvrep_panel`, `rosvrep_telemetry`. In addition, check the created topics by using `rostopic` and `rqt_graph`. However, before executing the package applications, the user must run the source command on the setup.sh file (line 2) at least once in the shell session, as well as open five terminals to run each application separately.

```

1 $ roscore &
2 $ source ~/catkin_hexacopter-devel/setup.sh
3 $ xterm & xterm & xterm & xterm & xterm &

```

The following commands should be executed in each terminal.

```

1 $ rosrun hexaplus_tutorial rosvrep_controller
2 $ rosrun hexaplus_tutorial rosvrep_panel
3 $ rosrun hexaplus_tutorial rosvrep_telemetry
4 $ rostopic list
5 $ rqt_graph

```

The `rostopic` command lists the topics beginning with `/vrep/`. These names can be easily modified through the remap argument of `rosrun` command; for details see [21].

4.3 Fuzzy Set Files

As discussed in Sect. 3.1, the fuzzy set is composed of linguistic variables for inputs and outputs, membership functions and rules. Such information can be hard-coded into the source code files. This way, the fussy set is stored in memory by using arrays or lists, and thus, the fuzzy inference engine is able to produce the expected outputs. However, a flexible fuzzy inference engine is able to load the fuzzy set from a file stored in a storage dive (e.g. disk). In the proposed fuzzy library, we implemented a flexible engine that loads the fuzzy sets from plain text files. As described bellow, we defined a simple format to describe linguistic variables, membership functions and rules, in order to facilitate the specification process and also the system maintenance.

The fuzzy set is located in “`fz`” subdirectory. In order to illustrate how to specify a fuzzy set, the vertical movement and hovering controller (see Sect. 3.3) is used as a case study. The files whose name ends with “`_stabZ.fz`” are related to this controller. Listing 1.1 describes the input linguistic variable depicted in Fig. 9 and specified in `HexaPlus_i_stabZ.fz` file. Likewise, Listing 1.2 describes the output linguistic variable depicted in Fig. 10 and specified in `HexaPlus_o_stabZ.fz` file.

Listing 1.1 HexaPlus_i_stabZ.fz: Input linguistic variables and membership functions of vertical movement and hovering controller

```

1 uavZ_error NFAR      -1001  -3.00  -3.00  -1.50
2 uavZ_error NMID     -3.00  -1.50  -1.50  -0.50
3 uavZ_error NNEAR    -1.50  -0.05  -0.05   0
4 uavZ_error ZERO     -0.05   0     0     0.05
5 uavZ_error PNEAR    0     0.05  0.05  1.50
6 uavZ_error PMID     0.50  1.50  1.50  3.00
7 uavZ_error PFAR     1.50  3.00  3.00  1001
8
9 speed_uavZ NFAST    -1000000 -1.00000 -1.00000 -0.50000
10 speed_uavZ NSLOW   -1.00000 -0.50000 -0.50000  0.00000
11 speed_uavZ ZERO    -0.50000 0.00000 0.00000  0.50000
12 speed_uavZ PSLow   0.00000 0.50000 0.50000  1.00000
13 speed_uavZ PFAST   0.50000 1.00000 1.00000  1000000

```

Listing 1.2 HexaPlus_o_stabZ.fz: Output linguistic variables and membership functions of vertical movement and hovering controller

```

1 uavZ_error NFAR      -1001  -3.00  -3.00  -1.50
2 uavZ_error NMID     -3.00  -1.50  -1.50  -0.50
3 uavZ_error NNEAR    -1.50  -0.05  -0.05   0
4 uavZ_error ZERO     -0.05   0     0     0.05
5 uavZ_error PNEAR    0     0.05  0.05  1.50
6 uavZ_error PMID     0.50  1.50  1.50  3.00
7 uavZ_error PFAR     1.50  3.00  3.00  1001
8
9 speed_uavZ NFAST    -1000000 -1.00000 -1.00000 -0.50000
10 speed_uavZ NSLOW   -1.00000 -0.50000 -0.50000  0.00000
11 speed_uavZ ZERO    -0.50000 0.00000 0.00000  0.50000
12 speed_uavZ PSLow   0.00000 0.50000 0.50000  1.00000
13 speed_uavZ PFAST   0.50000 1.00000 1.00000  1000000

```

Linguistic variables files follow the same format. Thus, these output linguistic variable files can be reused as input from different fuzzy controllers. As one can see in Listings 1.1 and 1.2, each line describes one linguistic variable. The first token is the name of linguistic variable, e.g. the vertical distance error `uavZ_error` and the vertical speed `speed_uavZ`. The second token is the name of the membership function, and the next four fields describe the range of values. There are four values in order to create membership function with triangular or trapezoidal shape. A triangle is formed when the second and third values are the same. These four points is referenced in the code implementation by variables “a”, “b”, “c” and “d”, respectively, defined in the `MembershipFunction` class (see `MembershipFunction.h` `MembershipFunction` in the `include` directory).

The vertical movement and hovering controller defines 35 fuzzy rules as depicted in Table 1 in Sect. 3.3. These rules are specified in `HexaPlus_r_stabZ.fz` file that is shown in Listing 1.3).

Listing 1.3 HexaPlus_r_stabZ.fz: fuzzy rules of vertical movement and hovering controller

```

1 STABZ_NFAR_01 if uavZ_error is NFAR and speed_uavZ is PFAST then Othrottle is NMAX
2 STABZ_NFAR_02 if uavZ_error is NFAR and speed_uavZ is PSLow then Othrottle is NMAX
3 STABZ_NFAR_03 if uavZ_error is NFAR and speed_uavZ is ZERO then Othrottle is NMAX
4 STABZ_NFAR_04 if uavZ_error is NFAR and speed_uavZ is NSLOW then Othrottle is NMAX
5 STABZ_NFAR_05 if uavZ_error is NFAR and speed_uavZ is NFAST then Othrottle is NMAX
6
7 STABZ_NMID_01 if uavZ_error is NMID and speed_uavZ is PFAST then Othrottle is NMAX
8 STABZ_NMID_02 if uavZ_error is NMID and speed_uavZ is PSLow then Othrottle is NMAX
9 STABZ_NMID_03 if uavZ_error is NMID and speed_uavZ is ZERO then Othrottle is NMID
10 STABZ_NMID_04 if uavZ_error is NMID and speed_uavZ is NSLOW then Othrottle is NMID

```

```

11| STABZ_NMID_05 if uavZ_error is NMID and speed_uavZ is NFAST then Othrottle is NMID
12|
13| STABZ_NNEAR_01 if uavZ_error is NNEAR and speed_uavZ is PFAST then Othrottle is NMID
14| STABZ_NNEAR_02 if uavZ_error is NNEAR and speed_uavZ is PSLOW then Othrottle is NMID
15| STABZ_NNEAR_03 if uavZ_error is NNEAR and speed_uavZ is ZERO then Othrottle is NMIN
16| STABZ_NNEAR_04 if uavZ_error is NNEAR and speed_uavZ is NSLOW then Othrottle is NMIN
17| STABZ_NNEAR_05 if uavZ_error is NNEAR and speed_uavZ is NFAST then Othrottle is NMIN
18|
19| STABZ_ZERO_01 if uavZ_error is ZERO and speed_uavZ is PFAST then Othrottle is NMIN
20| STABZ_ZERO_02 if uavZ_error is ZERO and speed_uavZ is PSLOW then Othrottle is NMIN
21| STABZ_ZERO_03 if uavZ_error is ZERO and speed_uavZ is ZERO then Othrottle is ZERO
22| STABZ_ZERO_04 if uavZ_error is ZERO and speed_uavZ is NSLOW then Othrottle is PMIN
23| STABZ_ZERO_05 if uavZ_error is ZERO and speed_uavZ is NFAST then Othrottle is PMIN
24|
25| STABZ_PNEAR_01 if uavZ_error is PNEAR and speed_uavZ is NFAST then Othrottle is PMID
26| STABZ_PNEAR_02 if uavZ_error is PNEAR and speed_uavZ is NSLOW then Othrottle is PMID
27| STABZ_PNEAR_03 if uavZ_error is PNEAR and speed_uavZ is ZERO then Othrottle is PMID
28| STABZ_PNEAR_04 if uavZ_error is PNEAR and speed_uavZ is PSLOW then Othrottle is PMIN
29| STABZ_PNEAR_05 if uavZ_error is PNEAR and speed_uavZ is PFAST then Othrottle is PMIN
30|
31| STABZ_PPID_01 if uavZ_error is PMID and speed_uavZ is NFAST then Othrottle is PMAX
32| STABZ_PPID_02 if uavZ_error is PMID and speed_uavZ is NSLOW then Othrottle is PMAX
33| STABZ_PPID_03 if uavZ_error is PMID and speed_uavZ is ZERO then Othrottle is PMID
34| STABZ_PPID_04 if uavZ_error is PMID and speed_uavZ is PSLOW then Othrottle is PMID
35| STABZ_PPID_05 if uavZ_error is PMID and speed_uavZ is PFAST then Othrottle is PMID
36|
37| STABZ_PFAR_01 if uavZ_error is PFAR and speed_uavZ is NFAST then Othrottle is PMAX
38| STABZ_PFAR_02 if uavZ_error is PFAR and speed_uavZ is NSLOW then Othrottle is PMAX
39| STABZ_PFAR_03 if uavZ_error is PFAR and speed_uavZ is ZERO then Othrottle is PMAX
40| STABZ_PFAR_04 if uavZ_error is PFAR and speed_uavZ is PSLOW then Othrottle is PMAX
41| STABZ_PFAR_05 if uavZ_error is PFAR and speed_uavZ is PFAST then Othrottle is PMAX

```

The first token is the rule name. Such information is used to identify each rule and it is not used for processing. The token “if” indicates that the next tokens are related to the rule premises. Each premise must specify one or more logical expressions following the format *name of linguistic variable* “is” *name of a membership function value*. The token “is” indicates that the activation level for each membership function must be calculated. A premise may indicate multiple expressions that are related by means of “AND” or “OR” operators. If “AND” operator is used, the rule activation (consequence) is equal to the minimum activation of all the premises of this rule. If “OR” operator is used, the rule activation is equal to the maximum activation of all premises.

The token “then” indicates that the next tokens are related to the consequence. During the rule inference process, an activation is assigned to the rule according to its premises activation. During the defuzzification, the activation level of each output linguistic variable is obtained by using the maximum operator, forming an area under all the membership functions that are activated in the rule (see Sect. 3.1). There may be one or more membership function values whose area value is greater than zero. The Center of Gravity (COG) approach is used to determine the weighted average of these areas which is used to determine the raw output value. Section 3.3 provides an example of this process.

4.4 Fuzzy Library Implementation

The fuzzy library provides the set of classes depicted in class diagram shown in Fig. 15. Our package provides a C++ implementation of the fuzzy library. A *Fuzzy Set* object is composed of two sets of *Linguistic Variables* objects: one for

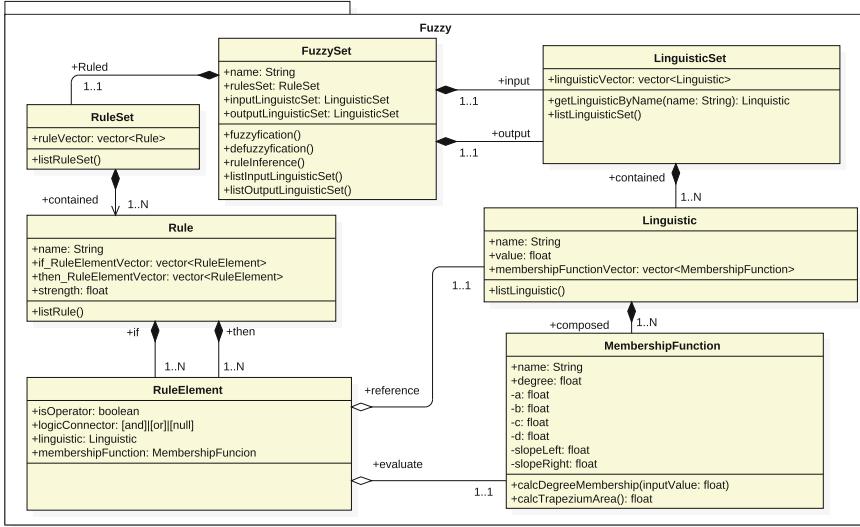


Fig. 15 The UML class diagram of the fuzzy logic library

representing input and another representing output linguistic variables. In addition, *Fuzzy Set* object owns a set of fuzzy *Rules*. Each *Rules* is composed of one or more premises and one or more consequences. Premises and consequences are represented as *Rules Element* objects. One premise describes a logic expression that includes a *Linguistic Variable* and *Membership Function* value. One consequence defines a value for a output *Linguistic Variable*. Each *Linguistic Variable* has a set of *Membership Functions*, which, in turn, represents triangular or trapezoidal shapes as depicted in Figs. 9, 10, and 11 in Sect. 3.3. The fuzzy library provides an additional class named *FuzzyLoader*. Such a class is responsible to load the information contained in the fuzzy set files, i.e. linguistic variables, membership functions, and rules (see Sect. 4.3).

In order to process the fuzzy control system implemented using the proposed library, the sequence of steps must be performed. First of all, the objects representing fuzzy set must be instantiated. For that, a *FuzzyLoader* object is created. It loads the information from the fuzzy set files (*.fz) and instantiates the library objects accordingly. In the hexacopter movement control system, the *HexaPlus* class implements a method called *HexaPlus::initFZ()* that is responsible for loading the fuzzy set. Listing 1.4 shows a code fragment of this method. This code shows how to load the fuzzy set files of the vertical movement and hovering controller.

Listing 1.4 Source code fragment of *HexaPlus::initFZ()* method

```

1 void HexaPlus::initFZ() {
2     // Getting the package path
3     std::string path = ros::package::getPath("hexaplus_tutorial");
4     // String objects declared
5     std::stringstream ssi, sso, ssr;
6

```

```

7 // Using the FuzzyLoader to upload the fuzzy artifacts
8 fuzzy::FuzzyLoader fzLoader;
9 ... // some lines are omitted
10
11 // Loading inputs and outputs linguistics variable
12 // with their membership functions
13 // and the rules of vertical controller
14 ssi.str(""); ssi << path << "/fz/HexaPlus_i_stabZ.fz";
15 sso.str(""); sso << path << "/fz/HexaPlus_o_stabZ.fz";
16 ssr.str(""); ssr << path << "/fz/HexaPlus_r_stabZ.fz";
17
18 fzLoader.loadFromFile(fS_stabZ,
19                         ssi.str().c_str(),
20                         sso.str().c_str(),
21                         ssr.str().c_str());
22 ... // remainder lines are omitted

```

Once the fuzzy objects are instantiated, the system controller can be initialized and executed. The `HexaPlus::initHexaPlus()` method initializes all variables related to the hexacopter movement control system. On the other hand, the source code file named `rosvrep_controller.cpp` contains the `main()` function. In addition to the invocation of `HexaPlus::initHexaPlus()` method, the `main()` function defines a ROS node and configures the data publishers and subscribers. Thereafter the main control loop is executed. Details are provided in Sect. 4.5.

The fuzzy system is processed within the `FuzzySet::fuzzifying()` method. As presented in Sect. 3.2, there are five fuzzy controllers, and hence, there are `FuzzySet` objects to control hovering, vertical and horizontal movements and heading. Listing 1.5 shows the implementation of `FuzzySet::fuzzifying()` method. As one can see three main steps are executed.

Listing 1.5 Source code of `FuzzySet::fuzzifying()` method

```

1 void FuzzySet::fuzzifying()
2 {
3     fuzzyfication();
4     ruleInference();
5     defuzzification();
6 }

```

The first step is called *fuzzification* and consists of converting the raw value of a sensor reading into a value of linguistic variable. For that, the raw value is compared to the range of values defined in a membership function, in order to calculate the degree of membership of the raw values. Therefore, once the degree of membership is calculated, activation level for the raw value is identified. Listing 1.6 shows `FuzzySet::fuzzyfication()` method implementation. Listing 1.7 shows `MembershipFunction::calcDegreeMembership()` method implementation. The `delta1` and `delta2` variables indicate the distance of `inputValue` parameter⁵ to the points `a` and `d` that represent the base of the trapezium or triangle. The membership degree is calculated by comparing the inclination of two segments (`slopeLeft` and `slopeRight`) multiplied by the corresponding delta values.

⁵I.e. the raw value read from a sensor.

Listing 1.6 Source code of `FuzzySet::fuzzyfication()` method

```

1 void FuzzySet::fuzzyfication()
2 {
3     typedef std::vector<Linguistic>::iterator
4         linguisticIterator_t;
5     typedef std::vector<MembershipFunction>::iterator
6         membershipFunctionIterator_t;
7     linguisticIterator_t it_Ling;
8     membershipFunctionIterator_t it_Mem;
9
10    for (it_Ling=inputLinguisticSet->linguisticVector.begin();
11        it_Ling !=inputLinguisticSet->linguisticVector.end();
12        it_Ling++) {
13        for (it_Mem=it_Ling->membershipFunctionVector.begin();
14            it_Mem!=it_Ling->membershipFunctionVector.end();
15            it_Mem++) {
16            it_Mem->calcDegreeMembership(it_Ling->value);
17        }
18    }
19 }
```

Listing 1.7 Source code of `MembershipFunction::calcDegreeMembership()` method

```

1 void MembershipFunction::calcDegreeMembership(float inputValue)
2 {
3     float delta1, delta2;
4     delta1 = inputValue - a;
5     delta2 = d - inputValue;
6     if ((delta1 <= 0) || (delta2 <= 0)) {
7         degree = 0;
8     } else {
9         degree= minimum((slopeLeft*delta1),(slopeRight*delta2));
10    }
11    degree = minimum(degree,FZ_MAX_LIMIT);
12 }
13 // Slopes are calculated during the loading of fuzzy set files
14 // slopeLeft = (float)FZ_MAX_LIMIT/(b-a);
15 // slopeRight = (float)FZ_MAX_LIMIT/(d-c);
```

The second step of fuzzifying process is to perform the rules inference process. Such a process is implemented in `FuzzySet::ruleInference()` method as shown in Listing 1.8. As one can see, the inference process has two main steps. In the first step, the inference values previously calculated are dismissed. On the other hand, in the second step, the strength of all rules is calculated. The degree of membership for each linguistic variable is used to define the rule strength which, in turn, is used to define the rule activation.

Listing 1.8 Source code of `FuzzySet::ruleInference()` method

```

1 void FuzzySet::ruleInference()
2 {
3     typedef std::vector<Rule>::iterator RuleIterator_t;
4     typedef std::vector<RuleElement>::iterator
5         RuleElementIterator_t;
6
7     RuleIterator_t           it_r;
8     RuleElementIterator_t it_e;
9
10    //Clean up IF elements for new round
11    for (it_r=ruleSet->ruleVector.begin(); it_r != ruleSet->
12        ruleVector.end(); it_r++) {
13        it_r->strength=0;
```

```

12     for (it_e=it_r->then_RuleElementVector.begin(); it_e != it_r->then_RuleElementVector.end(); it_e++) {
13         it_e->linguistic->value = 0;
14         it_e->membershipFunction->degree = 0;
15     }
16 }
17
18 // Inference
19 float strength_tmp;
20 for (it_r=ruleSet->ruleVector.begin(); it_r != ruleSet->
21     ruleVector.end(); it_r++) {
22     // Calculate the strength of the premises
23     strength_tmp = FZ_MAX_LIMIT;
24     for (it_e=it_r->if_RuleElementVector.begin(); it_e != it_r->
25         if_RuleElementVector.end(); it_e++) {
26         strength_tmp = minimum(strength_tmp, it_e->
27             membershipFunction->degree);
28     if (!it_e->isOperator)
29         strength_tmp = 1 - strength_tmp;
30
31     // Calculate the strength of the consequences
32     for (it_e=it_r->then_RuleElementVector.begin(); it_e != it_r->then_RuleElementVector.end(); it_e++) {
33         it_e->membershipFunction->degree = maximum(strength_tmp,
34             it_e->membershipFunction->degree);
35     if (!it_e->isOperator)
36         it_e->membershipFunction->degree = 1 - it_e->
37             membershipFunction->degree;
38     }
39     it_r->strength = strength_tmp;
40 }
41
42 }
```

Once all elements of rules are evaluated, the third step of fuzzifying process is the defuzzification process. As discussed in Sect. 3.2, during the defuzzification process, the triangle/trapezoid area of the output linguistic variables is calculated taking into account the membership function and the rule activation. Then the linguistic value chosen as output is converted into a raw value that may be applied to the rotors. Such a raw value is obtained by means of calculating an arithmetic average or a weighted average (Center of Gravity method) of two areas. Listing 1.9 shows the implementation of `FuzzySet::defuzzification()` method.

Listing 1.9 Source code of `FuzzySet::defuzzification()` method

```

1 void FuzzySet::defuzzification()
2 {
3     typedef std::vector<Linguistic>::iterator
4         LinguisticIterator_t;
5     typedef std::vector<MembershipFunction>::iterator
6         MembershipFunctionIterator_t;
7
8     LinguisticIterator_t           it_l;
9     MembershipFunctionIterator_t it_m;
10
11    float sum_prod;
12    float sum_area;
13    float area, centroide;
14
15    for (it_l=outputLinguisticSet->linguisticVector.begin(); it_l != outputLinguisticSet->linguisticVector.end(); it_l++)
16    {
17        sum_prod=sum_area=0;
```

```

16     for (it_m = it_l->membershipFunctionVector.begin(); it_m
17         != it_l->membershipFunctionVector.end(); it_m++)
18     {
19         area = it_m->calcTrapeziumArea();
20         centroide = it_m->a + ((it_m->d - it_m->a) / 2.0);
21         sum_prod += area * centroide;
22         sum_area += area;
23     }
24
25     if (sum_area==0)
26         it_l->value = FZ_MAX_OUTPUT;
27     else
28     {
29         it_l->value = sum_prod/sum_area;
30     }
31 }
```

4.5 Main Controller Implementation

The hexacopter movement fuzzy control system has been implemented in some distinct source code files. The source code file named `rosvrep_controller.cpp` implements the main control loop, i.e. the system `main()` function. The file `HexaPlus.cpp` contains the implementation of the `HexaPlus` class that is responsible for initializing the fuzzy library objects (see Sect. 4.4).

The `main()` function is divided in two parts. The first one performs all necessary initialization, i.e. it instantiates the `HexaPlus` object, loads the fuzzy set files, creates a ROS node, and configures the data publishers (i.e. callback functions) and subscribers. Listing 1.11 depicts fragments of the initialization part of the `main()` function.

Listing 1.10 Fragments of `main()` function in `rosvrep_controller.cpp`

```

1 //////////////// THE CALLBACK FUNCTION
2 /////////////////////////////////
3 // Subscriber callback functions for euler angles
4 ...
5 void callback_eulerZ(const std_msgs::Float32 f)
6     { eulerZ = f.data; }
7 // Subscriber callback functions for GPS position
8 ...
9 void callback_gpsZ(const std_msgs::Float32 f)
10    { gpsZ = f.data; }
11 // Subscriber callback functions for accelerometer sensor
12 ...
13 void callback_accelZ(const std_msgs::Float32 f)
14    { accelZ = f.data; }
15 // Subscriber callback functions for operator setpoints
16 ...
17 void callback_gpsZ_setpoint(const std_msgs::Float32 f)
18    { gpsZ_setpoint = f.data; }
19 ...
20 ////////////////// END CALLBACK FUNCTION
21 /////////////////////////////////
22 int main(int argc, char* argv[])
23 {
```

```

23     unsigned long int time_delay=0;
24
25 // Initialize the ros subscribers
26 ros::init(argc, argv, "rosvrep_controller");
27 ros::NodeHandle n;
28
29 // the rosSignal is used to send signal to uav via Publisher.
30 std_msgs::Float32 rosSignal;
31 // Hexacopter sensor subscribers
32 ... // some lines are omitted
33
34 // Initialize the ROS Publishers
35 // Rotors publishers
36 ros::Publisher rosAdv_propFRONT =
37     n.advertise<std_msgs::Float32>("/vrep/propFRONT",1);
38 ros::Publisher rosAdv_propLEFT_FRONT =
39     n.advertise<std_msgs::Float32>("/vrep/propLEFT_FRONT",1)
40     ;
41 ros::Publisher rosAdv_propLEFT_REAR =
42     n.advertise<std_msgs::Float32>("/vrep/propLEFT_REAR",1);
43 ros::Publisher rosAdv_propREAR =
44     n.advertise<std_msgs::Float32>("/vrep/propREAR",1);
45 ros::Publisher rosAdv_propRIGHT_FRONT =
46     n.advertise<std_msgs::Float32>("/vrep/propRIGHT_FRONT"
47     ,1);
48 ros::Publisher rosAdv_propRIGHT_REAR =
49     n.advertise<std_msgs::Float32>("/vrep/propRIGHT_REAR",1)
50     ;
51 ros::Publisher rosAdv_propYaw =
52     n.advertise<std_msgs::Float32>("/vrep/Yaw",1);
53 ... // remainder lines are omitted

```

The second part is the main control loop of the hexacopter movement fuzzy control system. Such a loop performs three main activities: (i) pre-processing phase, (ii) processing of five distinct fuzzy controllers, (iii) post-processing phase. These activities are discussed in Sect. 3.2. Moreover, the execution frequency of loop iterations is 10 Hz. Such an execution frequency is obtained by using the commands `loop_rate.sleep()` and `ros::spinOnce()` at the end of the loop. The 10 Hz timing requirement has been arbitrarily defined and has been demonstrated to be enough to control a simulated hexacopter as discussed in Sect. 5. However, it is important to highlight that a more careful and sound timing analysis is required in order to define the execution frequency of the main control loop for a real hexacopter. A discussion on such an issue is out of this chapter scope. Interested reader should refer to [19, 22–27].

The pre-processing phase is responsible for acquiring data from the input sensors, processing the input movement commands, as well as for calculating the controlled data used as input to the five fuzzy controllers. Two examples of data calculated in this phase are: (i) vertical and horizontal speed calculated using the hexacopter displacement over time; and (ii) the drift of new heading angle in comparison with the actual heading. Listing 1.11 presents some fragments of the code related to the pre-processing phase.

Listing 1.11 Fragments of pre-processing phase in `rosvrep_controller.cpp`

```

1     ... // some lines are omitted
2     // Determine the delta as errors.
3     // It means the difference between setpoint and current
        information

```

```

4   // GPS error
5   gpsX_error = (float) gpsX_setpoint - gpsX;
6   gpsY_error = (float) gpsY_setpoint - gpsY;
7   gpsZ_error = (float) gpsZ_setpoint - gpsZ;
8
9
10  // View position error (yaw or heading of the hexacopter)
11  viewX_error = (float) viewX_setpoint - gpsX;
12  viewY_error = (float) viewY_setpoint - gpsY;
13
14  // Calculate the drift_angle
15  // This angle is the difference between direction
16  // to navigate and direction of view (yaw).
17  drift_angle = (float) eulerZ - uav_goal_angle;
18  ... // remainder lines are omitted

```

Once the pre-processing phase is executed, the second activity is responsible to execute the five fuzzy controllers. This occurs by means of invoking the `fuzzifying()` method of each controller `FuzzySet` object. The “fuzzifying” process includes “fuzzification”, rules inference, and “defuzzification” (see Sect. 4.4). Listing 1.12 depicts the code fragment that processes the five fuzzy controllers.

Listing 1.12 Fragment depicting the processing five fuzzy controllers in `rosvrep_controller.cpp`

```

1   ... // previous lines are omitted
2   hexaplus.fS_stabX->fuzzifying();
3   hexaplus.fS_stabY->fuzzifying();
4   hexaplus.fS_stabZ->fuzzifying();
5   hexaplus.fS_yaw->fuzzifying();
6   hexaplus.fS_hnav->fuzzifying();
7   ... // remainder lines are omitted

```

The last activity is the post-processing phase. In this phase the output linguistic variables are transformed in raw values that are applied on the rotors in order to control the hexacopter movements. Listing 1.13 presents a fragment of post-processing phase code.

Listing 1.13 Fragment depicting the post-processing phase in `rosvrep_controller.cpp`

```

1   // fuzzifying is finished, applying the outputs
2   Opitch      = hexaplus.fz_Opitch->value;
3   Oroll       = hexaplus.fz_Oroll->value;
4   Othrottle   = hexaplus.fz_Othrottle->value;
5   Oyaw        = hexaplus.fz_Oyaw->value;
6   Opitch_nav  = hexaplus.fz_Opitch_nav->value;
7
8   propForceFRONT      = (float) Othrottle - 0.45*zOth*cos(
9     angleOth);
10  propForceRIGHT_FRONT = (float) Othrottle - (0.45*zOth*(sin(
11    angleOth)/2));
12  propForceRIGHT_REAR = (float) Othrottle - (0.45*zOth*(sin(
13    angleOth)/2));
14  propForceREAR        = (float) Othrottle + 0.45*zOth*cos(
15    angleOth);
16  propForceLEFT_REAR   = (float) Othrottle + (0.45*zOth*(sin(
17    angleOth)/2));
18  propForceLEFT_FRONT  = (float) Othrottle + (0.45*zOth*(sin(
19    angleOth)/2));
20
21  // Sending signals to the rotors
22  rosSignal.data = propForceFRONT;

```

```

17 rosAdv_propFRONT.publish(rosSignal);
18 rosSignal.data = propForceRIGHT_FRONT;
19 rosAdv_propRIGHT_FRONT.publish(rosSignal);
20 rosSignal.data = propForceRIGHT_REAR;
21 rosAdv_propRIGHT_REAR.publish(rosSignal);
22 rosSignal.data = propForceREAR;
23 rosAdv_propREAR.publish(rosSignal);
24 rosSignal.data = propForceLEFT_REAR;
25 rosAdv_propLEFT_REAR.publish(rosSignal);
26 rosSignal.data = propForceLEFT_FRONT;
27 rosAdv_propLEFT_FRONT.publish(rosSignal);
28 rosSignal.data = Oyaw;
29 rosAdv_propYaw.publish(rosSignal);

```

Finally, it is worth mentioning that the main controller interacts with other two applications. A command interface application named *Panel* sends commands to determine a new position, as well as new heading direction, towards which the hexacopter must fly. Moreover, some data produced in the main controller are published so that these telemetry data can be seen within an application named *Telemetry*. Listing 1.14 shows the code in `rosvrep_controller.cpp` that configures ROS publishers for the telemetry data. Next sections provide detail on these two applications.

Listing 1.14 Configuring ROS publisher for publishing telemetry data in `rosvrep_controller.cpp`

```

1 // Telemetry
2 ros::Publisher rosAdv_gpsX_error = n.advertise<std_msgs::
3   Float32>("/hexaplus_tutorial/gpsX_error",1);
4 ros::Publisher rosAdv_gpsY_error = n.advertise<std_msgs::
5   Float32>("/hexaplus_tutorial/gpsY_error",1);
6 ros::Publisher rosAdv_gpsZ_error = n.advertise<std_msgs::
7   Float32>("/hexaplus_tutorial/gpsZ_error",1);
8 ros::Publisher rosAdv_drift_angle = n.advertise<std_msgs::
9   Float32>("/hexaplus_tutorial/drift_angle",1);
10 ros::Publisher rosAdv_uav_goal_angle = n.advertise<std_msgs::
11   Float32>("/hexaplus_tutorial/uav_goal_angle",1);
12 ros::Publisher rosAdv_uav_goal_dist = n.advertise<std_msgs::
13   Float32>("/hexaplus_tutorial/uav_goal_dist",1);
14 ros::Publisher rosAdv_speed_uavZ = n.advertise<std_msgs::
15   Float32>("/hexaplus_tutorial/speed_uavZ",1);
16 ros::Publisher rosAdv_speed_goal = n.advertise<std_msgs::
17   Float32>("/hexaplus_tutorial/speed_goal",1);

```

4.6 Command Interface Implementation

The command interface application named *Panel* is a ROS node that allows a user to send commands to modify hexacopter pose and position. The implementation of such an application is provided in `rosvrep_panel.cpp` file. Two types of commands are allowed: (i) the user can set a new (X, Y, Z) position, and hence, the hexacopter will fly towards this target position; (ii) the user can set a new heading direction by setting a new (X, Y) position, and hence, the hexacopter will perform a yaw movement in order to aim the target position.

The *Panel* application is very simple: it publishes a setpoint position and a view direction, as well as provides means for user input. Listing 1.15 shows the code fragment that configures the ROS publisher for the new 3D position (i.e. setpoint) and new heading (i.e. view direction).

Listing 1.15 Configuring ROS publisher for telemetry data in `rosvrep_panel.cpp`

```

1 // Operator setpoint Publishers
2 ros::Publisher rosAdv_gpsX_setpoint = n.advertise<std_msgs::
3   Float32>("/hexaplus_tutorial/gpsX_setpoint",1);
4 ros::Publisher rosAdv_gpsY_setpoint = n.advertise<std_msgs::
5   Float32>("/hexaplus_tutorial/gpsY_setpoint",1);
6 ros::Publisher rosAdv_gpsZ_setpoint = n.advertise<std_msgs::
7   Float32>("/hexaplus_tutorial/gpsZ_setpoint",1);
8
9 ros::Publisher rosAdv_viewX_setpoint = n.advertise<std_msgs::
10  Float32>("/hexaplus_tutorial/viewX_setpoint",1);
11 ros::Publisher rosAdv_viewY_setpoint = n.advertise<std_msgs::
12  Float32>("/hexaplus_tutorial/viewY_setpoint",1);

```

The *Panel* application must be executed with the `rosrun` command as depicted in line 01 from Listing 1.16. When the user presses “s”, he/she is asked to inform new position setpoint in terms of X, Y, Z coordinates. When the user presses “y”, he/she is asked to inform the new heading direction new in terms of X, Y coordinates. In the example presented in lines 09–12 from Listing 1.16, the user sent (5, 3, 7) as new (X, Y, Z) target position. It is important to mention that the values for (X, Y, Z) coordinates are measured in meters. After sending the new setpoints, the hexacopter starts moving. If the user press **CTRL-C** and the **ENTER** keys, the program is finished and the hexacopter continues until it reaches the target position.

Listing 1.16 Panel application

```

1 $ rosrun hexaplus_tutorial rosvrep_panel
2 =====
3 Setpoints for position [s]
4 Setpoints for View heading [y]
5 Or CTRL-C to exit.
6
7 Enter the option: s
8 Enter X value: 5
9 Enter Y value: 3
10 Enter Z value: 7

```

4.7 Telemetry Implementation

The *Telemetry* application is also a very simple program. It receives the signals from the hexacopter sensors and from some data calculated during the execution of the control program. Likewise the *Panel* application, the *Telemetry* application is executed with the `rosrun` command as depicted in line 01 from Listing 1.18. The telemetry data is shown in lines 03–23.

Listing 1.17 Panel application

```

1 $ rosrun hexaplus_tutorial rosvrep_telemetry
2
3 ----- Telemetry -----
4 gpsX ..... : 0.000000
5 gpsY ..... : 0.000000
6 gpsZ ..... : 0.000000
7 gpsX_error ..... : 0.000000
8 gpsY_error ..... : 0.000000
9 gpsZ_error ..... : 0.000000
10 drift_angle ..... : 0.000000 (0.000000
11     degrees)
12 uav_goal_angle ..... : 0.000000 (0.000000
13     degrees)
14 uav_goal_dist ..... : 0.000000
15 speed_uavz ..... : 0.000000
16 speed_goal ..... : 0.000000
17 ----- Operator Command -----
18 gpsX_setpoint ..... : 0.000000
19 gpsY_setpoint ..... : 0.000000
20 gpsZ_setpoint ..... : 0.000000
21 viewX_setpoint ..... : 0.000000
22 viewY_setpoint ..... : 0.000000
23 ----- Press CTRL-C to exit

```

Telemetry application is a very simple program. It subscribes some ROS topics and displays them on the terminal. The program terminates when CTRL-C is pressed. The `rosvrep_telemetry.cpp` file implements this application. The main part of the code is the declaration of ROS subscribers and callback functions. Listing 1.18 shows these declarations. Callback functions declaration is depicted in line 02–04, while ROS subscribers in line 09–11. One can notice that some topics start with “/vrep”, others with “/hexaplus_tutorial”; this means that some topics came from V-REP and other ones from the control program.

Listing 1.18 Panel application

```

1 ... // previous line omitted
2 void callback_gpsX_error(const std_msgs::Float32 f) { gpsX_error
3     = f.data; }
4 void callback_gpsY_error(const std_msgs::Float32 f) { gpsY_error
5     = f.data; }
6 void callback_gpsZ_error(const std_msgs::Float32 f) { gpsZ_error
7     = f.data; }
8 ...
9 ros::Subscriber sub_gpsX_error = n.subscribe("/hexaplus_tutorial
10 /gpsX_error",1, callback_gpsX_error);
11 ros::Subscriber sub_gpsY_error = n.subscribe("/hexaplus_tutorial
12 /gpsY_error",1, callback_gpsY_error);
13 ros::Subscriber sub_gpsZ_error = n.subscribe("/hexaplus_tutorial
14 /gpsZ_error",1, callback_gpsZ_error);
15 ...
16 ... // remaining lines omitted

```

5 Virtual Experimentation Platform

5.1 *Introduction*

A common tool used during the design of control systems is the simulator. There is a number of different simulators available for using, e.g. Simulink, Gazebo and Stage. In special, for robotics control systems design, a virtual environment for simulation must allow the creation of objects and also the specification of some of the physical parameters for both objects and the environment. The virtual environment should also provide a programming interface to control not only the simulation, but also the objects behavior and the time elapsed in simulation.

Although there are some robotics simulators supported in ROS such as Gazebo and Stage, this tutorial discusses the use of a different robotics simulator named V-REP. The main goal is to show the feasibility of using other (non-standard) simulators, opening room for the engineer to choose the tools he/she finds suitable for his/her project. The hexacopter movement fuzzy control system is used to illustrate how to integrate V-REP with ROS. An overview on V-REP virtual simulation environment is presented, so that the reader can understand how a virtual hexacopter was created. In addition, the reader will learn how the V-REP acts as a ROS publisher/subscriber to exchange messages with `roscore`.

V-REP uses the Lua language [28] to implement scripts that access and control the simulator. Lua is quite easy to learn, and hence, only a few necessary instructions are presented herein. Although V-REP uses Lua for its internal scripts, there are many external interfaces to other languages, such as C/C++, Java, Python, Matlab and ROS. V-REP documentation is extensive, and hence, the interested reader should refer to [29].

The installation of the V-REP simulator on Linux is simple: the reader must download the compressed installation file from Coppelia Robotics' website [30] and expand it on a directory using the UNIX `tar` command. It is interesting to mention some subdirectories within V-REP directory:

- **scenes:** V-REP provides a number of scenes as examples. The scene files extension is “`ttt`”.
- **tutorial:** This directory provides all scenes used in the tutorials presented in the V-REP site [29].
- **programming:** This directory provides examples written in C/C++, Java, Lua and Python. In addition, it provides the `ros_packages` interface that are in this tutorial.

5.2 V-REP Basics

When V-REP is started, a blank scenario is open automatically for using. The user can start developing a new scenario, or open a scenario created previously, or open a scenario from `scenes` directory. Figure 16 shows a screenshot.

In order to illustrate the use of V-REP, select the menus `File` → `Open scene` and choose the scene `Hexacopter.ttt` provided in the directory `~/catkin_hexacopter/src/hexaplus_tutorial/scenes`. A complex object like a hexacopter is built by putting objects under a hierachic structure. For instance, the sensors such as GPS, gyroscope and accelerometer are under the `HexacopterPlus` object. During the simulation execution, if the `HexacopterPlus` or any subpart, is moved, all parts are moved as if they are a single object. Any primitive object, e.g. cuboids, cylinders and spheres, can be inserted into a scene. There are some especial objects such as joints, video sensors, force sensors, and other. The special objects have some specific attributes used during simulation, e.g. position information, angle measurements, force data, etc. The sensors and rotors are made from these kinds of objects. The user can get some already available devices from “Model Browser”. For instance, there are several sensors available in the `Model Browser` → `components` → `sensors`, e.g. laser scanners, the Kinect sensor, Velodyne, GPS, Gyrosensor and Accelerometer. The last three sensors were used in the hexacopter model.

It is important to mention that when a new robot is created, one must pay attention to the orientation between the robot body and its subparts, especially sensors. Sensors will not work properly whether there are inconsistencies in the parts orientation. The

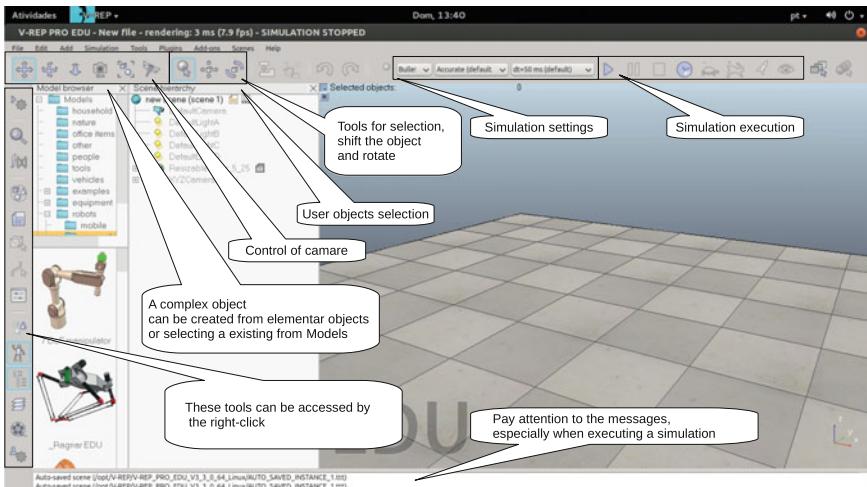


Fig. 16 Screenshot of the V-REP initial screen

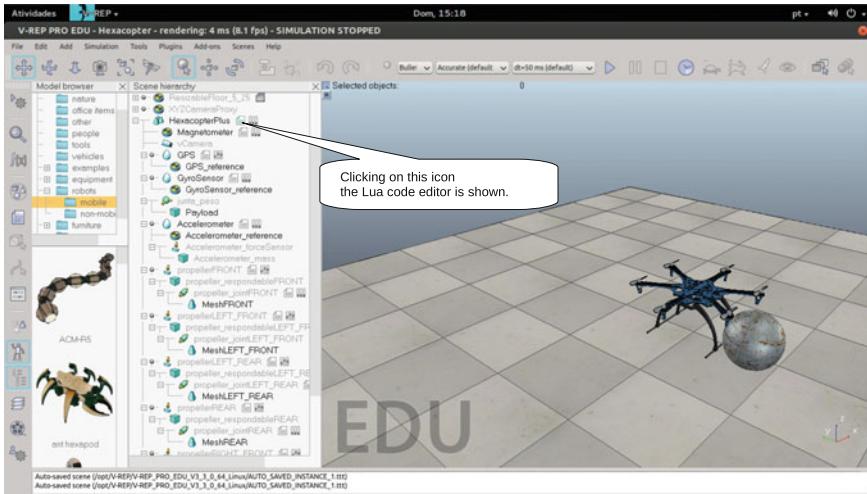


Fig. 17 Open the Lua script code

inertial frame 3D orientation is shown on the bottom right corner. When one clicks on any object, the 3D axes of the selected object body orientation is depicted.

The camera sensor is an exception. The camera orientation has a rotation of $+90^\circ$ over the Z-axis and -90° over the X-axis in relation to the axes of the robot body. Such a situation leads to an issue: Z-axis of the camera matches with the X-axis of the robot. Thus, the camera X-axis matches the robot Y-axis, and the camera Y-axis matches the robot Z-axis. Such a difference can be seen by clicking on vCamera and hexacopter object while pressing the shift key at the same time.

In addition, one can observe that some objects have an icon to edit its Lua script code, as shown at Fig. 17. If the object does not have a piece of code, it is possible to add one by the right-clicking on the object and choosing Add → Associated child script → Non Threaded (or Threaded).

While a simulation is running, V-REP executes the scripts associated to each object throughout the main internal loop. Script execution can be run in separate thread whether the associated script is indicated as threaded. V-REP controls the simulation elapsing time by means of time parameters. In order to execute the simulation of this tutorial, set the time configuration as “Bullet”, “Fast” at “dt = 10.0 ms”. This will ensure a suitable simulation speed.

5.3 Publishing ROS Topics

V-REP provides a plugin infrastructure that allows the engineer customize the simulation tool. *RosPlugin services* in V-REP is an interface to support general ROS

functionality. The V-REP has several mechanisms to communicate with the user code: (i) tubes are similar to the UNIX pipe mechanism; (ii) signals are similar to global variables; (iii) wireless communication simulation; (iv) persistent data blocks; (v) custom Lua functions; (vi) serial port; (vii) LuaSocket; (viii) custom libraries, etc. An easy way to communicate with ROS is creating a V-REP signal and publishing or subscribing its topic. *RosPlugin publishers* offer an API to setup and publish data within ROS topics.

An example on how the V-REP publishes messages to `roscore` can be found in the Lua child object script of *HexacopterPlus*. Let us consider the GPS as an example. Before publishing GPS data, it is necessary to check if the ROS module has been loaded. Listing 1.19 depicts the Lua script defined in the *HexacopterPlus* element as shown in Fig. 17.

Listing 1.19 Lua script to check whether ROS module is loaded

```

1 ... -- previous lines are omitted
2 -- Check if the required remote Api plugin is there:
3 moduleName=0
4 moduleVersion=0
5 index=0
6 pluginNotFound=true
7 while moduleName do
8     moduleName , moduleVersion=simGetModuleName(index)
9     if (moduleName=='Ros') then
10         pluginNotFound=false
11     end
12     index=index+1
13 end
14 if (pluginNotFound) then
15     -- Display an error message if the plugin was not found:
16     simDisplayDialog('Error',
17         'ROS plugin was not found.&&nSimulation will not run
18             properly',
19         simDlgStyle_ok ,false ,nil ,{0.8,0,0,0,0,0},{0.5,0,0,1,1,1})
20 else
21     -- Ok go on.
22 ... -- remainder lines are omitted

```

All plugins are loaded by executing the `simLoadModule` function, however, ROS plugins are loaded automatically during V-REP startup, i.e. the library `libv_repExtRos.so` is loaded automatically. This is achieved because the shared ROS libraries were generated and copied to the V-REP directory in Sect. 4.2.

The scripts in V-REP are divided into sections. At simulation time, all scripts are executed within the internal main loop. Some sections are executed once, whereas others are performed on each loop iteration. The script fragment presented in Listing 1.20 executes once on each simulation. For publishing the GPS data as topic to `roscore`, a special Lua function is called. There is a variety of Lua functions provided by V-REP team in order to work with ROS and others communication channels.

Listing 1.20 preencher

```

1 ... // previous lines are omitted
2 -- Publish the Euler angle as ROS topic
3 topicName=simExtROS_enablePublisher('eulerX' ,1 ,
        simros_strmcmd_get_float_signal ,-1,-1,'eulerX' ,0

```

```

4 topicName=simExtROS_enablePublisher('eulerY',1,
5     simros_strmcmd_get_float_signal,-1,-1,'eulerY',0)
6 topicName=simExtROS_enablePublisher('eulerZ',1,
7     simros_strmcmd_get_float_signal,-1,-1,'eulerZ',0)
8 ... // next lines are omitted

```

The `simExtROS_enablePublisher` function is used to enable a publisher on V-REP. The parameters are similar to the function used for publishing data by means of `ros::Publisher.advertise` method as follows:

1. The name of the target topic to which data is published, e.g. “eulerX”.
2. The queue size has the same meaning of the original queue size of ROS publisher.
3. The stream data type parameter is used define how to process the two following parameters, e.g. the user can use the `simros_strmcmd_get_float_signal` signal to publish floating-point data. There is a variety of predefined data types.
4. The meaning of `auxiInt1` parameter depends on the data type. When this parameter is not in use, the value is `-1`.
5. The `auxiInt2` parameter semantics is similar to `auxiInt1`.
6. The `auxString` parameter. The type `simros_strmcmd_get_float_signal` means that a float type from a V-REP signal is being published. This parameter must match with the signal name. Listing 1.21 depicts the GPS script code that is used to explain how a V-REP signal is declared within a Lua script.
7. The `publishCnt` parameter indicates the number of times a signal is published before it goes to sleep. The `-1` value lead to start the sleep mode, whereas values greater than zero indicates that data are published exactly `publishCnt` times. The publisher wakes up when `simExtROS_wakePublisher` is executed. All published data never sleep by setting this parameter to zero.

Some published or subscribed data types use the parameters `auxiInt1` or `auxiInt2`. For example, the `simros_strmcmd_get_joint_state` type was used to get the joint state. It uses the `auxiInt1` to indicate the joint handle. Other type is `simros_strmcmd_get_object_pose` which is used to enable data streaming from the object pose. This type uses the `auxiInt1` to identify V-REP object handle and the `auxiInt2` indicates the reference frame from which the pose is obtained. For more information please see [31].

Listing 1.21 presents a code fragment of the virtual GPS script. These lines create three distinct signals related to the object position information. The `objectAbsolutePosition` variable is a Lua vector with values calculated before in this fragment execution.

Listing 1.21 Fragment of Lua script of the virtual GPS

```

1 ... // previous lines are omitted
2 simSetFloatSignal('gpsX',objectAbsolutePosition[1])
3 simSetFloatSignal('gpsY',objectAbsolutePosition[2])
4 simSetFloatSignal('gpsZ',objectAbsolutePosition[3])
5 ... // next lines are omitted

```

5.4 Subscribing to ROS Topics

A ROS node (e.g. the hexacopter main controller) may subscribe to ROS topics in order to receive data published by other nodes, e.g. the sensor in the V-REP. Thus, a virtual object can be controlled during simulation by means of subscribing ROS topics within V-REP scripts. For instance, the rotor of the virtual hexacopter must receive throttle signals published by the ROS node created in the `main()` function in `rosvrep_controller.cpp` file (see Sect. 4.5). Listing 1.22 shows the fragment of `HexacopterPlus` object script that enables V-REP to subscribe topics and receive the ROS messages.

Listing 1.22 Fragment of Lua script of the `HexacopterPlus` object

```

1 ... // previous lines are omitted
2 -- Rotor Subscribers
3 simExtROS_enableSubscriber('propFRONT', 1,
    simros_strmcmd_set_float_signal, -1,-1, 'propFRONT')
4 simExtROS_enableSubscriber('Yaw', 1,
    simros_strmcmd_set_float_signal, -1,-1, 'Yaw')
5 ... // next lines are omitted

```

The parameters of `simExtROS_enableSubscriber` function are similar to the `simExtROS_enablePublisher` function (see Sect. 5.3), however, there is a difference in the specification on how data are handled. The `simros_strmcmd_set_float_signal` parameter indicates that V-REP subscribes to the topic, while `simros_strmcmd_get_float_signal` indicates that V-REP publishes in the topic. The last parameter is a signal that is used in a Lua script associated with any objects from V-REP virtual environment. For instance, `propFRONT` signal is used in the script of `propeller_jointFRONT` object by means of calling `simGetFloatSignal` function in the parameters list of `simSetJointTargetVelocity` function as shown in Listing 1.23. A V-REP signal is a global variable. When the `simExtROS_enableSubscriber` is executed, a value is assigned to that global variable. If such a global variable does not exist, the `simExtROS_enableSubscriber` creates it.

Listing 1.23 Fragment of Lua script of the `HexacopterPlus` object

```

1 ... // previous lines are omitted
2 simSetJointTargetVelocity(simGetObjectHandle(
    'propeller_jointFRONT'),
    simGetFloatSignal('propFRONT') * -200)
3 ...
4 ... // next lines are omitted

```

5.5 Publishing Images from V-REP

Many robotic applications usually demand some sort of video processing in order to perform advanced tasks. Cameras are commonly used in computational vision tasks, e.g. for collision avoidance while the robot is moving or for mapping and navigating towards the environment [32]. Thus, it is important to provide means

for video processing during the simulation phase of a robot design. This section presents how to setup a virtual video camera in V-REP and how to stream the capture video to a ROS node.

It is possible to many distinct data types within topics published or subscribe between ROS and V-REP, including images from the virtual video sensor. A ROS node receives an image that can be processed using the OpenCV API [33]. Although the OpenCV library is not a part of ROS, `vision_opencv` package [34] provides an interface between ROS and the OpenCV library. This package was used in the camera application⁶ implemented in `rosvrep_camera.cpp`. Although this tutorial does not discuss video processing, we show how to setup a ROS topic and stream the video captured within the V-REP simulated environment. For that, the *HexacopterPlus* has an object named *vCamera* attached onto its frame. *vCamera* object is a video sensor that streams images captured during simulation. Using the `simros_strmcmd_get_vision_sensor_image` type, V-REP is able to send images to a ROS node. Listing 1.24 depicts a fragment of the Lua script from *HexacopterPlus* object. The video streamed from the virtual camera can be seen in the camera application.

Listing 1.24 Fragment of Lua script from the *HexacopterPlus* object

```

1 ... // previous lines are omitted
2 vCameraHandle=simGetObjectHandle('vCamera')
3 topicName=simExtROS_enablePublisher('vCamera',1,
4     simros_strmcmd_get_vision_sensor_image,
5     vCameraHandle,0,'')
5 ...

```

5.6 Running the Sample Scenarios

Our package provides two scenes that are located in the `scenes` subdirectory. The first scene was modeled in `Hexacopter.ttt` file. It was created to illustrate how the hexacopter was created in V-REP. The second scene was modeled in `rosHexaPlus_scene.ttt`. This is a more elaborated scene whose environment presents trees and textures. The aim is to illustrate the hexacopter movements, and hence, it is the scene used in the rest of this section.

Before starting the scene execution, the reader must ensure that `roscore` and V-REP are running (see Sect. 4.2). In the V-REP, open the `rosHexaPlus_scene.ttt` file using the menu command `File → Open scene`. The reader must start the simulation by either choosing the menu option `Simulation → Start Simulation` or by clicking on `Start/resume simulation` in the toolbar.

Go to the terminal that is executing the `rosvrep_panel` application as shown in Fig. 18. The first set of coordinates must be inserted in the order presented in Table 2, aiming to command the hexacopter to fly around the environment. It is

⁶This application was created for debugging purposes and it is not discussed in this chapter.

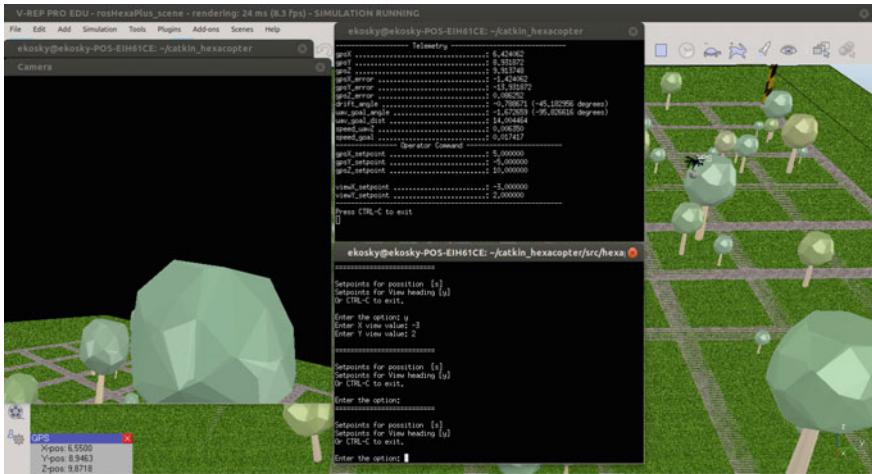


Fig. 18 A screenshot of the test

Table 2 First test: hexacopter flying around the environment

Commands sequence	Target coordinates			Heading	
	X	Y	Z	X	Y
1	-15	15	3	-17	11
2	-7	10	10	-17	11
3	7	10	10	-3	2
4	2	-3.5	10	-3	2
5	2	-3.5	0	-3	2

important to notice that the hexacopter carry a free payload. Insert the target position by using the “s” option and then heading directions using the “y” option. The next target position should be sent only after the hexacopter reaches the position indicated in previous command. The reader can see the execution using these coordinates in the youtube video <https://youtu.be/Pvve5IFz4e4>. This video shows a long distance movement.

The second simulation shows a flight in which the hexacopter moves to short distance target position. Figure 19 shows a screenshot on which one can see the hexacopter behavior carrying a free payload. In this second test, the reader should insert the commands shown in Table 3. The video of this test can be seen in in <https://youtu.be/7n8tThctAns>.



Fig. 19 The V-REP screenshot of short distance execution

Table 3 First test: hexacopter flying around the environment

Commands sequence	Target coordinates			Heading	
	X	Y	Z	X	Y
1	-20	18	3	-18	23
2	-18	18	3	-10	18
3	-23	22	5	-10	18

6 Final Remarks

This chapter describes a tutorial on how to implement a control system based on Fuzzy Logic. The movement control system of an hexacopter is used as a case study. A ROS package that includes a fuzzy library was presented. By using such a package, we discussed how to integrate the commercial robotics simulation environment named V-REP with a fuzzy control system implemented using ROS infrastructure. Instructions on how to perform a simulation with V-REP were presented. Therefore, this tutorial provides additional knowledge on using different tools for designing ROS-based systems.

This tutorial can be used as a starting point to make more experiences. The reader can modify or improve the proposed fuzzy control system by changing the “.fz” files. There is no need to modify the controller main controller implementation in `rosvrep_controller.cpp` file. As a suggestion to further improve the skills on using the propose fuzzy package and V-REP, the reader could create another ROS node which may act as a mission controller by sending automatically a set of target positions.

References

1. Koslosky, E., et al. Hexacopter tutorial package. <https://github.com/ekosky/hexaplus-ros-tutorial.git>. Accessed Nov 2016.
2. Bipin, K., V. Duggal, and K.M. Krishna. 2015. Autonomous navigation of generic monocular quadcopter in natural environment. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 1063–1070.
3. Haque, M.R., M. Muhammad, D. Swarnaker, and M. Arifuzzaman. 2014. Autonomous quadcopter for product home delivery. In *2014 International Conference on Electrical Engineering and Information Communication Technology (ICEEICT)*, 1–5.
4. Leishman, R., J. Macdonald, T. McLain, and R. Beard. 2012. Relative navigation and control of a hexacopter. In *2012 IEEE International Conference on Robotics and Automation (ICRA)*, 4937–4942.
5. Ahmed, O.A., M. Latief, M.A. Ali, and R. Akmeliawati. 2015. Stabilization and control of autonomous hexacopter via visual-servoing and cascaded-proportional and derivative (PD) controllers. In *2015 6th International Conference on Automation, Robotics and Applications (ICARA)*, 542–549.
6. Alaimo, A., V. Artale, C.L.R. Milazzo, and A. Ricciardello. 2014. PID controller applied to hexacopter flight. *Journal of Intelligent & Robotic Systems* 73 (1–4): 261–270.
7. Ołdziej, D., and Z. Gosiewski. 2013. Modelling of dynamic and control of six-rotor autonomous unmanned aerial vehicle. *Solid State Phenomena* 198: 220–225.
8. Collotta, M., G. Pau, and R. Caponetto. 2014. A real-time system based on a neural network model to control hexacopter trajectories. In *2014 International Symposium on Power Electronics, Electrical Drives, Automation and Motion (SPEEDAM)*, 222–227.
9. Artale, V., C.L. Milazzo, C. Orlando, and A. Ricciardello. 2015. Genetic algorithm applied to the stabilization control of a hexarotor. In *Proceedings of the International Conference on Numerical Analysis and Applied Mathematics 2014 (ICNAAM-2014)*, 222–227.
10. Bacik, J., D. Perdukova, and P. Fedor. 2015. Design of fuzzy controller for hexacopter position control. *Artificial Intelligence Perspectives and Applications*, 193–202. Berlin: Springer.
11. Koslosky, E., M.A. Wehrmeister, J.A. Fabro, and A.S. Oliveira. 2016. On using fuzzy logic to control a simulated hexacopter carrying an attached pendulum. In *Designing with Computational Intelligence*, vol. 664, ed. N. Nedjah, H.S. Lopes, and L.M. Mourelle. Studies in Computational Intelligence. Berlin: Springer. 01–32 *Expected publication on Dec. 2016*.
12. Open Source Robotics Foundation: ROS basic tutorials. <http://wiki.ros.org/ROS/Tutorials>. Accessed March 2016.
13. Coppelia Robotics: V-REP: Virtual robot experimentation platform. <http://www.coppeliarobotics.com>. Accessed March 2016.
14. Coppelia Robotics: V-REP bubblerob tutorial. <http://www.coppeliarobotics.com/helpFiles/en/bubbleRobTutorial.htm>. Accessed March 2016.
15. Coppelia Robotics: V-REP tutorial for ROS indigo integration. <http://www.coppeliarobotics.com/helpFiles/en/rosTutorialIndigo.htm>. Accessed March 2016.
16. Yoshida, K., I. Kawanishi, and H. Kawabe. 1997. Stabilizing control for a single pendulum by moving the center of gravity: theory and experiment. In *American Control Conference, 1997. Proceedings of the 1997*, vol. 5, 3405–3410.
17. Passino, K.M., and S. Yurkovich. 1998. *Fuzzy Control*. Reading: Addison-Wesley.
18. Hwang, G.C., and S.C. Lin. 1992. A stability approach to fuzzy control design for nonlinear systems. *Fuzzy Sets and Systems* 48 (3): 279–287.
19. Pedro, J.O., and C. Mathe. 2015. Nonlinear direct adaptive control of quadrotor UAV using fuzzy logic technique. In *2015 10th Asian Control Conference (ASCC)*, 1–6.
20. Pedrycz, W., and F. Gomide. 2007. *RuleBased Fuzzy Models*, 276–334. New York: Wiley-IEEE Press.
21. Open Source Robotics Foundation: ROS remapping. <http://wiki.ros.org/Remapping%20Arguments>. Accessed March 2016.

22. Chak, Y.C., and R. Varatharajoo. 2014. A heuristic cascading fuzzy logic approach to reactive navigation for UAV. *IIUM Engineering Journal, Selangor - Malaysia* 15 (2).
23. Sureshkumar, V., and K. Cohen. Autonomous control of a quadrotor UAV using fuzzy logic. *Unisys Digita - Journal of Unmanned System Technology, Cincinnati, Ohio*.
24. Eusebiu Marcu, C.B. UAV fuzzy logic control system stability analysis in the sense of Lyapunov. *UPB Scientific Bulletin, Series D* 76 (2).
25. Abeywardena, D.M.W., L.A.K. Amaralunga, S.A.A. Shakoor, and S.R. Munasinghe. 2009. A velocity feedback fuzzy logic controller for stable hovering of a quad rotor UAV. In *2009 International Conference on Industrial and Information Systems (ICIIS)*, 558–562.
26. Gomez, J.F., and M. Jamshidi. 2011. Fuzzy adaptive control for a UAV. *Journal of Intelligent & Robotic Systems* 62 (2): 271–293.
27. Limnaios, G., and N. Tsourveloudis. 2012. Fuzzy logic controller for a mini coaxial indoor helicopter. *Journal of Intelligent & Robotic Systems* 65 (1): 187–201.
28. Ierusalimschy, R., W. Celes, and L.H. de Figueiredo. 2016. Lua documentation. <https://www.lua.org/>. Accessed March 2016.
29. Coppelia Robotics: V-REP help. <http://www.coppeliarobotics.com/helpFiles/>. Accessed March 2016.
30. Coppelia Robotics: V-REP download page. <http://www.coppeliarobotics.com/downloads.html>. Accessed March 2016.
31. Coppelia Robotics: ROS publisher typer for V-REP. <http://www.coppeliarobotics.com/helpFiles/en/rosPublisherTypes.htm>. Accessed March 2016.
32. Steder, B., G. Grisetti, C. Stachniss, and W. Burgard. 2008. Visual SLAM for flying vehicles. *IEEE Transactions on Robotics* 24 (5): 1088–1093.
33. Itseez: OpenCV - Open Source Computer Vision Library. <http://opencv.org/>. Accessed Nov 2016.
34. Mihelich, P., and J. Bowman. 2016. *vision_opencv* documentation. Accessed March 2016.

Author Biographies

Emanoel Koslosky is Master's degree student in the applied computing and embedded systems. As a student, he took classes about Mobile Robotics, Image Processing, Hardware Architecture for Embedded Systems, Operating Systems in Real Time. As a professional, he received Certifications of Oracle Real Application Clusters 11g Certified Implementation - Specialist, Oracle Database 10g Administrator Certified Professional - OCP, Oracle8i Database Administrator Certified Professional - OCP. He has professionally worked as programmer and developer since 1988 using languages such as C/C++, Oracle Pro*C/C++, Pro*COBOL, Java and Oracle Tools like Oracle Designer, Oracle Developer, as a Database Administrator worked with high availability and scalability environment, also as a System Administrator of Oracle e-Business Suite - EBS.

Marco Aurélio Wehrmeister received the Ph.D. degree in Computer Science from the Federal University of Rio Grande do Sul (Brazil) and the University of Paderborn (Germany) in 2009 (double-degree). In 2009, he worked as Lecturer and Postdoctoral Researcher for the Federal University of Santa Catarina (Brazil). From 2010 to 2013, he worked as tenure track Professor with the Department of Computer Science from the Santa Catarina State University (UDESC, Brazil). Since 2013, he has been working as a tenure track Professor with the Department of Informatics from the Federal University of Technology - Paraná (UTFPR, Brazil). From 2014 to 2016, he was the Head of the M.Sc. course on Applied Computing of UTFPR. In 2015, Prof. Dr. Wehrmeister was a Visiting Fellow (short stay) with School of Electronic, Electrical and Systems Engineering from the University of Birmingham (UK). Prof. Dr. Wehrmeister's thesis was selected by the Brazilian Computer Society as one of the six best theses on Computer Science in 2009. He is member of the special commission on Computing Systems Engineering of the Brazilian Computer Society. Since 2015, he is a member of the IFIP Working Group 10.2 on Embedded Systems.

His research interests are in the areas of embedded and real-time systems, aerial robots, model-driven engineering, and hardware/software engineering for embedded systems and robotics. Prof. Dr. Wehrmeister has co-authored more than 70 papers in international peer-reviewed journals and conference proceedings. He has been involved in various research projects funded by Brazilian R&D agencies.

Andre Schneider de Oliveira holds a degree in Computer Engineering from the University of Vale do Itajaí (2004), master's degree in Mechanical Engineering from the Federal University of Santa Catarina (2007) and Doctorate in Automation and Systems Engineering from the Federal University of Santa Catarina (2011). He is currently Assistant Professor at the Federal Technological University of Paraná - Curitiba campus. He has carried out research in Electrical Engineering with emphasis on Robotics, Mechatronics and Automation, working mainly with the following topics: navigation and positioning of mobile robots; autonomous and intelligent systems; perception and environmental identification; and control systems for navigation.

João Alberto Fabro is an Associate Professor at Federal University of Technology - Paraná (UTFPR), where he has been working since 2008. From 1998 to 2007, he was with the State University of West-Paraná (UNIOESTE). He has an undergraduate degree in Informatics, from Federal University of Paraná (UFPR 1994), a Master's Degree in Computing and Electric Engineering, from Campinas State University (UNICAMP 1996), a Ph.D. degree in Electric Engineering and Industrial Informatics(CPGEI) from UTFPR (2003) and recently actuated as a Post-Doc at the Faculty of Engineering, University of Porto, Portugal (FEUP, 2014). He has experience in Computer Science, specially Computational Intelligence, actively researching on the following subjects: Computational Intelligence (neural networks, evolutionary computing and fuzzy systems), and Autonomous Mobile Robotics. Since 2009, he has participated in several Robotics Competitions, in Brazil, Latin America and World Robocup, both with soccer robots and service robots.

Flying Multiple UAVs Using ROS

Wolfgang Hönig and Nora Ayanian

Abstract This tutorial chapter will teach readers how to use ROS to fly a small quadcopter both individually and as a group. We will discuss the hardware platform, the Bitcraze Crazyflie 2.0, which is well suited for swarm robotics due to its small size and weight. After first introducing the `crazyflie_ros` stack and its use on an individual robot, we will extend scenarios of hovering and waypoint following from a single robot to the more complex multi-UAV case. Readers will gain insight into physical challenges, such as radio interference, and how to solve them in practice. Ultimately, this chapter will prepare readers not only to use the stack as-is, but also to extend it or to develop their own innovations on other robot platforms.

Keywords ROS · UAV · Multi-Robot-System · Crazyflie · Swarm

1 Introduction

Unmanned aerial vehicles (UAVs) such as AscTec Pelican, Parrot AR.Drone, and Erle-Copter have a long tradition of being controlled with ROS. As a result, there are many ROS packages devoted to controlling such UAVs as individuals.¹ However, using multiple UAVs creates entirely new challenges that such packages cannot address, including, but not limited to, the physical space required to operate the robots, the interference of sensors and network communication, and safety requirements.

Multiple UAVs have been used in recent research [1–5], but such research can be overly complicated and tedious due to the lack of tutorials and books. In fact,

¹E.g., http://wiki.ros.org/ardrone_autonomy, <http://wiki.ros.org/mavros>, http://wiki.ros.org/asc tec_mav_framework.

W. Hönig (✉) · N. Ayanian

Department of Computer Science, University of Southern California,
Los Angeles, CA, USA

email: whoenig@usc.edu
URL: <http://act.usc.edu>

N. Ayanian

e-mail: ayanian@usc.edu

even with packages that can support multiple UAVs, documentation focuses on the single UAV case, not considering the challenges that occur once multiple UAVs are used. Research publications often skip implementation details, making it difficult to replicate the results. Papers about specialized setups exist [6, 7], but rely on expensive or commercially unavailable solutions.

This chapter will attempt to fill this gap in documentation. In particular, we try to provide a step-by-step guide on how to reproduce results we presented in an earlier research paper [3], which used up to six UAVs.² We focus on a small quadcopter — the Bitcraze Crazyflie 2.0 — and how to use it with the `crazyflie_ros` stack, particularly as part of a group of 2 or more UAVs. We will assume that an external position tracking system, such as a motion capture system, is available because the Crazyflie is not able to localize itself with just onboard sensing. We will discuss the physical setup and how to support a single human pilot. Each step will start with the single UAV case and then extend to the more challenging multi-UAV case.

We begin with an introduction to the target platform, including the software setup of the vendor’s software and the `crazyflie_ros` stack. We then show teleoperation of multiple Crazyflies using joysticks. The usage of a motion capture system allows us to autonomously hover multiple Crazyflies. We then extend this to multiple UAVs following waypoints. The chapter will also contain important insights into the `crazyflie_ros` stack, allowing the user to understand the design in-depth. This can be helpful for users interested in implementing other multi-UAV projects using different hardware or adding extensions to the existing stack.

Everything discussed here has been tested on Ubuntu 14.04 using ROS Indigo. The stack and discussed software also work with ROS Jade (Ubuntu 14.04) and ROS Kinetic (Ubuntu 16.04).

2 Target Platform

As our target platform we use the Bitcraze Crazyflie 2.0 platform, an open-source, open-hardware nano quadcopter that targets hobbyists and researchers alike. Its small size (92 mm diagonal rotor-to-rotor) and weight (29 g) make it ideal for indoor swarming applications. Additionally, its size allows users to operate the UAVs safely even with humans or other robots around. The low inertia causes only few parts to break after a crash — the authors had several crashes from a height of 3 m to a concrete floor with damage only to cheaply replaceable plastic parts. A Crazyflie can communicate with a phone or PC using BlueTooth. Additionally, a custom USB dongle called Crazyradio PA, or Crazyradio for short, allows lower latency communication. The Crazyflie 2.0 and Crazyradio PA are shown in Fig. 1.

A block diagram of the Crazyflie’s architecture is shown in Fig. 2. The communication system is used to send the setpoint, consisting of thrust and attitude, tweak internal parameters, and stream telemetry data, such as sensor readings. It is also

²Video available at <http://youtu.be/px9iHkA0nOI>.



Fig. 1 Our target platform Bitcraze Crazyflie 2.0 quadcopter (*left*), which can be controlled from a PC using a custom USB dongle called Crazyradio PA (*right*). Image credit: Bitcraze AB

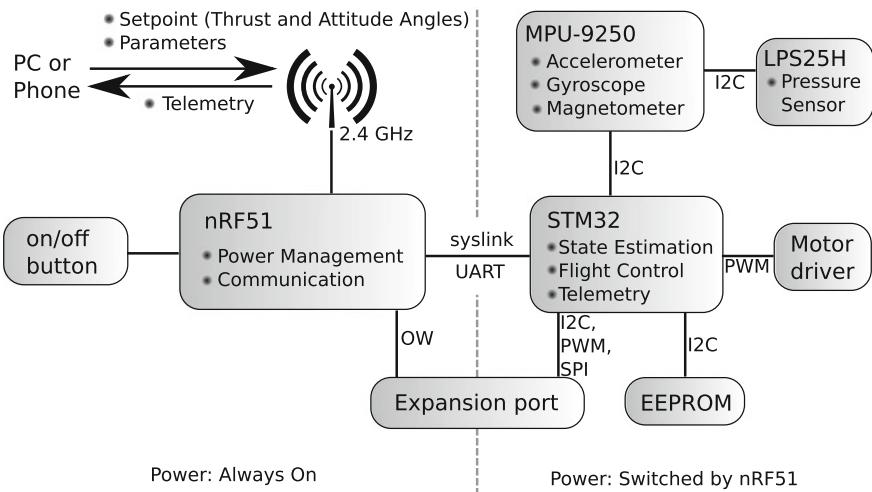


Fig. 2 Components and architecture of the Crazyflie 2.0 quadcopter. Based on images by Bitcraze AB

possible to update the onboard software wirelessly. The Crazyflie has a 9-axis inertial measurement unit (IMU) onboard, consisting of gyroscope, accelerometer, and magnetometer. Moreover, a pressure sensor can be used to estimate the height. Most of the processing is done on the main microcontroller (STM32). It runs FreeRTOS as its operating system and state estimation and attitude control are executed at 250 Hz. A second microcontroller (nRF51) is used for the wireless communication and as a power manager. The two microcontrollers can exchange data over the syslink, which is a protocol using UART as a physical interface. An extension port permits the addition of additional hardware. The official extensions include an inductive charger, LED headlights, and buzzer. Finally, it is possible to use the platform on a bigger frame if higher payload capabilities are desired. Extensions are called “decks” and are also used by the community to add additional capabilities.³ The schematics

³<https://www.hackster.io/bitcraze/products/crazyflie-2-0>.

as well as all firmwares are publicly available.⁴ The technical specifications are as follows:

- STM32F405: main microcontroller, used for state-estimation, control, and handling of extensions. We will call this STM32.
(Cortex-M4, 168 MHz, 192 kB SRAM, 1 MB flash).
- nRF51822: radio and power management microcontroller. We will call this nRF51.
(Cortex-M0, 32 MHz, 16 kB SRAM, 128 kB flash).
- MPU-9250: 9-axis inertial measurement unit.
- LPS25H: pressure sensor.
- 8 kB EEPROM.
- uUSB: charging and wired communication.
- Expansion port (I2C, UART, SPI, GPIO).
- Debug port for STM32. An optional debug-kit can be used to convert to a standard JTAG-connector and to debug the nRF51 as well.

The onboard sensors are sufficient to stabilize the attitude, but not the position. In particular, external feedback is required to fly to predefined positions. By default, this is the human who teleoperates the quadcopter either using a joystick connected to a PC, or a phone. In this chapter, we will use a motion-capture system for fully autonomous flights.

The vendor provides an SDK written in Python which runs on Windows, Linux, and Mac. It can be used to teleoperate a single Crazyflie using a joystick, to plot sensor data in real-time, and to write custom applications. We will use ROS in the remainder of this chapter to control the Crazyflie; however, ROS is only used on the PC controlling one or more Crazyflies. The ROS driver sends the data to the different quadcopters using the protocol defined in the Crazyflie firmware.

The Crazyflie has been featured in a number of research papers. The mathematical model and system identification of important parameters, such as the inertia matrix, have been discussed in [8, 9]. An updated list with applications can be found on the official webpage.⁵

3 Setup

In this section we will describe how to set up the Crazyflie software. We cover both the official Python SDK and how to install the `crazyflie_ros` stack. The first is useful to reconfigure the Crazyflie as well as for troubleshooting, while the later will allow us to use multiple Crazyflies with ROS.

We assume a PC with Ubuntu 14.04 as operating system, which has ROS Indigo (desktop-full) installed.⁶ It is better to install Ubuntu directly on a PC rather than

⁴<https://github.com/bitcraze/>.

⁵<https://www.bitcraze.io/research/>.

⁶<http://wiki.ros.org/indigo/Installation/Ubuntu>.

using a virtual machine for two reasons: First, you will be using graphical tools, such as `rviz`, which rely on OpenGL and therefore do not perform as well on a virtual machine as when natively installed. Second, the communication using the Crazyradio would have additional latency in a virtual machine since the USB signals would go through the host system first. This might cause less stable control.

In particular, we will follow the following steps:

1. Configure the PC such that the Crazyradio will work for any user.
2. Install the official software package to test the Crazyflie.
3. Update Crazyflie's onboard software to the latest version to ensure that it will work with the ROS package.
4. Install the `crazyflie_ros` package and run a first simple connection test.

The later sections in this chapter assume that everything is set up as outlined here to perform higher-level tasks.

3.1 Setting PC Permissions

By default, the Crazyradio will only work for a user with superuser rights when plugged in to a PC. This is not only a security concern but also makes it harder to use with ROS. In order to use it without `sudo`, we first add a group (`plugdev`) and then add ourselves as a member of that group:

```
$ sudo groupadd plugdev  
$ sudo usermod -a -G plugdev $USER
```

Now, we create a udev-rule, setting the permission such that anyone who is a member of our newly created group can access the Crazyradio. We create a new rules file using `gedit`:

```
$ sudo gedit /etc/udev/rules.d/99-crazyradio.rules
```

and add the following text to it:

```
1 # Crazyradio (normal operation)  
2 SUBSYSTEM=="usb", ATTRS{idVendor}=="1915",  
    ATTRS{idProduct}=="7777", MODE="0664", GROUP="plugdev"  
3 # Bootloader  
4 SUBSYSTEM=="usb", ATTRS{idVendor}=="1915",  
    ATTRS{idProduct}=="0101", MODE="0664", GROUP="plugdev"
```

The second entry is useful for firmware updates of the Crazyradio.

In order to use the Crazyflie when directly connected via USB, you need to create another file named `99-crazyflie.rules` in the same folder, with the following content:

```
1 SUBSYSTEM=="usb", ATTRS{idVendor}=="0483",
    ATTRS{idProduct}=="5740", MODE=="0664", GROUP="plugdev"
```

Finally, we reload the udev-rules:

```
$ sudo udevadm control --reload-rules
$ sudo udevadm trigger
```

You will need to log out and log in again in order to be a member of the `plugdev` group. You can then plug in your Crazyradio (and follow the instructions in the next section to actually use it).

3.2 Bitcraze Crazyflie PC Client

The Bitcraze SDK is composed of two parts. The first is `crazyflie-lib-python`, which is a Python library to control the Crazyflie without any graphical user interface. The second is `crazyflie-client-python`, which makes use of that library and adds a graphical user interface.

We start by installing the required dependencies:

```
$ sudo apt-get install git python3 python3-pip python3-pyqt4
    python3-numpy python3-zmq
$ sudo pip3 install pyusb==1.0.0b2
$ sudo pip3 install pyqtgraph appdirs
```

To install `crazyflie-lib-python`, use the following commands:

```
$ mkdir ~/crazyflie
$ cd ~/crazyflie
$ git clone
    https://github.com/bitcraze/crazyflie-lib-python.git
$ cd crazyflie-lib-python
$ pip3 install --user -e .
```

Here, the Python package manager `pip` is used to install the library only for the current user. The library uses Python 3. In contrast, ROS Indigo, Jade, and Kinetic use Python 2.

Similarly, `crazyflie-client-python` can be installed using the following commands:

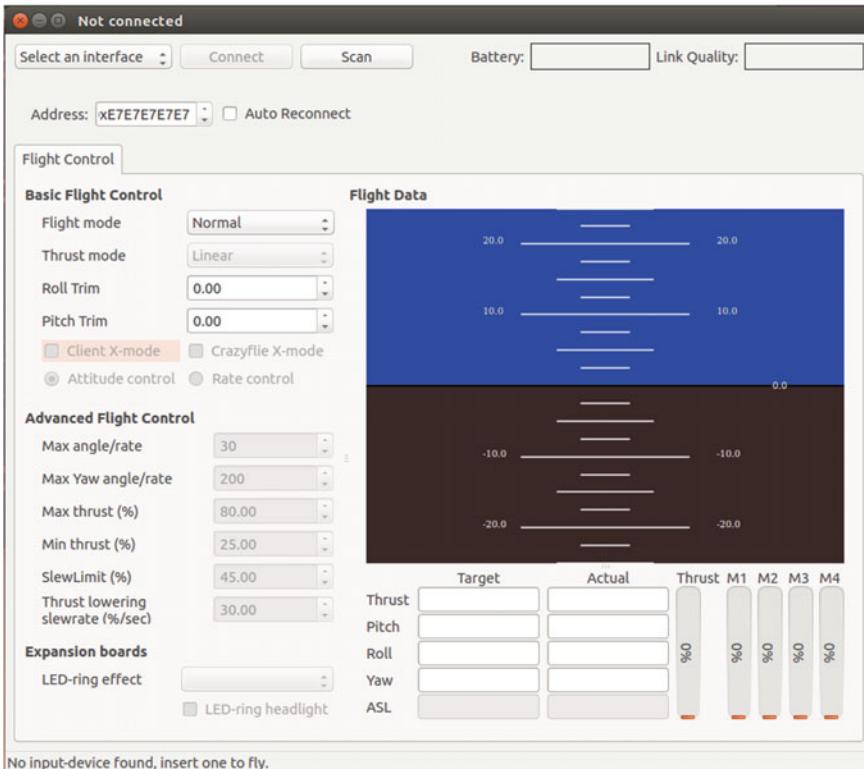


Fig. 3 Screenshot of the Bitcraze Crazyflie PC Client

```
$ cd ~/crazyflie
$ git clone
  https://github.com/bitcraze/crazyflie-clients-python.git
$ cd crazyflie-clients-python
$ pip3 install --user -e .
```

To start the client, execute the following:

```
$ cd ~/crazyflie/crazyflie-clients-python
$ python3 bin/cfclient
```

You should see the graphical user interface, as shown in Fig. 3.

Versions Might Change

Since the Crazyflie software is under active development, the installation procedure and required dependencies might change in the future. You can use the exact same versions as used in the chapter by using the following commands after `git clone`. Use the following for `crazyflie-lib-python`

```
$ git checkout a0397675376a57adf4e7c911f43df885a45690d1
```

and use the following for `crazyflie-clients-python`:

```
$ git checkout 2dff614df756f1e814538fbe78fe7929779a9846
```

If you want to use the latest version please follow the instructions provided in the `README.md` file in the respective repositories.

3.3 Firmware

Everything described in this chapter works with the Crazyflie’s default firmware. You can obtain the latest compiled firmware from the repository⁷ — this chapter was tested with the 2016.02 release. Make sure that you update the firmware for both STM32 and nRF51 chips by downloading the zip-file. Execute the following steps to update both firmwares:

1. Start the Bitcraze Crazyflie PC Client.
2. In the menu select “Connect”/“Bootloader.”
3. Turn your Crazyflie off by pressing the power button. Turn it back on by pressing the power button for 3 seconds. The blue tail lights should start blinking: The Crazyflie is now waiting for a new firmware.
4. Click “Initiate bootloader cold boot.” The status should switch to “Connected to bootloader.”
5. Select the downloaded `crazyflie-2016.02.zip` and press “Program.” Click the “Restart in firmware mode” button after it is finished.

If you prefer compiling the firmware yourself, please follow the instructions in the respective repositories.⁸

⁷<https://github.com/bitcraze/crazyflie-release/releases>.

⁸<https://github.com/bitcraze/crazyflie-firmware>,
<https://github.com/bitcraze/crazyflie2-nrf-firmware>.

3.4 *Crazyflie ROS Stack*

The `crazyflie_ros` stack contains the driver, a position controller, and various examples. We will explore the different possibilities later in this chapter and concentrate on the initial setup first.

We first create a new ROS workspace:

```
$ mkdir -p ~/crazyflie_ws/src
$ cd ~/crazyflie_ws/src
$ catkin_init_workspace
```

Next, we add the required packages to the workspace and build them:

```
$ git clone https://github.com/whoenig/crazyflie_ros.git
$ cd ~/crazyflie_ws
$ catkin_make
```

In order to use your workspace add the following line to your `~/.bashrc`:

```
$ source ~/crazyflie_ws/devel/setup.bash
```

This will ensure that all ROS related commands will find the packages in all terminals. To update your current terminal window, use `source ~/.bashrc`, which will reload the file.

You can test your setup by typing:

```
$ rosrun crazyflie_tools scan
```

This should print the *uniform-resource-identifier* (URI) of any Crazyflie found in range. For example, the output might look like this:

```
Configured Dongle with version 0.54
radio://0/100/2M
```

In this case, the URI of your Crazyflie is `radio://0/100/2M`. Each URI has several components. Here, the Crazyradio is used (*radio*). Since you might have multiple radios in use, you can specify a zero-based index on the device to use (*0*). The next number (*100*) specifies the channel, which is a number between 0 and 125. Finally, the datarate (*2M*) (one of 250K, 1M, 2M) specifies the speed to use in bits per second. There is an optional address as well, which we will discuss in Sect. 5.1.

Versions

As before, the instructions might be different in future versions. Use the following to get the exact same version of the `crazyflie_ros` stack:

```
$ git checkout 34beecd2a8d7ab02378bcd tcb9adf5a7a0eb50ea
```

Install the following additional dependency in order to use the teleoperation:

```
$ sudo apt-get install ros-indigo-hector-quadrotor-teleop
```

If you are using ROS Jade or Kinetic, you will need to add the package to your workspace manually.

4 Teleoperation of a Single Quadcopter

In this section we will use ROS to control a single Crazyflie using a joystick. Moreover, we will gain access to the internal sensors and show how to visualize the data using `rviz` and `rqt_plot`.

This is a useful first step to understand the `cmd_vel` interface of the `crazyflie_ros` stack. Later, we will build on this knowledge to let the Crazyflie fly autonomously. Furthermore, teleoperation is useful for debugging. For example, it can be used to verify that there is no mechanical hardware defect.

In the first subsection, we assume that you have access to a specific joystick, the Microsoft XBox360 controller. We show how to connect to the Crazyflie using ROS and how to eventually fly it manually. The second subsection relaxes this assumption by discussing the required steps needed to add support for another joystick.

4.1 Using an XBox360 Controller

For this example, we assume that you have an Xbox360 controller plugged into your machine. We will show how to use different joysticks later in this section. Use the following command to run the teleoperation example:

```
$ roslaunch crazyflie_demo teleop_xbox360.launch
uri:=radio://0/100/2M
```

Make sure that you adjust the URI based on your Crazyflie.

The launch file `teleop_xbox360.launch` has the following structure:

teleop_xbox360.launch

```

1 <launch>
2   <arg name="uri" default="radio://0/80/2M" />
3   <arg name="joy_dev" default="/dev/input/js0" />
4   <include file="$(find
5     crazyflie_driver)/launch/crazyflie_server.launch" />
6   <group ns="crazyflie">
7     <include file="$(find
8       crazyflie_driver)/launch/crazyflie_add.launch">
9       <arg name="uri" value="$(arg uri)" />
10      <arg name="tf_prefix" value="crazyflie" />
11      <arg name="enable_logging" value="True" />
12    </include>
13    <node name="joy" pkg="joy" type="joy_node"
14      output="screen" >
15      <param name="dev" value="$(arg joy_dev)" />
16    </node>
17    <include file="$(find
18      crazyflie_demo)/launch/xbox360.launch" />
19    <node name="crazyflie_demo_controller"
20      pkg="crazyflie_demo" type="controller.py"
21      output="screen" />
22  </group>
23  <node pkg="rviz" type="rviz" name="rviz" args="-d $(find
24    crazyflie_demo)/launch/crazyflie.rviz" />
25  <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot1"
26    args="/crazyflie/battery"/>
27  <node pkg="rqt_plot" type="rqt_plot" name="rqt_plot2"
28    args="/crazyflie/rssi"/>
29 </launch>
```

In line 4 the `crazyflie_server` is launched, which accesses the Crazyradio to communicate with the Crazyflie. Lines 5–16 contain information about the Crazyflie we want to control. First, the Crazyflie is added with a specified URI. Second, the `joy_node` is launched to create the `joy` topic. This particular joystick is configured by including `xbox360.launch`. This file will launch a `hector_quadcopter_teleop` node with the appropriate settings for the XBox360 controller. Furthermore, `controller.py` is started; this maps additional joystick buttons to Crazyflie specific behaviors. For example, the red button on your controller will cause the Crazyflie to turn off all propellers (emergency mode). Finally, lines 17–19 start `rviz` and two instances of `rqt_plot` for visualization. Figure 4 shows a screenshot of `rviz` as it visualizes the data from the inertial measurement unit as streamed from the Crazyflie at 100Hz. The other two `rqt_plot` instances show the current battery voltage and radio signal strength indicator (RSSI), respectively.

If you tilt your Crazyflie, you should instantly see the IMU arrow changing in `rviz`. You can now use the joystick to fly — the default uses the left stick for thrust (up/down) and yaw (left/right) and the right stick for pitch (up/down) and roll

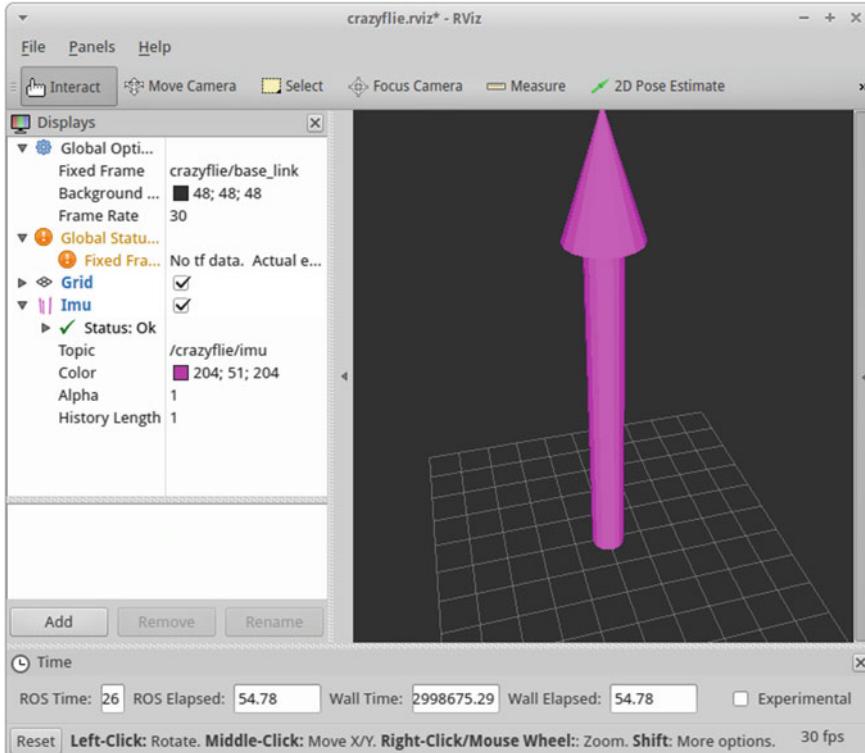


Fig. 4 Screenshot of `rviz` showing the IMU data

(left/right). Also, the red B-button can be used to put the ROS driver in emergency mode. In that case, your Crazyflie will immediately turn off its engines (and if it was flying it will fall to the ground).

4.2 Add Support for Another Controller

Support for another joystick can be easily added, as long as it is recognized as a joystick by the operating system. The major difference between joysticks is the mapping between the different axes and buttons of a joystick to the desired functionality. In the following steps we first try to find the desired mapping and use that to configure the `crazyflie_ros` stack accordingly.

1. Attach your joystick. This will create a new device file, e.g., `/dev/input/js0`. You can use `dmesg` to find details about which device file was used in the system log.
2. Run the following command in order to execute the `joy_node`:

```
$ rosrun joy joy_node _dev:=/dev/input/js0
```

3. In another terminal, execute:

```
$ rostopic echo /joy
```

This will print the joystick messages published by `joy_node`. Move your joystick to find the desired axes mapping. For example, you might increase the thrust on your joystick and see that the second number of the `axes` array decreases.

4. Change the axis mapping in `xbox360.launch` (or create a new file) by updating parameters `x_axis`, `y_axis`, `z_axis`, and `yaw_axis` accordingly. You can use negative axis values to indicate that this axis should be inverted. For example, in the previous example for the thrust changes, you would choose `-2` as axis for `z_axis`.
5. Update the button mapping in `crazyflie_demo/scripts/controller.py` to change which button triggers high-level behavior such as emergency.

The PS3 controller is already part of the `crazyflie_ros` stack and the mapping was found in the same way as described above.

5 Teleoperation of Multiple UAVs

This section discusses the initial setup: how to assign unique addresses to each UAV, how to communicate using fewer radios than UAVs, and how to find good communication channels to decrease interference between UAVs as well as between UAVs and existing infrastructure such as WiFi.

Flying multiple Crazyflies is mainly limited by the communication bandwidth. One way to handle this issue is to have one Crazyradio per Crazyfly and to use a different channel for each of them. There are two major disadvantages to this approach:

- The number of USB ports on a computer is limited. Even if you would add additional USB hubs, this adds additional latency because USB operates serially.
- There are 125 channels available; however, not all of them might lead to good performance since the 2.4 GHz band is shared. For example, BlueTooth and WiFi operate in the same band.

Therefore, we will use a single Crazyradio to control multiple Crazyflies and share the channels used. Hence, we will need to assign unique addresses to each Crazyfly to avoid crosstalking between the different quadcopters.

5.1 Assigning a Unique Address

The communication chips used in the Crazyflie and Crazyradio (nRF51 and nRF24LU1+ respectively) permit 40-bit addresses. By default, each Crazyflie has 0xE7E7E7E7E7 assigned as address. You can use the Bitcraze PC Client to change the address using the following steps:

1. Start the Bitcraze PC Client.
2. Make sure the address field is set to 0xE7E7E7E7E7 and click “Scan.” The drop-down box containing “Select an interface” should now have another entry containing the URI of your Crazyflie, for example `radio://0/100/2M` (See Fig. 5, left). Select this entry and click “Connect.”
3. In the “Connect” menu, select the item “Configure 2.0.” In the resulting dialog (see Fig. 5, right) change the address to a unique number, for example 0xE7E7E7E701 for your first Crazyflie, 0xE7E7E7E702 for the second one and so on. Select “Write” followed by “Exit.”
4. In the PC Client, select “Disconnect.”
5. Restart your Crazyflie.
6. Update the address field of the client (1 in Fig. 5, left) and click “Scan.” If everything was successful, you should now see a longer URI in the drop-down box containing `radio://0/100/2M/E7E7E7E701`.

If it does not work, verify that you have the latest firmware for both nRF51 and STM32 flashed. This feature might not be available or working properly otherwise. The address (and other radio parameters) are stored in EEPROM and therefore will remain even if you upgrade the firmware.

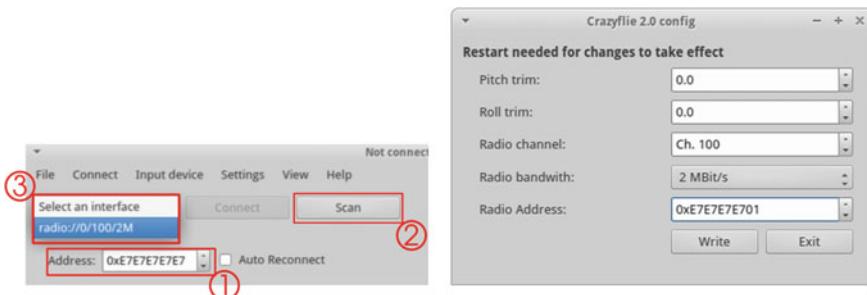


Fig. 5 *Left* To connect to a Crazyflie, first enter its address, click “Scan”, and finally select the found Crazyflie in the drop-down box. *Right* The configuration dialog for the Crazyflie to update radio related parameters

Scanning Limitation

The scan feature of both ROS driver and Bitcraze PC client assume that you know the address of your Crazyflie (it is not feasible to try 2^{40} different addresses during scanning). If you forget the address, you will need to reset the EEPROM to its default values by connecting the Crazyflie directly to the PC using a USB cable and running a Python script.^a

^ahttps://wiki.bitcraze.io/doc:crazyflie:dev:starting#reset_eeprom.

5.2 Finding Good Communication Parameters

The radio can be tuned by changing two parameters: datarate and channel. The datarate can be 250 kBit/s, 1, or 2 MBit/s. A higher datarate has a lower chance of collision with other networks such as WiFi but less range. Hence, for indoor applications the highest datarate (2 MBit/s) is recommended.

The channel number defines the offset in MHz from the base frequency of 2400 MHz. For example, channel 15 sets the operating frequency to 2415 MHz and channel 80 refers to an operating frequency of 2480 MHz. If you selected 2 MBit/s as datarate, the channels need to have a spacing of at least 2 MHz (otherwise, a 1 MHz spacing is sufficient).

Unlike WiFi, there is no channel hopping implemented in the Crazyflie. That means that the selected channel is very important because it will not change over time. On the other hand, interference can change over time; for example, a WiFi router might switch channels at runtime. Therefore, it is best if, during your flights, you can disable any interfering signal such as WiFi or wireless mouse/keyboards which use the 2.4 GHz band. If that is not possible, you can use the following experiments to find a set of good channels:

- Use the Bitcraze PC Client to teleoperate the Crazyflie in the intended space. Look at the “Link Quality” indicator on the top right. This indicator shows the percentage of successfully delivered packets. If it is low, there is likely interference.
- If you teleoperate the Crazyflie using ROS, there will be a ROS warning if the link quality is below a certain threshold. Avoid those channels. Additionally, `rqt_plot` shows the Radio Signal Strength Indicator (RSSI). This value, measured in -dBm, indicates the signal strength, which is affected both by distance and interference. A low value (e.g., 35) is good, while a high value (>80) is bad. For example, the output in Fig. 6 suggests that another channel should be used, because the second half of the plot shows additional noise caused by interference.

Once you have found a set of good channels, you can assign them to your Crazyflies, using the Bitcraze PC Client (see Sect. 5.1 for details). You can share

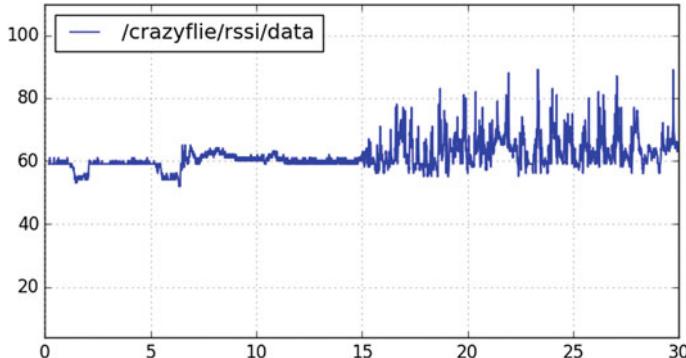


Fig. 6 Output of `rqt_plot` showing the radio signal strength indicator. The first 15 s show a good signal, while the second half shows higher values and noise caused by interference

up to four Crazyflies per Crazyradio with reasonable performance. Hence, the number of channels you need is about a quarter of the number of Crazyflies you intend to fly.

Legal Restrictions

In some countries the 2.4 GHz band is limited to certain channels. Please refer to your local regulations before you adjust the channel.

For example, in the United States frequencies between 2483.5 and 2500 MHz are power-restricted and as a result frequently not used by WiFi routers. Hence, channels 84 to 100 might be a good choice there. Channels above 2500 MHz are not allowed to be used in the United States.

5.3 ROS Usage (Multiple Crazyflies)

Let's assume that you have two Crazyflies with unique addresses, two joysticks, and a single Crazyradio. You can teleoperate them using the following command:

```
$ roslaunch crazyflie_demo multi_teleop_xbox360.launch
uri1:=radio:///0/100/2M/E7E7E7E701
uri2:=radio:///0/100/2M/E7E7E7E702
```

This should connect to both Crazyflies, visualize their state in `rivz`, and plot real-time data using `rqt_graph`. Furthermore, each joystick can be used to teleoperate one of the Crazyflies.

The launch file looks very similar to the single UAV case:

multi_teleop_xbox360.launch

```

1 <launch>
2   <arg name="uril" default="radio://0/90/2M" />
3   <arg name="uri2" default="radio://0/80/2M" />
4   <arg name="joy_dev1" default="/dev/input/js0" />
5   <arg name="joy_dev2" default="/dev/input/js1" />
6
7   <include file="$(find
8     crazyflie_driver)/launch/crazyflie_server.launch" />
9   <group ns="crazyfliel">
10    <!-- Similar to before -->
11   </group>
12   <group ns="crazyfli2">
13    <!-- Similar to before -->
14   </group>
15   <!-- Visualization (Similar to before) -->
</launch>
```

In particular, we still have a single `crazyflie_server` (which now manages both Crazyflies). However, we have two different namespaces (`crazyfliel` and `crazyfli2`). The content of those namespaces is nearly identical to the single UAV case (compare lines 5–16 in `teleop_xbox360.launch`, Sect. 4) and thus not repeated here for clarity.

In order to teleoperate more than two Crazyflies, you simply need to add more groups with different namespaces to the launch file. If you want the ROS driver to use a different Crazyradio, you can adjust the first number in the URI. For example, `radio://1/100/2M/E7E7E7E701` uses the second Crazyradio (or reports an error if only one is plugged in). It is important to consider the following for the usage of multiple radios:

- For improved performance, use the same channel per Crazyradio. This avoids that the radio changes channels whenever it switches between sending to different Crazyflies.
- If you do not need the IMU raw data, disable it by setting `enable_logging` to `False` when you include the `crazyflie_add.launch` file. This saves bandwidth and allows you to use more than two Crazyflies per radio.

Depending on your packet drop rate, you can use up to two Crazyflies per Crazyradio if logging is enabled and up to four otherwise. It does work with a higher number as well, but you will see decreasing controllability since the radio is used in a time-slice fashion.

6 Hovering

A first important step for autonomous flight of a quadcopter is hovering in place. This also requires the ability to take off from the ground and land after the flight. All of these basic motions require a position controller, which takes the Crazyflie's current position as input in order to compute new commands for the Crazyflie. Hence, this position controller is replacing the teleoperating human we had before.

This section describes the `crazyflie_controller` package and how it is used for autonomous take-off, landing, and hovering. As before, first the single UAV case is considered and later it is extended to the multi-UAV case. Furthermore, this section will cover working strategies on how to use the crazyflie with optical motion capture systems such as VICON⁹ and OptiTrack.¹⁰ This is, due to the size of the UAV, a non-trivial task, particularly for swarming applications.

6.1 Position Estimate

We assume that there is already a way to track the position and preferably yaw of the Crazyflies at least 30Hz. It is possible to use Microsoft Kinect,¹¹ AR tags, or Ultra-Wideband Localization [10] for this task. However, those solutions are not as accurate at specialized motion capture systems, which can reach sub-millimeter accuracy. We want to fly many small quadcopters, perhaps in a dense formation, and hence need a very accurate position feedback. Therefore, we will discuss the usage of optical motion capture systems such as VICON or OptiTrack.

We run our experiments in a space of approximately 5 m × 4 m equipped with a 12-camera VICON MX motion capture system. Optical motion capture systems typically require spatially unique marker configurations for each object to track such that it is possible to identify each object.¹² Otherwise, occlusions or a short-term camera outage would result in unrecoverable tracking failures. For a small platform like the Crazyflie, there are not many ways to place markers uniquely on the existing frame. In particular, if you need more than four Crazyflies, you will need to add additional structures where you can place the markers:

- Propeller guards. They are commercially available for the Hubsan X4 toy quadrotor,¹³ which has identical physical dimensions. Moreover, you can use a 3D printer to print your own guard based on published files on thingiverse.¹⁴

⁹<http://www.vicon.com/>.

¹⁰<https://www.optitrack.com/>.

¹¹<https://github.com/ataffanel/crazyflie-ros-kinect2-detector>.

¹²Some solutions, like the Crazyswarm project [5], use identical marker configurations.

¹³You can search on amazon for “propeller guard hubsan x4.”

¹⁴<http://www.thingiverse.com/search?q=crazyflie&sa=>.

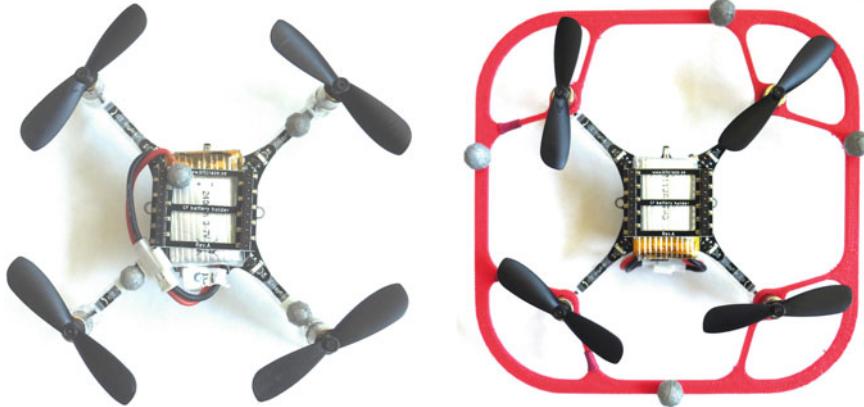


Fig. 7 Left Crazyflie with four optical markers (6.4 mm) attached and no additional guard used. Right Crazyflie with markers on propeller guard to allow a higher number of unique marker configurations

- Custom motor mounts. OpenSCAD¹⁵ files can be found in the official mechanical repository.¹⁶
- Spatial extensions in form of sticks, either mounted on the rotor arms or on top of the Crazyflie as extension board.

For small groups of up to four Crazyflies, we place the markers directly on the Crazyflies. We flew up to six Crazyflies (using three Crazyradios) using the propeller guard approach. However, this significantly reduces flight times and changes the flight dynamics. Figure 7 shows examples of Crazyflies equipped with markers.

The exact method is highly dependent on your motion capture system, so there will be some experimentation involved. Similarly, the best markers to use depend on the system as well. We successfully use 6.4 and 7.9 mm spherical traditional reflective markers from B&L Engineering.¹⁷ A smaller size impacts the flight dynamics less (and fits underneath the rotors) and is preferred as long as the motion capture system is able to detect the markers properly. We use the No-Base option of the markers and small pieces of Command Poster Strips¹⁸ to attach them to the Crazyflie.

If you use VICON, it is best to install the `vicon_bridge` ROS package using the following steps:

¹⁵<http://www.openscad.org/>.

¹⁶<https://github.com/bitcraze/bitcraze-mechanics/tree/master/cf2-mount-openscad>.

¹⁷<http://www.bleng.com/>.

¹⁸<http://www.command.com>.

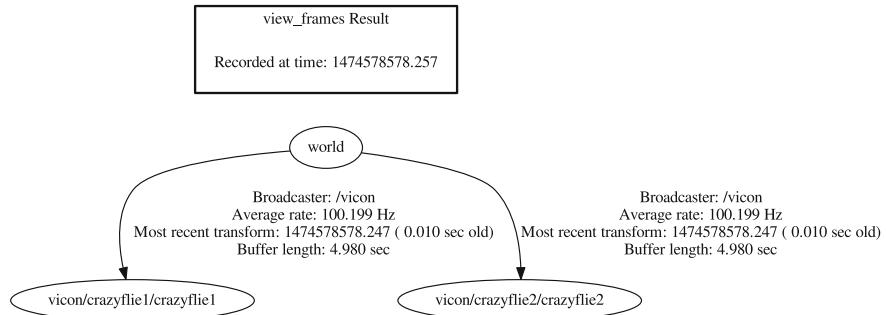


Fig. 8 Output of `view_frames` for two objects names `crazyflie1` and `crazyflie2`, respectively

```

$ cd ~/crazyflie_ws/src
$ git clone https://github.com/ethz-asl/vicon_bridge.git
$ cd ~/crazyflie_ws
$ catkin_make

```

This will add the source to your workspace and compile it. The package assumes that you have another PC with VICON Tracker running in the same network, accessible under the hostname `vicon` and with no firewalls in between. You can test your installation by running:

```
$ roslaunch vicon_bridge vicon.launch
```

In another terminal, execute:

```
$ rosrun tf view_frames
```

and open the resulting `frames.pdf` file to check your transformations. It should look like Fig. 8.

If you use OptiTrack (or any other motion capture system which supports VRPN¹⁹), you can install the `vrpn_client_ros` package using:

```
$ sudo apt-get install ros-indigo-vrpn-client-ros
```

In order to test it, you will need to write a custom launch file, similar to the sample file provided in the package.²⁰ Afterwards, you can check if it works using `view_frames`.

¹⁹<https://github.com/vrpn/vrpn/wiki>.

²⁰https://github.com/clearpathrobotics/vrpn_client_ros/blob/indigo-devel/launch/sample.launch.

Coordinate System

It is important to verify that your transformations match the ROS standard.^a That means we use a right-handed coordinate system with x forward, y left, and z pointing up. One way to check is to launch `rviz`, and add a “TF” visualization. Move the Crazyflie around in your hand, while verifying that the visualization in `rviz` matches the expected coordinate system.

^a<http://www.ros.org/reps/rep-0103.html>.

6.2 ROS Usage (Single Crazyflie)

Here, we assume that you have a working localization for a single Crazyflie already. We assume that there is a ROS transform between the frames `/world` and `/crazyflie1` and that `radio://0/100/2M/E7E7E7E701` is the URI of your Crazyflie.

With VICON you can launch the following:

```
$ rosrun crazyflie_demo hover_vicon.launch
    uri:=radio://0/100/2M/E7E7E7E701 frame:=crazyflie1 x:=0
    y:=0 z:=0.5
```

Once the Crazyflie is connected, you can press the blue (X) button on the XBox360 controller to take off and the green (A) button to land. If successful, the Crazyflie should hover at $(0, 0, 0.5)$. Use the red (B) button to handle any emergency situation (or unplug the Crazyradio to get the same effect). The launch file starts `rviz` as well, visualizing both the Crazyflie’s current position and goal position (indicated by a red arrow).

If you are using OptiTrack, you can use `hover_vrp.launch` rather than `hover_vicon.launch`.

The launch file is similar to before, but adds a few more elements:

`hover_vicon.launch`

```
1 <launch>
2   <!-- Launch file arguments -->
3   <include file="$(find
4     crazyflie_driver)/launch/crazyflie_server.launch" />
5   <group ns="crazyflie">
6     <!-- Similar to before -->
7     <node name="joystick_controller" pkg="crazyflie_demo"
          type="controller.py" output="screen">
```

```

7      <param name="use_crazyflie_controller" value="True"
8          />
9      </node>
10     <include file="$(find
11         crazyflie_controller)/launch/crazyflie2.launch">
12         <arg name="frame" value="$(arg frame)" />
13     </include>
14     <node name="pose" pkg="crazyflie_demo"
15         type="publish_pose.py" output="screen">
16         <param name="name" value="goal" />
17         <param name="rate" value="30" />
18         <param name="x" value="$(arg x)" />
19         <param name="y" value="$(arg y)" />
20         <param name="z" value="$(arg z)" />
21     </node>
22     <node pkg="tf" type="static_transform_publisher"
23         name="baselink_broadcaster" args="0 0 0 0 0 0 1
24             $(arg frame) /crazyflie/base_link 100" />
25 </group>
26 <!-- run vicon bridge or vrpn_client_ros -->
27     <param name="robot_description" command="$(find
28         xacro)/xacro.py $(find
29             crazyflie_description)/urdf/crazyflie.urdf.xacro" />
30     <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
31         crazyflie_demo)/launch/crazyflie_pos.rviz"
32         required="true" />
33 </launch>

```

We start by defining the arguments (not shown here for brevity) and launching the `crazyflie_server` (line 3). Within the group element, we include `crazyflie_add` (not shown). Now we get a few differences: in line 7 we set `use_crazyflie_controller` to True to enable the takeoff and landing behavior using the joystick. Moreover, we add a position controller node by including `crazyflie2.launch` (lines 9–11). The static goal position for this controller is published in the `/crazyflie/goal` topic in lines 12–18. The group ends by publishing a static transform from the given frame to the Crazyflie’s base link. This allows us to visualize the current pose of the Crazyflie in `rviz` using the 3D model provided in the `crazyflie_description` package (lines 19 and 22).

6.3 ROS Usage (Multiple Crazyflies)

The main difference between the single UAV and multi-UAV case is that the joystick should be shared: Takeoff, landing, and an emergency should trigger the appropriate behavior on all Crazyflies. This allows us to have a single backup pilot who can trigger emergency power-off in case of an issue. The low inertia of the Crazyflies causes them to be very robust to mechanical failures when dropping from the air.

We have had crashes from heights of up to 4m on a slightly padded floor, with only propellers and/or motor mounts needing replacement. (Replacement parts are available for purchase separately.)

The `crazyflie_demo` package contains an example for hovering two Crazyflies. You can run it for VICON by executing:

```
$ rosrun crazyflie_demo multi_hover_vicon.launch
uri1:=radio://0/100/2M/E7E7E7E701 frame1:=crazyflie1
uri2:=radio://0/100/2M/E7E7E7E702 frame2:=crazyflie2
```

There is also an example using VRPN (`multi_hover_vrpn.launch`). The launch file is similar to the single Crazyflie case:

multi_hover_vicon.launch

```

1 <launch>
2   <!-- Launch file arguments -->
3   <include file="$(find
4     crazyflie_driver)/launch/crazyflie_server.launch" />
5   <node name="joy" pkg="joy" type="joy_node"
6     output="screen">
7     <param name="dev" value="$(arg joy_dev)" />
8   </node>
9   <group ns="crazyflie1">
10    <!-- Similar to before -->
11    <node name="joystick_controller" pkg="crazyflie_demo"
12      type="controller.py" output="screen">
13      <param name="use_crazyflie_controller" value="True"
14        />
15      <param name="joy_topic" value="/joy" />
16    </node>
17  </group>
18  <group ns="crazyflie2">
19    <!-- Similar to first group -->
20  </group>
21  <!-- Similar to before -->
```

In this case, we only need a single joystick; the `joy` node for it is instantiated in lines 4–6. In order to use that topic, we need to supply `controller.py` with the correct topic name (line 11).

We can summarize what we have learned so far by looking at the output of `rqt_graph`, as shown in Fig. 9. It shows the various nodes (ellipsoid), namespaces (rectangles), and topics (arrows). In particular, we have two namespaces: `crazyflie1` and `crazyflie2`. Each namespace contains the nodes used for a single Crazyflie: `joystick_controller` to deal with the user-input, `pose` to publish the (static) goal position for that particular Crazyflie, and `controller` to compute the low-level attitude commands based on the high-level user input. The attitude commands are transmitted using the `cmd_vel` topics. There is only

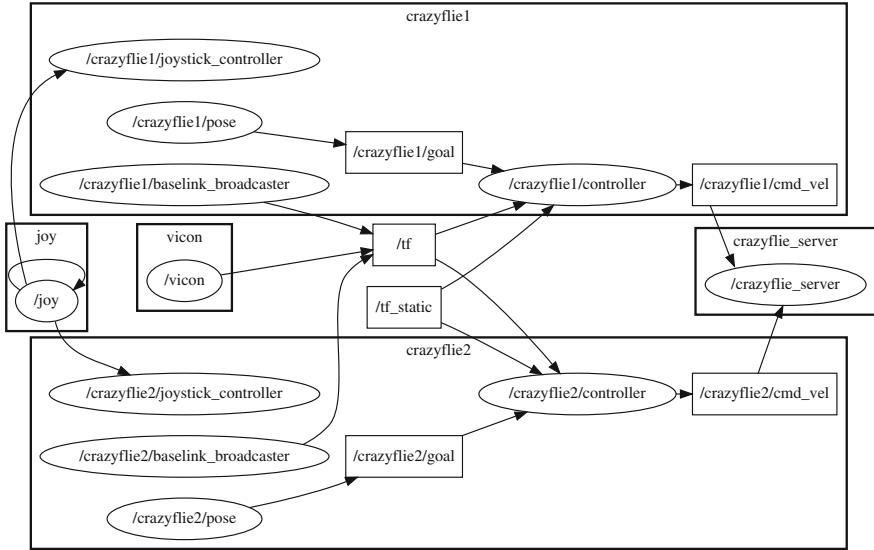


Fig. 9 Visualization of the different nodes and their communication using `rqt_graph`

one node, the `crazyflie_server`, which listens on those topics and transmits the data to both Crazyflies, using the Crazyradio. The `joy` node provides the input to both namespaces, allowing a single user to control both Crazyflies. Similarly, the `vicon` node is shared between Crazyflies, because the motion-capture system provides feedback (in terms of `tf` messages) of all quadcopters. The `baselink_broadcaster` nodes are only used for visualization purposes, allowing us to visualize a 3D model of the Crazyfly in `rviz`. More than two Crazyflies can be used by duplicating the groups in the launch file accordingly. This will result in more namespaces; however, the `crazyflie_server`, `vicon`, and `joy` nodes will always be shared between all Crazyflies.

7 Waypoint Following

The hovering of the previous section is extended to let the UAVs follow specified waypoints. This is useful if you want the robots to fly specified routes, for example for delivery systems or construction tasks. As before, first the single-UAV case is presented, followed by how to use it in the multi-UAV case.

Here, we concentrate on the ROS-specific changes in a toy example where the waypoints are static and known upfront. Planning such routes for a group of quadcopters is a difficult task in itself and we refer the reader to related publications [11–13].

The main difference between hovering and waypoint following is that, for the latter, the goal changes dynamically. First, test the behavior of the controller for dynamic waypoint changes:

```
$ roslaunch crazyflie_demo teleop_vicon.launch
```

Here, the joystick is used to change the goal pose rather than influencing the motor outputs directly. The visualization in rviz shows the current goal pose as well as the quadcopter pose to provide some feedback.

Waypoint following works in a similar fashion: the first waypoint is set as goal position and, once the Crazyflie reaches its current goal (within some radius), the goal point is set to the next position. This simple behavior is implemented in `crazyflie_demo/scripts/demo.py`. Each Crazyflie can have its own waypoint defined in a Python script, for example:

```
1  #!/usr/bin/env python
2  from demo import Demo
3
4  if __name__ == '__main__':
5      demo = Demo(
6          [
7              #x , y, z, yaw, sleep
8              [0.0 , 0.0, 0.5, 0, 2],
9              [1.5 , 0.0, 0.5, 0, 2],
10             [-1.5 , 0.0, 0.75, 0, 2],
11             [-1.5 , 0.5, 0.5, 0, 2],
12             [0.0 , 0.0, 0.5, 0, 0],
13         ]
14     )
15     demo.run()
```

Here, x , y , and z are in meters, yaw is in radians, and sleep is the delay in seconds before the goal switches to the next waypoint.

Adjust `demo1.py` and `demo2.py` to match your coordinate system and run the demo for two Crazyflies using:

```
$ roslaunch crazyflie_demo multi_waypoint_vicon.launch
```

The path for the two Crazyflies should not be overlapping because simple waypoint following does not have any time guarantees. Hence, it is possible that the first Crazyflie finishes much earlier than the second one, even if the total path length and sleep time are the same. This limitation can be overcome by generating a trajectory for each Crazyflie and setting the goal points dynamically accordingly.

The launch file looks very similar to before:

```
multi_waypoint_vicon.launch

1 <launch>
2   <!-- Launch file arguments -->
3   <include file="$(find
4     crazyflie_driver)/launch/crazyflie_server.launch" />
5   <node name="joy" pkg="joy" type="joy_node"
6     output="screen">
7     <param name="dev" value="$(arg joy_dev)" />
8   </node>
9   <group ns="crazyfliel">
10    <!-- Similar to before -->
11    <node name="pose" pkg="crazyflie_demo" type="demo1.py"
12      output="screen">
13      <param name="frame" value="$(arg frame1)" />
14    </node>
15  </group>
16  <group ns="crazyfliel2">
17    <!-- Similar to first group -->
18  </group>
19  <!-- Similar to before -->
20 </launch>
```

Instead of publishing a static pose, each Crazyflie now executes its own `demo<x>.py` node, which in turn publishes goals dynamically. An example video demonstrating six Crazyflies following dynamically changing goals is available online.²¹

8 Troubleshooting

As with most physical robots, debugging can be difficult. In order to identify and eventually solve the problem, it helps to simplify the failing case until it is easier to analyze. In this section, we provide several actions which have helped us resolve issues in the past. In particular, we first identify if the issue is on the hardware or software side, and provide recipes to address both kinds of issues.

1. Verify that the position estimate works correctly. For example, use `rviz` to visualize the current pose of all quadrotors, move a single quadrotor manually at a time and make sure that `rviz` reflects the changes accordingly.
2. Check the wireless connection between the PC and the Crazyflies. If the packet drop rate is high, the `crazyflie_server` will output ROS warnings. Similarly, you can check the LEDs on each Crazyradio; ideally the LEDs show mostly green. If there is a communication issue the LEDs will frequently flash red as well. If communication is an issue, try a different channel by following Sect. 5.2.

²¹<http://youtu.be/px9iHkA0nOI>.

3. Work your way backwards: If a swarm fails, test the individual Crazyflies (or subgroups of them). If waypoint following fails, test hovering and, if there is an issue there as well, test teleoperation using ROS followed by teleoperation using the Bitcraze PC Client.
4. Issues with many Crazyflies but not smaller subgroups can occur if there are communication issues or if the position estimate suddenly worsens. For the first case, try reducing the number of Crazyflies per Crazyradio and adjusting the channel. For the second case try to estimate the latency of your position estimator. If you have multiple objects enabled, there might be axis-flips (marker configurations might not be unique enough) or the computer doing the tracking might be adding too much latency for the controller to operate properly.
5. If waypoint following does not work, make sure that you visualize the current waypoint in `rviz`. In general, the waypoints should not jump around very much. The provided controller is a hover controller which works well if the goal point is within a reasonable range of the Crazyflie's current position.
6. If hovering does not work, you can try to tune the provided controller. For example, if you have a higher payload you might increase the proportional gains. You can find the gains in `crazyflie_controller/config/crazyflie2.yaml`.
7. If teleoperation does not work or it is very hard to keep the Crazyflie hovering in place, there is most likely an issue with your hardware. Make sure that the propellers are balanced²² and that the battery is placed in the center of mass. When in doubt, replace the propellers.

9 Inside the `crazyflie_ros` Stack

This section will cover some more details of the stack. The knowledge you gain will not only help you better understand what is happening under the hood, but also provide the foundations to change or add features. Furthermore, some of the design insights given might be helpful for similar projects.

We will start with a detailed explanation of the different packages that compose the stack and their relationship. For each package, we will discuss important components and the underlying architecture. For example, for the `crazyflie_driver` package we will explain the different ROS topics and services, why there is a server, and how the radio time-slicing works. Guidelines for possible extensions will conclude the section.

²²<https://www.bitcraze.io/balancing-propellers/>.

9.1 Overview

The `crazyflie_ros` stack is composed of six different packages:

crazyflie_cpp contains a C++11 implementation for the Crazyradio driver as well as the Crazyflie. It supports the logging framework streaming data in real-time and the parameter framework adjusting parameters such as PID gains. This package has no ROS dependency and only requires `libusb` and `boost`. Unlike the official Python SDK it supports multiple Crazyflies over a shared radio.

crazyflie_tools contains standalone command line tools which use the `crazyflie_cpp` library. Currently, there is a tool to find any Crazyflies in range and tools to list the available logging variables and parameters. Because there is no ROS dependency, the tools can be used without ROS as well.

crazyflie_description contains the URDF description of the Crazyflie to visualize in `rviz`. The models are currently not accurate enough to be used for simulation.

crazyflie_driver contains a ROS wrapper around `crazyflie_cpp`. The logging subsystem is mapped to ROS messages and parameters are mapped to ROS parameters. One node (`crazyflie_server`) manages all Crazyflies.

crazyflie_controller contains a PID position controller for the Crazyflie. As long as the position of the Crazyflie is known (e.g., by using a motion capture system or a camera), it can be used to hover or execute (non-aggressive) flight maneuvers.

crazyflie_demo contains sample scripts and launch files for teleoperation, hovering, and waypoint following for both single and multi Crazyflie cases.

The dependencies between the packages are shown in Fig. 10. Both `crazyflie_tools` and `crazyflie_demo` contain high-level examples. Because `crazyflie_cpp` does not have any ROS dependency, it can be used with other frameworks as well. We will now discuss the different packages in more detail.

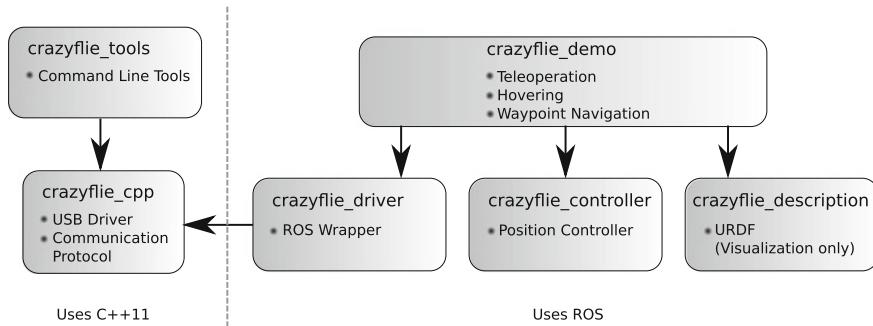


Fig. 10 Dependencies between the different packages within the `crazyflie_ros` stack

9.2 *crazyflie_cpp*

The `crazyflie_cpp` package is a static C++ library, with some components being header-only to maximize type-safety and efficiency. The library consists of four classes:

Crazyradio This class uses `libusb` to communicate with a Crazyradio. It supports the complete protocol²³ implemented in the Crazyradio firmware. The typical approach is to configure the radio (such as channel and datarate to use) first, followed by actual sending and receiving of data. The Crazyradio operates in Primary Transmitter Mode (PTX), while the Crazyflie operates in Primary Receiver Mode (PRX). That means that the Crazyradio is sending data (with up to 32 bytes of payload) using the radio and, if the data is successfully received, will receive an acknowledgement from the Crazyflie. The acknowledgment packet might contain up to 32 bytes of user-data as well. However, since the acknowledgment has to be sent immediately, the acknowledgment is not a direct response to the request sent. Instead, the communication can be seen as two asynchronous data streams, with one stream going from the Crazyradio to the Crazyflie and another stream for the reverse direction. If a request-respond like protocol is desired, it has to be implemented on top of the low-level communication infrastructure. The Crazyradio will automatically resend packets if no acknowledgment has been received.

Below is a small example on how to use the class to send a custom packet:

```

1 Crazyradio radio(0); // Instantiate an object bound to the
                      // first Crazyflie found
2 radio.setChannel(100); // Update the base frequency to 2500
                        // MHz
3 radio.setAddress(0xE7E7E7E701); // Set the address to send
                                // to
4 // Send a packet
5 uint8_t data[] = {0xCA, 0xFE};
6 Crazyradio::Ack ack;
7 radio.sendPacket(data, sizeof(data), ack);
8 if (ack.ack) {
9     // Parse ack.data and ack.size
10 }
```

Exceptions are thrown in cases of error, for example if no Crazyradio could be found or if the user does not have the permission to access the USB dongle.

Crazyflie This class implements the protocol of the Crazyflie²⁴ and provides high-level functions to send new setpoints and update parameters. In order to support multiple Crazyflies correctly, it instantiates the Crazyradio automatically. A global static array of Crazyradio instances and mutexes is used. Whenever the Crazyflie needs to send a packet, it first uses the mutex to lock its Crazyradio, followed

²³<https://wiki.bitcraze.io/projects:crazyradio:protocol>.

²⁴<https://wiki.bitcraze.io/projects:crazyflie:crtsp>.

by checking if the radio is configured properly. If not, the configuration (such as address) is updated and finally the packet is sent. The mutex ensures that multiple Crazyflies can be used in separate threads, even if they share a Crazyradio. The critical section of sending a packet causes the radio to multiplex the requests in time. Therefore, the bandwidth is split between all Crazyflies which share the same radio.

Below is a small example demonstrating how multiple Crazyflies can be used with the same radio:

```

1  Crazyflie cf1("radio://0/100/2M/E7E7E7E701"); // Instantiate
   first Crazyflie object
2  Crazyflie cf2("radio://0/100/2M/E7E7E7E702"); // Instantiate
   second Crazyflie object
3  // launch two threads and set new setpoint at 100 Hz
4  std::thread t1([&] {
5      while (true) {
6          cf1.sendSetpoint(0, 0, 0, 10000); // send roll, pitch,
   yaw, and thrust
7          std::this_thread::sleep_for(std::chrono::milliseconds(10));
8      }
9  });
10 std::thread t2([&] {
11     while (true) {
12         cf2.sendSetpoint(0, 0, 0, 20000); // send roll, pitch,
   yaw, and thrust
13         std::this_thread::sleep_for(std::chrono::milliseconds(10));
14     }
15 });
16 t1.join();
17 );

```

First, two `Crazyflie` objects are instantiated. Then two threads are launched using C++11 and lambda functions. Each thread sends an updated setpoint consisting of roll, pitch, yaw, and thrust to its `Crazyflie` at about 100Hz.

LogBlock<T> This set of templated classes is used to stream out sensor data from the `Crazyflie`. The logging framework on the `Crazyflie` allows to create so-called log blocks. Each log block is a struct with a maximum size of 28 bytes, freely arranged based on global variables available for logging in the `Crazyflie` firmware. The list of available variables and their types can be queried at runtime (`requestLogToc` method in the `Crazyflie` class).

This templated version provides maximum typesafety at the cost that you need to know at compile time which log blocks to request.

LogBlockGeneric This class is very similar to `LogBlock<T>` but also allows the user to dynamically create log blocks at runtime. The disadvantages of this approach are that it does not provide typesafety and that it is slightly slower at runtime.

9.3 *crazyflie_driver*

We first give a brief overview of the ROS interface, including services, subscribed topics, and published topics. In the second part we describe the usage and internal infrastructure in more detail.

Most of the services and topics are within the namespace of a particular Crazyflie, denoted with `<crazyflie>`. For example, if you have two Crazyflies, there will be namespaces `crazyflie1` and `crazyflie2`.

The driver supports the following services:

add_crazyflie Adds a Crazyflie with known URI to the `crazyflie_server` node. Typically, this is used with the helper application from `crazyflie_add` from a launch file.

Type: `crazyflie_ros/AddCrazyflie`

<crazyflie>/emergency Triggers an emergency state, in which no further messages to the Crazyflie are sent. The onboard firmware will stop all rotors if it did not receive a message for 500 ms, causing the Crazyflie to fall shortly after the emergency was requested.

Type: `std_srvs/Empty`

<crazyflie>/update_params Uploads updated values of the specified parameters to the Crazyflie. The parameters are stored locally on the ROS parameter server. This service first reads the current values and then uploads them to the Crazyflie.

Type: `crazyflie_ros/UpdataParams`

The driver subscribes the following topics:

<crazyflie>/cmd_vel Encodes the setpoint (attitude and thrust) of the Crazyflie. This can be used for teleoperation or automatic position control.

Type: `geometry_msgs/Twist`

The following topics are being published:

<crazyflie>/imu Samples the inertial measurement unit of the Crazyflie every 10 ms, including the data from the gyroscope and accelerometer. The orientation and covariance are not known and therefore not included in the messages.

Type: `sensor_msgs/Imu`

<crazyflie>/temperature Samples the temperature as reported by the barometer every 100 ms. This might not be the ambient temperature, as the Crazyflie tends to heat up during operation.

Type: `sensor_msgs/Temperature`

<crazyflie>/magnetic_field Samples the magnetic field as measured by the IMU every 100 ms. Currently, the onboard magnetometer is not calibrated in the firmware. Therefore, external calibration is required to use it for navigation.

Type: `sensor_msgs/MagneticField`

<crazyflie>/pressure Samples the air pressure as measured by the barometer every 100 ms in mbar.

Type: `std_msgs/Float32`

<crazyflie>/battery Samples the battery voltage every 100 ms in V.

Type: std_msgs/Float32

<crazyflie>/rsssi Samples the Radio Signal Strength Indicator (RSSI) of the onboard radio in -dBm.

Type: std_msgs/Float32

The `crazyflie_driver` consists of two ROS nodes: `crazyflie_server` and `crazyflie_add`. The first manages all Crazyflies in the system (using one thread for each), while the second one is just a helper node to be able to add Crazyflies from a launch file.

It is possible to launch multiple `crazyflie_server`'s, but these cannot share a Crazyradio. This is mainly a limitation of the operating system, which limits the ownership of a USB device to one process. In order to hide this implementation detail, each Crazyflie thread will operate in its own namespace. If you use `rostopic`, the topics of the first Crazyflie will be in the `crazyflie1` namespace (or whatever `tf_frame` you assigned to it), even though the code is actually executed within the `crazyflie_server` context. Each Crazyflie offers a topic `cmd_vel` which is used to send the current setpoint (roll, pitch, yaw, and thrust) and, if logging is enabled, topics such as `imu`, `battery`, and `rssi`. Furthermore, services are used to trigger the emergency mode and to re-upload specified parameters. The values of the parameters themselves are stored within the ROS parameter server. They are added dynamically once the Crazyflie is connected, because parameter names, types, and values are all dynamic and dependent on your firmware version. For that reason, it is currently not possible to use the `dynamic_reconfigure` package, because in this case the parameter names and types need to be known at compile time. Instead, a custom service call needs to be triggered containing a list of parameters to update once a user changed a parameter on the ROS parameter server. The following Python example can be used to turn the headlight on (if the LED expansion is installed):

```

1 import rospy
2 from crazyflie_driver.srv import UpdateParams
3 rospy.wait_for_service("/crazyflie1/update_params")
4 update_params =
5     rospy.ServiceProxy("/crazyflie1/update_params",
6                       UpdateParams)
5 rospy.set_param("/crazyflie1/ring/headlightEnable", 1)
6 update_params([ "ring/headlightEnable"])

```

After the service has become available, a service proxy is created and can be used to call the service whenever a parameter needs to be updated. Updating the parameter sets the parameter to a new value followed by a service call, which will trigger an upload to the Crazyflie.

Another important part of the driver is the logging system support. If logging is enabled, the Crazyflie will advertise a number of fixed topics. In order to receive custom logging values (or at custom frequencies), you will either need to change the source code or use custom log blocks. The latter has the disadvantage that it is not

typesafe (it just uses an array of floats as message type) and that it will be slightly slower at runtime. You can use custom log blocks as follows:

customLogBlocks.launch

```

1  <launch>
2    <arg name="uri" default="radio://0/80/2M" />
3    <include file="$(find
4      crazyflie_driver)/launch/crazyflie_server.launch" />
5    <group ns="crazyflie">
6      <node pkg="crazyflie_driver" type="crazyflie_add"
7        name="crazyflie_add" output="screen">
8        <param name="uri" value="$(arg uri)" />
9        <param name="tf_prefix" value="crazyflie" />
10       <rosparam>
11         genericLogTopics: ["log1", "log2"]
12         genericLogTopicFrequencies: [10, 100]
13         genericLogTopic_log1_Variables: ["pm.vbat"]
14         genericLogTopic_log2_Variables: ["acc.x", "acc.y",
15           "acc.z"]
16       </rosparam>
17     </node>
18   </group>
19 </launch>
```

Here, additional parameters are used within the `crazyflie_add` node to specify which log blocks to get. The first log block only contains `pm.vbat` and is sampled every 10 ms. A new topic named `/crazyflie1/log1` will be published. Similarly, the `/crazyflie1/log2` topic will contain three values (x , y , and z of the accelerometer), published every 100 ms.

The easiest way to find the names of variables is by using the Bitcraze PC Client. After connecting to a Crazyflie select “Logging Configurations” in the “Settings” menu. A new dialog will open and list all variables with their respective types. Each log block can only hold up 28 bytes and the minimum update period is 10 ms. You can also use the `listLogVariables` command line tool which is part of the `crazyflie_tools` package to obtain a list with their respective types.

9.4 *crazyflie_controller*

The Crazyflie is controlled by a cascaded PID controller. The inner attitude controller is part of the firmware. The inputs are the current attitude, as estimated using the IMU sensor data, and the setpoint (attitude and thrust), as received over the radio. The controller runs at 250 Hz.

The `crazyflie_controller` node runs another outer PID controller, which takes the current and goal position as input and produces a setpoint (attitude and thrust) for the inner controller. This cascaded design is typical if the sensor update

rates are different [11]. In this case, the IMU can be sampled much more frequently than the position.

A PID controller has absolute, integral, and differential terms on an error variable:

$$u(t) = K_P e(t) + K_I \int_0^t e(t) dt + K_D \frac{de(t)}{dt}, \quad (1)$$

where $u(t)$ is the control output and K_P , K_I and K_D are scalar parameters. The error $e(t)$ is defined as the difference between the goal and current value. The `crazyflie_controller` uses four independent PID controllers for x , y , z , and yaw, respectively. The controller also handles autonomous takeoff and landing. The integral part of the z -PID controller is initialized during takeoff with the estimated required base thrust to keep the Crazyflie hovering. The takeoff routine linearly increases the thrust, until the takeoff is detected by the external position system. A state machine switches to the PID controller, using the current thrust value as initial guess for the integral part of the z -axis PID controller. This avoids retuning of a manual offset in case the payload changes or a different battery is used.

The current goal can be changed by publishing to the `goal` topic. However, since the controller makes the hover assumption, large jumps between different control points should be avoided.

The various parameters can be tuned in a config file (`crazyflie_controller/config/crazyflie2.yaml`), or a custom config file can be loaded instead of the default one (see `crazyflie_controller/launch/crazyflie2.launch` for an example).

9.5 Possible Extensions

The overview of the `crazyflie_ros` stack should allow you to reuse some of its architecture ideas or to extend it further. For example, you can use the Crazyradio and `crazyflie_cpp` for any other remote-controlled robot which requires a low-latency radio link. The presented controller of the `crazyflie_controller` package is a simple hover controller. A non-linear controller, as presented in [14] or [11] might be an interesting extension to improve the controller performance. Higher-level behaviors, such as following a trajectory rather than just goal points, could make more interesting flight patterns possible. Finally, including simulation for the Crazyflie²⁵ could help research and development by enabling simulated experiments.

²⁵E.g., adding support to the RotorS package (http://wiki.ros.org/rotors_simulator).

10 Conclusion

In this chapter we showed how to use multiple small quadcopters with ROS in practice. We discussed our target platform, the Bitcraze Crazyflie 2.0, and guided the reader step-by-step to the process of letting multiple Crazyflies following waypoints. We tested our approach on up to six Crazyflies, using three radios. We hope that this detailed description will help other researchers use the platform to verify algorithms on physical robots.

More recent research has shown that the platform can even be used for swarms of up to 49 robots [5]. In the future, we would like to provide a similar step-by-step tutorial about the additional required steps to guide other researchers in working on larger swarms. Furthermore, it would be interesting to make the work more accessible to a broader audience once more inexpensive but accurate localization systems become available.

References

1. Michael, N., J. Fink, and V. Kumar. 2011. Cooperative manipulation and transportation with aerial robots. *Autonomous Robots* 30 (1): 73–86.
2. Augugliaro, F., S. Lupashin, M. Hamer, C. Male, M. Hehn, M.W. Mueller, J.S. Willmann, F. Gramazio, M. Kohler, and R. D’Andrea. 2014. The flight assembled architecture installation: Cooperative construction with flying machines. *IEEE Control Systems* 34 (4): 46–64.
3. Hönig, W., Milanes, C., Scaria, L., Phan, T., Bolas, M., and N. Ayanian. 2015. Mixed reality for robotics. In *IEEE/RSJ Intl Conference Intelligent Robots and Systems*, 5382–5387.
4. Mirjan, A., Augugliaro, F., D’Andrea, R., Gramazio, F., and M. Kohler. 2016. Building a Bridge with Flying Robots. In *Robotic Fabrication in Architecture, Art and Design 2016*. Cham: Springer International Publishing, 34–47.
5. Preiss, J.A., Hönig, W., Sukhatme, G.S., and N. Ayanian. 2016. Crazyswarm: A large nano-quadcopter swarm. In *IEEE/RSJ Intl Conference Intelligent Robots and Systems (Late Breaking Results)*.
6. Michael, N., D. Mellinger, Q. Lindsey, and V. Kumar. 2010. The GRASP multiple micro-uav testbed. *IEEE Robotics and Automation Magazine* 17 (3): 56–65.
7. Lupashin, S., Hehn, M., Mueller, M.W., Schoellig, A.P., Sherback, M., and R. D’Andrea. 2014. A platform for aerial robotics research and demonstration: The flying machine arena. *Mechatronics* 24(1):41–54.
8. Landry, B. 2015. Planning and control for quadrotor flight through cluttered environments, Master’s thesis, MIT.
9. Förster, J. 2015. System identification of the crazyflie 2.0 nano quadrocopter, Bachelor’s Thesis, ETH Zurich.
10. Ledigerger, A., Hamer, M., and R. D’Andrea. 2015. A robot self-localization system using one-way ultra-wideband communication. In *IEEE/RSJ Intl Conference Intelligent Robots and Systems*, 3131–3137.
11. Mellinger, D. 2012. Trajectory generation and control for quadrotors, Ph.D. dissertation, University of Pennsylvania.
12. Kushleyev, A., D. Mellinger, C. Powers, and V. Kumar. 2013. Towards a swarm of agile micro quadrotors. *Autonomous Robots* 35 (4): 287–300.
13. Hönig, W., Kumar, T.K.S., Ma, H., Koenig, S., and N. Ayanian. 2016. Formation change for robot groups in occluded environments. In *IEEE/RSJ Intl Conference Intelligent Robots and Systems*.

14. Lee, T., Leok, M., and N.H. McClamroch. 2010. Geometric tracking control of a quadrotor UAV on $\text{SE}(3)$. In *IEEE Conference on Decision and Control*, 5420–5425.

Author Biographies

Wolfgang Hönig has been a Ph.D. student at ACT Lab at University of Southern California since 2014. He holds a Diploma in Computer Science from Technical University Dresden, Germany. He is the author and maintainer of the `crazyflie_ros` stack.

Nora Ayanian is an Assistant Professor at University of Southern California. She is the director of the ACT Lab at USC and received her Ph.D. from the University of Pennsylvania in 2011. Her research focuses on creating end-to-end solutions for multirobot coordination.

Part II

Control of Mobile Robots

SkiROS—A Skill-Based Robot Control Platform on Top of ROS

**Francesco Rovida, Matthew Crosby, Dirk Holz,
Athanasios S. Polydoros, Bjarne Großmann,
Ronald P.A. Petrick and Volker Krüger**

Abstract The development of cognitive robots in ROS still lacks the support of some key components: a knowledge integration framework and a framework for autonomous mission execution. In this research chapter, we will discuss our skill-based platform SkiROS, that was developed on top of ROS in order to organize robot knowledge and its behavior. We will show how SkiROS offers the possibility to integrate different functionalities in form of skill ‘apps’ and how SkiROS offers services for integrating these skill-apps into a consistent workspace. Furthermore, we will show how these skill-apps can be automatically executed based on autonomous, goal-directed task planning. SkiROS helps the developers to program and port their high-level code over a heterogeneous range of robots, meanwhile the minimal Graphical User Interface (GUI) allows non-expert users to start and super-

This project has received funding from the European Union’s Seventh Framework Programme for research, technological development and demonstration under grant agreement no 610917 (STAMINA).

F. Rovida (✉) · A.S. Polydoros · B. Großmann · V. Krüger
Aalborg University Copenhagen, A.C. Meyers Vænge 15,
2450 Copenhagen, Denmark
e-mail: francesco@m-tech.aau.dk

A.S. Polydoros
e-mail: athanasios@m-tech.aau.dk

B. Großmann
e-mail: bjarne@m-tech.aau.dk

V. Krüger
e-mail: vok@m-tech.aau.dk

D. Holz
Bonn University, Bonn, Germany
e-mail: holz@ais.uni-bonn.de

M. Crosby · R.P.A. Petrick
Heriot-Watt University, Edinburgh, UK
e-mail: m.crosby@hw.ac.uk

R.P.A. Petrick
e-mail: r.petrick@hw.ac.uk

vise the execution. As an application example, we present how SkiROS was used to vertically integrate a robot into the manufacturing system of PSA Peugeot-Citroën. We will discuss the characteristics of the SkiROS architecture which makes it not limited to the automotive industry but flexible enough to be used in other application areas as well. SkiROS has been developed on Ubuntu 14.04 LTS and ROS indigo and it can be downloaded at <https://github.com/frovida/skiros>. A demonstration video is also available at <https://youtu.be/mo7UbwXW5W0>.

Keywords Autonomous robot · Planning · Skills · Software engineering · Knowledge integration · Kitting task

1 Introduction

In robotics the ever increasing level of system complexity and autonomy is naturally demanding a more powerful system architecture to relieve developers from reoccurring integration issues and to increase the robot's reasoning capabilities. Nowadays, several middleware-based component platforms, such as ROS, are available to support the composition of different control structures. Nevertheless, these middlewares are not sufficient, by themselves, to support the software organization of a full-featured autonomous robot (Fig. 1).

First, the presence of a knowledge-integration framework is necessary to support logic programming and increase software composability and reusability. In traditional robotic systems, knowledge is usually hidden or implicitly described in terms of if-then statements. With logic programming, the knowledge is integrated in a shared semantic database and the programming is based on queries over the database. This facilitates further software compositions since the robot's control program does not need to be changed, and the extended knowledge will automatically introduce more solutions. Also, reusability is improved because knowledge that has been described once, can now be used multiple times for recognizing objects, inferring facts, or parametrizing actions.



Fig. 1 SkiROS and the kitting pipeline ported on 3 heterogeneous industrial mobile manipulators

Second, the complex design process and integration of different robot's behaviors requires the support of a well-defined framework. The framework is not only necessary to simplify the software integration, but it is also fundamental to extend scripted behaviors with autonomous task planning based on context awareness. In fact, task planning in robotics is still not largely used due to the complexity of defining a planning domain and keeping it constantly updated with the robot's available capabilities and sensors readings.

In the course of a larger project on kitting using mobile manipulators [1–3], we have developed a particularly efficient pipeline for automated grasping of parts from pallets and bins, and a pipeline for placing into kits. To integrate these pipelines, together and with other ones, into different robot platforms, the Skill-based platform for ROS (*SkiROS*) was developed. The proposal for implementing such a programming platform defines *tasks* as sequences of *skills*, where skills are identified as the re-occurring actions that are needed to execute standard operating procedures in a factory (e.g., operations like pick ‘object’ or place at ‘location’). Embedded within the skill definitions are the sensing and motor operations, or *primitives*, that accomplish the goals of the skill, as well as a set of condition checks that are made before and after execution, to ensure robustness. This methodology provides a process to sequence the skills automatically using a modular task planner based on a standard domain description, namely the Planning Domain Definition Language (PDDL) [4]. The planning domain is automatically inferred from the robot's available skill set and therefore does not require to be stated explicitly from a domain expert.

In this research chapter we present a complete in-depth description of the platform, how it is implemented in ROS, and how it can be used to implement perception and manipulation pipelines on the example of mobile robot depalletizing, bin picking and placing in kits. The chapter is structured as follows. Section 2, discusses related work in general with a focus on the existing ROS applications. Section 3, discusses the software architecture theoretical background. Section 4, holds a tutorial on the graphical user interface. Section 5, holds a tutorial on the plug-ins development. Section 6, discusses the task planner theoretical background and tutorial on planner plug-in development. Section 7, presents an application on a real industrial kitting task. Section 8, discusses relevant conclusions.

1.1 Environment Configuration

SkiROS consist of a core packages set that can be extended during the development process with plug-ins. The SkiROS package, and some initial plug-ins sets can be downloaded from, respectively:

- <https://github.com/frovida/skiros>, core package
- https://github.com/frovida/skiros_std_lib, extension with task planner, drive-pick-place skills and spatial reasoner

- https://github.com/frovida/skiros_simple_uav, extension with drive-pick-place for UAVs, plus takeoff and landing skills

SkiROS has been developed and tested on ubuntu 14.04 with ROS indigo and the compilation is not guaranteed to work within a different setup.

Dependencies Skiros requires the oracle database and the redland library installed on the system. These are necessary for the world model activity. To install all dependencies is possible to use the script included in the SkiROS repository `skiros/scripts/install_dependencies.sh`. Other dependencies necessary for the planner can be installed running the script `skiros_std_lib/scripts/install_dependencies.sh`. After these steps, SkiROS can be compiled with the standard “`catkin_make`” command. For a guide on how to launch the system after compilation, refer to Sect. 3.1.

2 Related Work

During the last three decades, three main approaches to robot control have dominated the research community: reactive, deliberative, and hybrid control [5]. Reactive systems rely on a set of concurrently running modules, called behaviours, which directly connect input sensors to particular output actuators [6, 7]. In contrast, deliberative systems employ a sense-plan-act paradigm, where reasoning plays a key role in an explicit planning process. Deliberative systems can work with longer timeframes and goal-directed behaviour, while reactive systems respond to more immediate changes in the world. Hybrid systems attempt to exploit the best of both worlds, through mixed architectures with a deliberative high level, a reactive low level, and a synchronisation mechanism in the middle that mediates between the two [8]. Most modern autonomous robots follow a hybrid approach [9–12], with researchers focused on finding appropriate interfaces between the declarative descriptions needed for high-level reasoning and the procedural ones needed for low-level control.

In the ROS ecosystem, we find *ROSco*,¹ Smach² and *pi_trees*³ which are architectures for rapidly creating complex robot behaviors, under the form of Hierarchical Finite State Machine (HFSM) or Behavior Trees (BT). These softwares are useful to model small concatenations of primitives with a fair reactive behavior. The approach can be used successfully up to a level comparable to our skills’ executive, but doesn’t scale up for high dynamic contexts. In fact the architectures allow only static composition of behaviors and cannot adapt those to new situations during execution. At time being, and at the best of author knowledge, we find in the ROS ecosystem only one maintained package for automated planning: *rosplan*⁴ (Trex is no longer main-

¹<http://pwp.gatech.edu/hrl/ros-commander-rosco-behavior-creation-for-home-robots/>.

²<http://wiki.ros.org/smach>.

³http://wiki.ros.org/pi_trees.

⁴<https://github.com/KCL-Planning/ROSPlan>.

tained). In rosplan, the planning domain has to be defined manually from a domain expert. With our approach, the planning domain is automatically inferred at run-time from the available skill set, which results in a higher flexibility and usability.

Knowledge representation plays a fundamental role in cognitive robotic systems [13], especially with respect to defining world models. The most relevant approach for our work is the cognitivist approach, which highlights the importance of symbolic computation: symbolic representations are produced by a human designer and formalised in an ontology. Several modern approaches for real use-cases rely on semantic databases [14–16] for logic programming. It allows robotic system to remain flexible at run-time and easy to re-program. A prominent example of knowledge processing in ROS is the KnowRob system [17], which combines knowledge representation and reasoning methods for acquiring and grounding knowledge in physical systems. KnowRob uses a semantic library which facilitates loading and accessing ontologies represented in the Web Ontology Language (OWL). Despite its advanced technology, KnowRob is a framework with a bulky knowledge base and a strict dependency with the ‘Prolog’ language. For our project, a simpler and minimal implementation has been preferred, still compliant with the widely used OWL standard. Coupled with KnowRob there is CRAM (Cognitive Robot Abstract Machine) [18]. Like SkiROS, CRAM is a software toolbox for the design, the implementation, and the deployment of cognition-enabled autonomous robots, that do not require the whole planning domain to be stated explicitly. The CRAM kernel consists of the CPL plan language, based on Lisp, and the KnowRob knowledge processing system. SkiROS presents a similar theoretical background with CRAM, but differs in several implementations choices. For example, SkiROS doesn’t provide a domain specific language such as CPL to support low-level behavior design, but relies on straight C++ code and the planner in CRAM is proprietary, meanwhile in SkiROS is modular and compatible with every PDDL planner.

3 Conceptual Overview

The Skill-based platform for ROS (*SkiROS*) [19] helps to design and execute the high-level skills of an hybrid behavior-deliberative architecture, commonly referred with the name of *3-tiered architecture* [9–12]. As such, it manages the executive and deliberative layers, that are expected to control a behavior layer implemented using ROS nodes. While the theory regarding 3-tiered architectures is well known, it is still an open question how to build a general and scalable platform with well-defined interfaces. In this sense, the development of the SkiROS platform has been carried out taking into consideration the needs of two key stakeholders: the developer and the end-user. This approach derives from the field of the interaction design, where the human’s needs are placed as focal point of the research process. It is also included in the ISO standard [ISO9241-210;ISO16982].

Briefly, SkiROS provide: (i) a workspace to support the development process and software integration between different sources and (ii) an intuitive interface to instruct

missions to the robot. The main idea is that the developers can equip the robots with *skills*, defined as the fundamental software building blocks operating a modification on the world state. The end-user can configure the robot by defining a scene and a goal and, given this information, the robot is able to plan and execute autonomously the sequence of skills necessary to reach the required goal state. The possibility of specifying complex states is tightly coupled with the amount of skills that the robot is equipped with and the richness of the robot's world representation. Nevertheless, developing and maintaining a large skill set and a rich knowledge base can be an overwhelming task, even for big research groups. Modular and shareable skills are mandatory to take advantage of the *network effect* - a phenomenon occurring when the number of developers of the platform grows. When developers start to share skills and knowledge bases, there is possible to develop a robot able to understand and reach highly articulated goals. This is particularly achievable for the industrial case, where the skills necessary to fulfill most of use-cases have been identified from different researchers as a very compact set [20, 21].

The ROS software development approach is great to develop a large variety of different control systems, but lacks support to the reuse of effective solutions to recurrent architectural design problems. Consequently, we opted for a software development based on App-like plug-ins, that limits the developer to program a specific set of functionalities, specifically: primitives, skills, conditions, task planners and discrete reasoners. This approach partially locks the architecture of the system, but ensure a straightforward re-usability of all the modules. On the other side, we also modularized the core parts of the system into ROS nodes, so that the platform itself doesn't become a black box w.r.t. to ROS and can be re-used in some of its parts, like e.g. the world model.

Several iterative processes of trial and refinement has been necessary in order to identify:

- how to structure the system
- the part of the system that needs to be easily editable or replaced by the developer
- the interface required from the user in order to control and monitor the system, without the necessity of becoming an expert on all its parts

The application on a real use-case has been fundamental to apply these iterations.

3.1 Packages Structure

SkiROS is a collection of ROS packages that implements a layered architecture. Each layer is a stand-alone package, which shares few dependencies with other layers. The packages in the SkiROS core repository are:

- **skiros** - the skiros meta-package contains ROS launch files, logs, ontologies, saved instances and scripts to install system dependencies

- **skiros_resource, skiros_primitive** - these packages are still highly experimental and are not taken into consideration in this paper. The primitives are currently managed together with skills, in the skill layer.
- **skiros_skill** - contains the skill manager node and the base class for the skills and the primitives
- **skiros_world_model** - contain the world model node, C++ interfaces to the ROS services, utilities to treat ontologies and the base class for conditions and reasoners
- **skiros_common** - shared utilities
- **skiros_msgs** - shared ROS actions, services and messages
- **skiros_config** - contains definition of URIs, ROS topic names and other reconfigurable parameters
- **skiros_task** - the higher SkiROS layer, contain the task manager node and the base class for task planner plug-in
- **skiros_rqt_gui** - the Graphical User Interface, a plug-in for the ROS rqt package

Each layer implements core functionalities with plug-ins, using the ROS ‘plugin-lib’ system. The plug-ins are the basic building blocks available to the developer to design the robot behavior and tailor the system to his specific needs. This methodology ensure a complete inter-independence of the modules at compile time. Every node has clear ROS interfaces with others so that, if necessary, any implementation can be replaced. The system is also based on two standards: the Web Ontology Language (OWL) standard [22] for the knowledge base and the Planning Domain Definition Language (PDDL) standard [4] for the planner. The platform architecture is visualized in Fig. 2. The complete platform consist of three ROS nodes - task manager, skill manager and world model - plus a Graphical User Interface (GUI). It can be executed using the command:

```
roslaunch skiros skiros_system.launch robot_name:=my_robot
```

Where my_robot should be replaced with the desired semantic robot description in the knowledge base (see Sect. 5.1). The default robot model loaded is aau_stamina_robot. In the `skiros_std_lib` repository there is an example of the STAMINA use-case specific launch file:

```
roslaunch skiros_std_lib skiros_system_fake_skills.launch
```

This launch file runs the SkiROS system with two skill managers: one for the mobile base, loading the drive skill, and one for the arm, loading pick and place skills.

3.2 World Model

Generally speaking, it is possible to subdivide the robot knowledge into three main domains: continuous, discrete and semantic. Continuous data is extracted directly

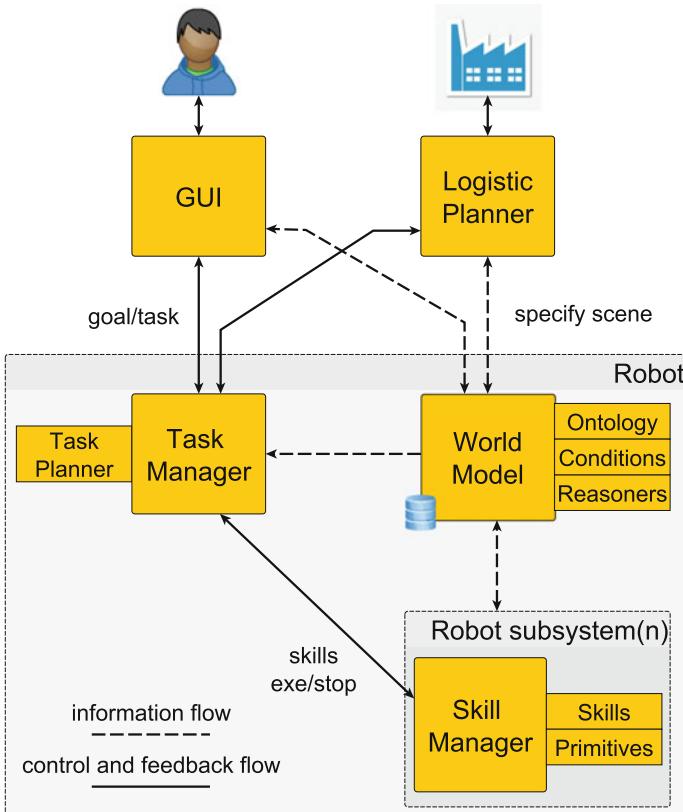


Fig. 2 An overview of the SkiROS architecture, with squares representing ROS nodes and rectangles representing plug-ins. The robot presents an external interface to specify the scene and receive a goal, that can be accessed by the GUI or directly from a factory system. Internally, the task manager dispatches the generated plans to the skill managers in each subsystem of the robot. A skill manager is the coordinator of a subset of capabilities, keeping the world model updated with its available hardware, skills and primitives. The world model is the focal point of the system: all knowledge is collected and shared through it

from sensors. Discrete data are relevant features that are computed from the continuous data and are sufficient to describe a certain aspect of the environment. Semantic data is abstract data, that qualitatively describes a certain aspect of the environment. Our world model stores semantic data. It works as a knowledge integration framework and supports the other subsystems' logic reasoning by providing knowledge on any relevant topic. In particular, the robot's knowledge is organised into an ontology that can be easily embedded, edited and extracted from the system. It is defined in the Web Ontology Language (OWL) standard which ensures greater portability and maintainability. The OWL ontology files have usually a .owl extension, and are based on XML syntax. An ontology consists of a set of definitions of basic categories

(objects, relations, properties) which describe the elements of the domain of interest, their properties, and the relations they maintain with each other [23]. Ontologies are defined in Description Logic (DL), a specialisation of first-order logic, which is designed to simplify the description of definitions and properties of categories. The knowledge base consists of a terminological component (T-Box), that contains the description of the relevant concepts in the domain and their relations, and an assertional component (A-Box) that stores concept instances (and assertions about those instances).

The SkiROS core ontology `skiros/owl/stamina.owl` gives a structure to organize the knowledge of 3 fundamental categories:

- the objects in the world
- the robot hardware
- the robot available capabilities (skills and primitives)

The knowledge base can be extended from the developer at will. It is possible to modify the default OWL loading path `skiros/owl`, by specifying the parameter `skiros/owl_workspace`. All the OWL files found in the specified path are automatically loaded from the world model at boot and merged to the SkiROS knowledge core - that is always loaded first. The world model node can be executed individually with the command:

```
rosrun skiros_world_model world_model_node
```

At run-time, the world model allows all the modules to maintain a shared working memory in a world instance, or scene, which forms a database complementary to the ontology database. The scenes are managed in the path specified in the `skiros/scene_workspace` parameter (default: `skiros/scene`). It is possible to start the world model with a predefined scene, by specifying the `skiros/scene_name` parameter. It is also possible to load and save the scene using the ROS service or the SkiROS GUI. An example of the scene tree structure is showed

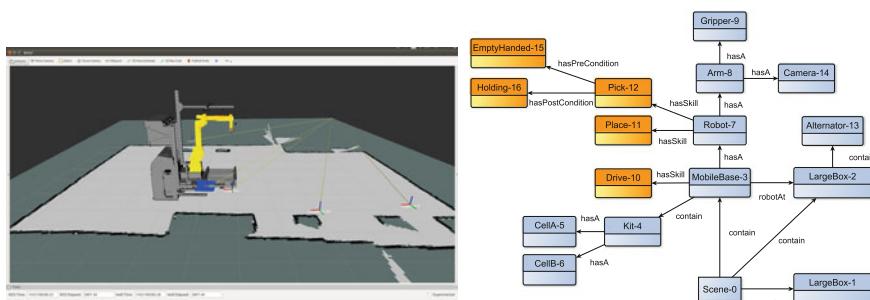


Fig. 3 An example of a possible scene, with the robot visualized on rviz (*left*) and the corresponding semantic representation (*right*). The scene includes both physical objects (*blue boxes*) and abstract objects (*orange boxes*)

in Fig. 3. The ontology can be extended automatically by the modules, to learn new concepts in a long-term memory (e.g. to learn a new grasping pose). The modules can modify the A-Box but not the T-Box. It is possible to interface with the world model using the following ROS services and topics:

- `/skiros_wm/lock_unlock` shared mutex for exclusive access (service)
- `/skiros_wm/query_ontology` query the world model with SPARQL syntax (service)
- `/skiros_wm/modify_ontology` add and remove statements in the ontology. New statements are saved in the file `learned_concepts.owl` in the owl workspace path. The imported ontologies are never modified (service)
- `/skiros_wm/element_get` get one or more elements from the scene (service)
- `/skiros_wm/element_modify` modify one or more elements in the scene (service)
- `/skiros_wm/set_relation` set relations in the scene (service)
- `/skiros_wm/query_model` query relations in the scene (service)
- `/skiros_wm/scene_load_and_save` save or load a scene from file (service)
- `/skiros_wm/monitor` publish any change done to the world model, both ontology and scene (topic)

It is also available a C++ interface class `skiros_world_model/world_model_interface.h` that wraps the ROS interface and can be included in every C++ program. This interface is natively available for all the skills and primitives plug-ins (see Sect. 5.2).

3.3 Skill Manager

The skill manager is a ROS node that collects the capabilities of a specific robot's subsystem. A skill manager is launched with the command:

```
rosrun skiros_skill skill_manager_node __name:=my_robot
```

Where `my_robot` has to be replaced with the identifier of the robot in the world model ontology. Since many of the skill managers' operations are based on the information stored in the world model, it requires the world model node to be running. Each skill manager in the system is responsible to instantiate in world scene its subsystem information: hardware, available primitives and available skills. Similarly, each primitive and skill can extend the scene information with the results of robot operation or sensing. To see how to create a new robot definition refer to Sect. 5.1. A skill manager, by default, tries to load all the skills and primitives that are been defined in the pluginlib system.⁵ It is also possible to load only a specific set by defining the parameters: `skill_list` and `module_list`. For example:

```
<node name="my_robot" pkg="skiros_skill" type="skill_manager_node">
```

⁵<http://wiki.ros.org/pluginlib>.

```
<param name="skill_list" type="string" value="pick place"/>
<param name="module_list" type="string" value="arm_motion locate"/>
</node>
```

In this case the robot `my_robot` will try to load pick and place skill, and the `arm_motion` and `locate` primitives. If the modules are loaded correctly, they will appear on the world model, associated to the robot name. It is possible to interface with the skill manager using the following ROS services and topics:

- `/my_robot/module_command` command execution or stop of a primitive (service)
- `/my_robot/module_list_query` get the primitive list (service)
- `/my_robot/skill_command` command execution or stop of a skill (service)
- `/my_robot/skill_list_query` get the skill list (service)
- `/my_robot/monitor` publish execution feedback (topic)

It is also available a C++ interface class `skiros_skill/skill_manager_interface.h` that wraps the ROS interface and can be included in every C++ program and an high-level interface class `skiros_skill/skill_layer_interface.h` to handle multiple skill managers. Note that, on every skill manager, the same module can be executed once at a time, but different modules can be executed concurrently.

3.4 Task Manager

The task manager acts as the general robot coordinator. It monitors the presence of robot's subsystems via the world model and use this information to connect to the associated skill manager. The task manager is the interface for external systems, designed to be controlled by a GUI or the manufacturing execution system (MES) of a factory. The task manager is launched individually with the command:

```
rosrun skiros_task task_manager_node
```

It is possible to interface with the task manager using the following ROS services and topics:

- `/skiros_task_manager/task_modify` add or remove a skill from the list (service)
- `/skiros_task_manager/task_plan` send a goal to plan a skill sequence (service)
- `/skiros_task_manager/task_query` get the skill sequence (service)
- `/skiros_task_manager/task_exe` start or stop a task execution (topic)
- `/skiros_task_manager/monitor` publish execution feedback (topic)

3.5 Plugins

The plug-ins are C++ classes, derived from an abstract base class. Several plug-ins can derive from the same abstract class. For example, any skill derives from the abstract class skill base. The following system parts have been identified as modules:

- **skill** - an action with pre- and postconditions that can be concatenated to form a complete task
- **primitive** - a simple action without pre- and postconditions, that is concatenated manually from an expert programmer inside a skill. The primitives support hierarchical composition
- **condition** - a desired world state. It is expressed as a boolean variable (true/false) applied on a property of an element (property condition) or a relation between two elements (relation condition). The plug-in can wrap methods to evaluate the condition using sensors.
- **discrete reasoner** - an helper class necessary to link the semantic object definition to discrete data necessary for the robot operation
- **task planner** - a plug-in to plan the sequence of skills given a goal state. Any planner compatible with PDDL and satisfying the requirements described in Sect. 6 can be used

These software pieces are developed by programmers during the development phase and are inserted as plug-in into the system.

3.6 Multiple Robots Control

SkiROS can be used in multi-robot system in two ways. In the first solution, each skill manager is used to represent a robot in itself, and the task manager is used to plan and dispatch plans to each one of them. The solution is simple to implement and the robots will have a straightforward way to share the information via the single shared world model. The main limitation is that the skill execution is at the moment strictly sequential. Therefore, the task manager will move the robots one at a time. The second solution consist in implementing a high-level mission planner, and use this to dispatch goals for the SkiROS system running on each one of the robots. The latter solution is the one currently used for the integration in the PSA factory system [24].

4 User Interface

To allow any kind of user to be able to run and monitor the execution of the autonomous robot, the support of a clean and easy-to-use UI is necessary. At the

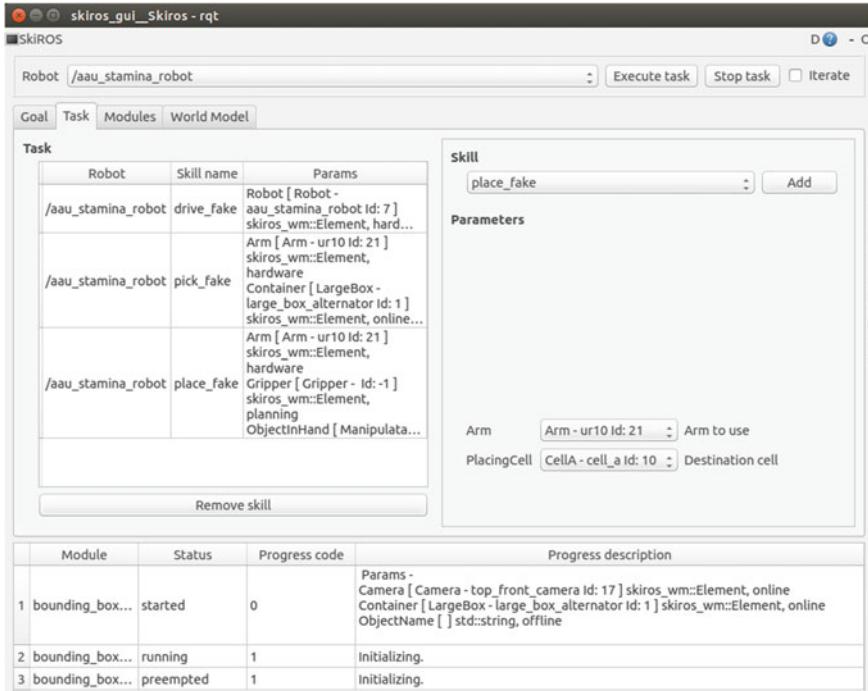


Fig. 4 The full GUI showing the task tab

moment, the interaction between human and robot is based on a Graphical UI, which in the future can be extended with more advanced and intuitive ways of interaction, like voice or motion capture. The full GUI is presented in Fig. 4. It consists of 4 tabs:

- **Goal** - from this tab is possible to specify the desired goal state and trigger an action planning
- **Task** - this tab visualize the planned skill sequence and allows to edit it
- **Module** - this tab allows to run modules (skills and primitives). It is principally used for testing purposes
- **World model** - from this tab is possible to load, edit and save the world scene

The GUI is structured for different level of user skill. The most basic user is going to use the Goal tab and the World model tab. First of all, he can build up a scene, then can specify the goals, plan a task and run or stop the execution. More advanced user can edit the planned task or build it by themselves from the Task tab. System tester can use the Module tab for module testing.

4.1 Edit, Execute and Monitor the Task

From the task tab presented in Fig. 4 is possible to edit a planned task or create a new one from scratch. The menu on the left allows to add a skill. First, the right robot must be selected from the top bar (e.g. /aau_stamina_robot). After this, a skill can be selected from the menu (e.g. place_fake). The skill must be parametrized appropriately and then can be added to the task list clicking the ‘Add’ button. The user can select each skill on the task list and remove it with the ‘Remove skill’ button. On the top bar there are two buttons to execute and stop the task execution and the ‘Iterate’ check box, that can be selected to repeat the task execution in loop (useful for testing a particular sequence). At the bottom is visualized the execution output of all the modules, with the fields:

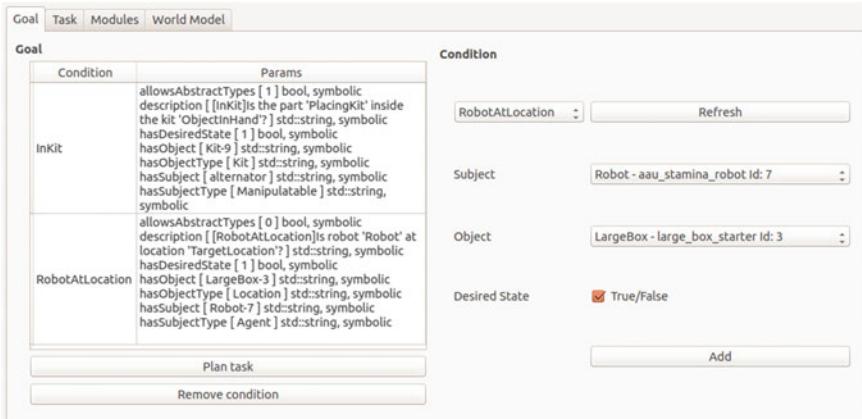
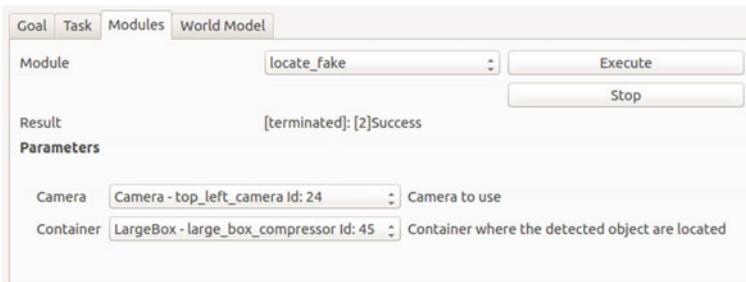
- **Module** - the module name
- **Status** - this can be: *started*, *running*, *preempted*, *terminated* or *error*
- **Progress code** - a positive number related to the progress of the module execution. A negative number indicates an error
- **Progress description** - a string describing the progress

4.2 Plan a Task

From the goal tab it is possible to specify the desired goal state and generate automatically a skill sequence, that will be then available in the task tab. The goal is expressed as a set of conditions required to be fulfilled. These can be chosen between the set of available ones. The available conditions set is calculated at run-time depending on the robots’ skill set and can be updated using the ‘Refresh’ button. It is possible to specify as goals only conditions that the robot can fulfill with its skills, or in other words, conditions that appear in at least one of the skills. In the example in Fig. 5, we require an abstract alternator to be in Kit-9 and we require the robot to be at LargeBox-3. By *abstract* we refer to individuals that are defined in the ontology, but not instantiated in the scene. Specify an abstract object means specify any object which match the generic description. The `InKit` condition allows abstract types, meanwhile `RobotAtLocation` can be applied only on instantiated objects (objects in the scene). For more details about conditions the reader can refer to Sect. 5.3.

4.3 Module Testing

From the Modules tab is possible to execute primitives. The procedure is exactly the same presented previously for the skills, except that the primitives are executed singularly. The module tab becomes handy to test the modules singularly or to setup

**Fig. 5** The goal tab**Fig. 6** The Modules tab

the robot, e.g. to teach a new grasping pose or to move the arm back to home position (Fig. 6).

4.4 Edit the Scene

From the world model tab Fig. 7 is possible to visualize and edit the world scene. On the left the scene is visualized in a tree structure. The limit of the tree structure doesn't allow to visualize the whole semantic graph, which can count several relations between the objects. We opted to limit the visualization to a scene graph, that is a general data structure commonly used in modern computer games to arrange the logical and spatial representation of a graphical scene. Therefore, only the spatial relations (contain and hasA) are visible, starting from the scene root node. On the right there are buttons to add, modify and remove objects in the scene. When an element in the tree is selected, its properties are displayed in the box on the right.

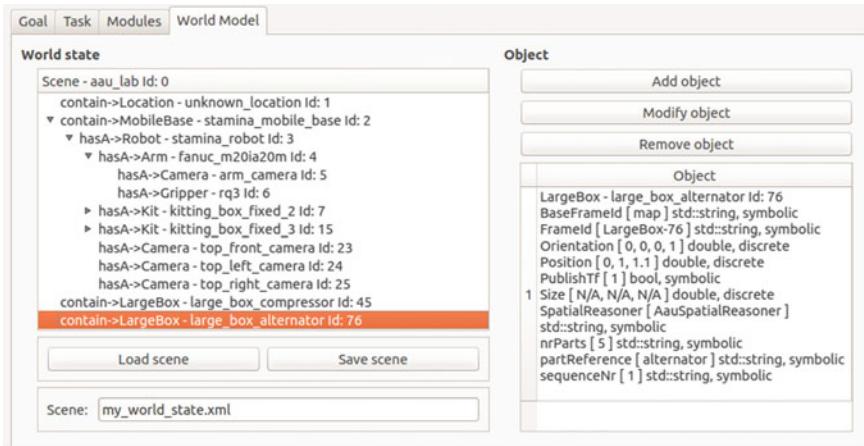


Fig. 7 The World Model tab

In the figure, for example, we are displaying the properties of the LargeBox 76. It is possible to edit its property by clicking on the ‘Modify object’ button. This opens a pop-up window where properties can be changed one by one, removed (by leaving the field blank) or added with the ‘+’ button on bottom. It is also possible to add an object by clicking on the ‘Add object’ button. When an object is added to the scene, it becomes child of the selected element in the tree. The properties and objects are limited to the set specified in the ontology. This not only helps to avoid input mistakes, but also to give the user an intuitive feedback on what it possible to put in the scene. Once the scene is defined, the interface on the bottom left allows to save the scene for future use.

5 Development

In this section we discuss how to develop an action context for the robot planning, using as an example a simplified version of the kitting planning context. The development process consist of two steps: specify the domain knowledge in the ontology and develop the plug-ins specified in Sect. 3.5. Some plug-ins and templates are available together with the SkiROS core package (see Sect. 1.1).

5.1 Edit the Ontology

Before starting to program, an ontology must be defined to describe the objects in the domain of interest. This regards in particular:

- **Data** - define which kind of properties can be related to elements
- **Concepts** - define the set of types expected to find using a taxonomy, a hierarchical tree where is expressed the notion about types and subtypes
- **Relations** - define a set of relations between elements
- **Individuals** - some predefined instances with associated data. E.g, a specific device or a specific robot

The world model is compliant with the OWL w3c standard that counts several tools for managing ontologies. A well-known open-source program is Protege (<http://protege.stanford.edu/>). To install and use it, the reader can refer to one of the several guides available on internet.⁶ As introduced in Sect. 3.2, it is possible to have several custom ontologies in the defined OWL path and these get automatically loaded and merged with the SkiROS knowledge core at boot. The reader can refer to the `uav.owl` file in the `skiros_simple_uav` repository to see a practical example on how to create an ontology extension. The launch file in the same package provide an example on how to load it. Note in particular the importance of providing the right ontology *prefix* in the launch file and in general when referencing entities in the ontology (Fig. 8).

The entities defined in the ontology are going to constraint the development of skill and primitives. For example, a place skill will require as input only objects that are subtypes of `Container`. To avoid the use of strings in the code, that are impossible to track down, an utility has been implemented in the `skiros_world_model` package to generate an enum directly from the ontology. It is possible to run this utility with the command:

```
rosrun skiros_world_model uri_header_generator
```

This utility updates the `skiros_config/declared_uri.h`, automatically included in all modules. Using the generated enum for the logic queries allows to get an error at compile time, if the name changes or is missing.

Create a new robot definition The semantic robot structure is the necessary information for the skill manager to manage the available hardware. E.g. if the robot has a camera mount on the arm, it can move the camera to look better at an object. The robot and its devices must be described in detail with all the information that the developer wants to have stated explicitly. The user should use protege to create an ontology with an *individual* for each device and an *individual* for the robot, collecting the devices using the *hasA* relation. To give a concrete example, lets consider the `aau_stamina_robot`, the smaller stamina prototype used for laboratory test:

- **NamedIndividual**: `aau_stamina_robot`
- **Type**: Robot
- **LinkedToFrameId**: `base_link`
- **hasA** – >`top_front_camera`
- **hasA** – >`top_left_camera`

⁶e.g. <http://protegewiki.stanford.edu/wiki/Protege4GettingStarted>.



Fig. 8 The taxonomy of spatial things for the kitting application

- *hasA* – > top_right_camera
- *hasStartLocation* – > unknown_location
- *hasA* – > ur10

The robot has a `LinkedToFrameId` property, related to the `AauSpatialReasoner`, and a start location, used for the drive skill. For more information about this properties, refer to the plug-ins description. The robot hardware consist of 3 cameras and a robotic arm (ur10). If we expand the ur10 description we find:

- **NamedIndividual:** ur10
- **Type:** Arm
- **MoveItGroup:** arm
- **DriverAddress:** /ur10
- **MotionPlanner:** planner/plan_action
- **MotionExe:** /arm_controller/follow_joint_trajectory

- *hasA – >arm_camera*
- *hasA – >rq3*

The arm has an additional camera and a end-effector rq3. Moreover, it has useful properties for the configuration of MoveIt. Thanks also to the simple parametrization of MoveIt, we have been able to port the *same* skills on 3 heterogeneous arms (ur, kuka and fanuc) by changing only the arm description as presented here.

5.2 Create a Primitive

A SkiROS module is a C++ software class based on the standard ROS plug-in system. In particular, those who are experienced in programming ROS nodelets⁷ will probably find it straightforward to program SkiROS modules. Developing a module consists of programming a C++ class derived from an abstract template. The basic module, or primitive, usually implements an atomic functionality like opening a gripper, locating an object with a camera, etc. These functionalities can be reused in other primitives or skills or executed individually from the module tab. A primitive inherits from the template defined in `skiros_skill/module_base.h`. It requires to specify the following virtual functions:

```

1 //! \brief personalized initialization routine
2 virtual bool onInit() = 0;
3 //! \brief module main
4 virtual int execute() = 0;
5 //! \brief specialized pre-preempt routine
6 virtual void onPreempt();
```

The `onInit()` function is called when the skill manager is started to initialize the primitive. The `execute()` function is called when the primitive is executed from the GUI or called by another module. The `onPreempt()` function is called when the primitive is stopped, e.g. from the GUI. All primitives have protected access to a standard set of interfaces:

```

1 //! \brief Interface with the parameter set
2 boost::shared_ptr<ParamHandler> getParamHandler();
3 //! \brief Interface with the skiros world model
4 boost::shared_ptr<WorldModelInterfaceS> getWorldHandler();
5 //! \brief Interface to modules
6 boost::shared_ptr<SkillManagerInterface> getModulesHandler();
7 //! \brief Interface with the ROS network
8 boost::shared_ptr<NodeHandle> getNodeHandler();
```

The `ParamHandler` allows to define and retrieve parameters.

The `WorldModelInterface` allows to interact with the world model. The primitive's parameter set must be defined in the `class constructor` and never modified afterwards.

⁷<http://wiki.ros.org/nodelet>.

The SkillManagerInterface allows the primitive to interact with other modules.

World model interface Modules' operations apply over an abstract world model, which has to be constantly matched to the real world. There is no space in this chapter to describe in detail the world model interface. Nevertheless, it is important to present the atomic world model's data type, defined as *element*. In fact, the element is the most common input parameter for a module. An element structure is the following:

```

1 //Unique Identifier of the element in the DB
2 int id;
3 //Individual identifier in the ontology
4 std::string label;
5 //Category identifier in the ontology
6 std::string type;
7 //Last update time stamp
8 ros::Time last_update;
9 //A list of properties (color, pose, size, etc.)
10 std::map<std::string, skiros_common::Param> properties;
```

The first 3 fields are necessary to relate the element in the ontology (label and type) and the scene database (id). The properties list contains all relevant information associated to the object.

Parameters Every module relies on a dynamic set of parameters to configure the execution. The parameters are divided in the following categories:

- *online* parameters that must always be specified
- *offline* usually are configuration parameters with a default value, such as the desired movement speed, grasp force, or stiffness of the manipulator
- *optional* like an offline parameter, but can be left unspecified
- *hardware* indicate a robot's device the module need to access. This can be changed at every module call (e.g. to locate with different cameras)
- *config* like an hardware parameter, but it is specified when the module loads and *cannot* be changed afterwards. (e.g. an arm motion module bounded to a particular arm)
- *planning* a parameter necessary for pre and post condition check in a skill. This is set automatically and doesn't appear on the UI.

To understand the concept, we present a code example. First, we show how to insert a parameter:

```

1 getParamHandler() -> addParam<my::Type>("myKey", "My description", ←
    skiros_common::online, 3);
```

Here we are adding a parameter definition, specifying in the order: the key, a brief description, the parameter type, and the vector length. The template argument has to be specified explicitly too. In the above example we define an online parameter as a vector of 3 doubles. The key myKey can be used at execution time to access the parameter value:

```
1 std::vector<double> myValue = getParamHandler()->getParamValues<double>("myKey");
```

The parameter state is defined as initialized until its value get specified. After this the state changes to specified. A module cannot run until all unspecified parameters are specified. It is also possible to define parameters with a default value:

```
1 getParamHandler()->addParamWithDefaultValue("myKey", true, "My ← description", skiros_common::offline, 1);
```

In this case, the parameter will be initialized to true. When an input parameter is a world's element, it is possible to apply a special rule to limit the input range. In fact sometimes a module requires a precise type of element as input. For example, a pick skill can pick up only elements of type Manipulatable. In this case, it is possible to use a partial definition. For example:

```
1 '/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.34:
2 getParamHandler()->addParamWithDefaultValue("object", skiros_wm::Element("Manipulatable"), "Object to pick up");
```

In this example, only subtypes of Manipulatable will be valid as input for the object parameter. Every module can have a customized amount of parameters. The parameters support any data type that can be serialized in a ROS message. This means all the standard data types and all the ROS messages. ROS messages requires a quick but non-trivial procedure to be included in the system, that is excluded from the chapter for space reasons.

Invoke modules Each module can recursively invoke other modules' execution. For example, the pick skill invoke the locate module with the following:

```
1 '/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.155:
2 skiros::Module locate(getModulesHandler(), "locate_fake", this->moduleType());
3 locate.setParam("Camera", camera_up_);
4 locate.setParam("Container", container_);
5 locate.exe();
6 locate.waitResult();
7 v = getWorldHandle()->getChildElements(container_, "", objObject.type());
```

Here, line 2 instantiate a proxy class for the module named locate_fake. Line 3 and 4 set the parameters and line 5 request the execution. The execution is non blocking, so that is possible to call several modules in parallel. In this case, we wait for the execution end and then we retrieve the list of located object.

5.3 Create a Skill

A skill is a complex type of module, which inherits from the template defined in `skiros_skill/skill_base.h`. A conceptual model of a complete robot skill is shown in Fig. 9. A skill extends the basic module definition with the presence of pre- and postcondition checks. By implementing pre- and postcondition checking procedures the skills themselves verify their applicability and outcome. This enables the skill-equipped robot to alert an operator or task-level planner if a skill cannot be executed (precondition failures) or if it did not execute correctly (postcondition failures). A formal definition of pre- and postconditions is not only useful for robustness, but also task planning, which utilizes the preconditions and prediction to determine the state transitions for a skill, and can thus select appropriate skills to be concatenated for achieving a desired goal state. A skill adds also two more virtual functions, `preSense()` and `postSense()`, where sensing routines can be called before evaluating pre- and postconditions. Internally, a skill results into a concatenation of primitives, that can be both sequential or parallel. The planned sequence of skills forms the highest level of execution, that is dynamically concatenated and parametrized at run-time. The further hierarchical expansion of primitives is scripted by the developer, but still modular, so that its parts can be reused in different pipelines or replaced easily. Still, if the developer has some code that doesn't want to modularize, e.g. a complete pick pipeline embedded in an ROS action, it is allowed, but unrecommended, to implement a skill as a single block that only makes the action call.

Create a condition Preconditions and postconditions are based on sensing operations and expected changes to the world model from executing the skill. The user defines the pre- and postconditions in the skill `onInit()` function, after the parameter definitions. The conditions can be applied only on world model's `elements` input parameters. While some ready-to-use conditions are available in the system, it's also possible to create a new condition by deriving it from the

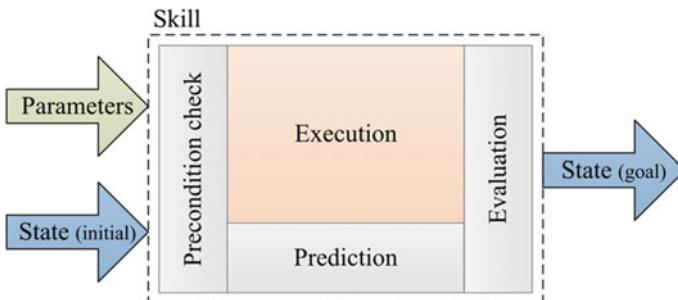


Fig. 9 The model of a robot skill [21]. Execution is based on the input parameters and the world state, and the skill effectuates a change in the world state. The checking procedures before and after execution verify that the necessary conditions are satisfied

condition templates `skiros_wm/condition.h`. There are two base templates: `ConditionRelation` and `ConditionProperty`. The first put a condition on a relation between two individuals. The second put a condition on a property of an individual. When implementing a new condition, two virtual functions have to be implemented:

- **void init()** - here define the property (or the relation) on which the condition is applied
- **bool evaluate()** - a function that returns true if the condition is satisfied or false otherwise

Once the condition is defined, every skill can add it to its own list of pre or postconditions in the `onInit()` function. For example:

```
1 '/skiros_std_lib/skiros_lib_dummy_skills/src/pick.cpp' L.62:
2 addPrecondition(newCondition("RobotAtLocation", true, "Robot", "←
  Container"));
```

This command instantiate a new condition `RobotAtLocation` between `Robot` and `Container` and add it to the list of preconditions. Note that `Robot` and `Container` refer to the key defined in the input parameters. In this case, the skill requires the `Robot` parameter to have a specific relation with the `Container` parameter. If this relation doesn't hold, the skill will return a failure without being executed.

5.4 Create a Discrete Reasoner

The world's *elements* are agnostic placeholders where any kind of data can be stored and retrieved. Their structure is general and flexible, but this flexibility requires that no data-related methods are implemented. The methods are therefore implemented in another code structure, called *discrete reasoner*, that is imported in the SkiROS system as a plug-in. Any reasoner inherits from the base class `skiros_world_model/discrete_reasoner.h`. The standardized interface allow to use the reasoners as utilities to (i) store/retrieve data to/from elements and (ii) to reason about the data to compare and classify elements at a semantic level.

Spatial reasoner A fundamental reasoner for manipulation is the spatial reasoner, developed specifically to manage position and orientation properties. The `AauSpatialReasoner`, an implementation based on the standard 'tf' library of ROS, is included in the `skiros_std_lib/skiros_lib_reasoner` package. An example of the reasoner use is in the following:

```
1 container_ =getParamHandler()->getParamValue<skiros_wm::Element>("←
  Container");
2 skiros_wm::Element object;
3 object.type()=concept::Str[concept::Compressor];
4 object.storeData(tf::Vector3(0.5,0.0,0.0),data::Position, "←
  AauSpatialReasoner");
```

```

5 tf :: Quaternion q;
6 q.setRPY(0.0,0.0,0.0);
7 object.storeData(q, data::Orientation);
8 object.storeData(string("map"), data::BaseFrameId);
9 tf::Pose pose= object.getData<tf::Pose>(data::Pose);
10 std::set<std::string> relations = object.getRelationsWrt(container_)<-
    ;

```

In this example, we first get the container variable from the input parameters. Then we create an new object instance and we use the AauSpatialReasoner reasoner to store a position, an orientation and the reference frame. Note that it necessary to specify the reasoner only on the first call. At line 8, we get back the object pose (a combination of position and orientation). At line 9 we use the reasoner to calculate semantic relations between the object itself and the container. The relations will contain predicates like front/back, left/right, under/over, etc. It is also possible to get relations with associated literal values for more advanced reasoning. It is up to the developer to define the supported data structures in I/O and which relevant semantic relations are extracted.

Example To give an example of some of the concepts presented in the section, lets consider the code of a skill to start the flying of an UAV (note: the code is slightly simplified w.r.t. the real file):

```

1 '/skiros_simple_uav/simple_uav_skills/src/flyToAltitude.cpp':
2 class FlyAltitude : public SkillBase
3 {
4 public:
5     FlyAltitude()
6     {
7         this->setSkillType("Drive");
8         this->setVersion("0.0.1");
9         getParamHandle()->addParamWithDefaultValue("Robot", <-
10             skiros_wm::Element(concept::Str[concept::Robot]), "Robot<-
11                 to control", skiros_common::online);
12         getParamHandle()->addParamWithDefaultValue("Altitude", 1.0, <-
13             "Altitude to reach (meters)", skiros_common::offline);
14     }
15     bool OnInit()
16     {
17         addPrecondition(newCondition("uav:LowBattery", false, "Robot<-
18             "));
19         addPostcondition("NotLanded", newCondition("uav:Landed", <-
20             false, "Robot"));
21         return true;
22     }
23     int preSense()
24     {
25         skiros::Module monitor(getModulesHandler(), "monitor_battery" <-
26             , this->skillType());
27         monitor.setParam("Robot", getParamHandle()->getParamValue<<-
28             Element>("Robot"));
29         monitor.setParam("f", 10.0);
30     }

```

```

23     monitor.exe();
24     return 1;
25 }
26 int execute()
27 {
28     double altitude = getParamHandle()->getParamValue<double>("Altitude");
29     this->setProgress("Going to altitude" + std::to_string(altitude));
30     ros::Duration(2.0).sleep(); //Fake an execution time
31     setAllPostConditions();
32     return 1;
33 }
34 };
35 //Export
36 PLUGINLIB_EXPORT_CLASS(FlyAltitude, skiros_skill::SkillBase)

```

Lets go through the code line by line:

- **Constructor** - line 7–8 define constants to describe the module itself. Line 9–10 define the required parameters.
- **onInit()** - line 14 add the precondition of having a charged battery, line 15 add a postcondition of having the robot no more on the ground. Note the use of the prefix ‘uav:’ to the condition names. This because the conditions are defined in the `uav.owl` ontology.
- **preSense()** - invoke the `monitor_battery` module, to update the condition of the battery.
- **execute()** - at line 28 the parameter `Altitude` is retrieved. Line 29 print out a progress message. Line 30 and 31 are in place of a real implementation. In particular, the `setAllPostConditions()` command set all postconditions true, in order to simulate the execution at a high-level. Line 32 return a positive value, to signal that the skill terminated correctly.

At the very end, line 36 exports the plug-in definition.

6 Task Planner

The Task Planner’s function is to provide a sequence of instantiated skills that, when carried out successfully, will lead to a desired goal state that has been specified by the user or some other automated process. For example, the goal state may be that a certain object has been placed in a kit that is being carried by the robot, and the returned sequence of skills (the plan) may be to *drive* to the location where the object can be found, to *pick* the object, and then to *place* the object in the kit. Of course, real goals and plans can be much more complicated than this, only limited by the relations that exist in the world model and the skills that have been defined.

This section discusses the general translation algorithm to the Planning Domain Definition Language (PDDL) aswell as the additions tailored for the STAMINA use-

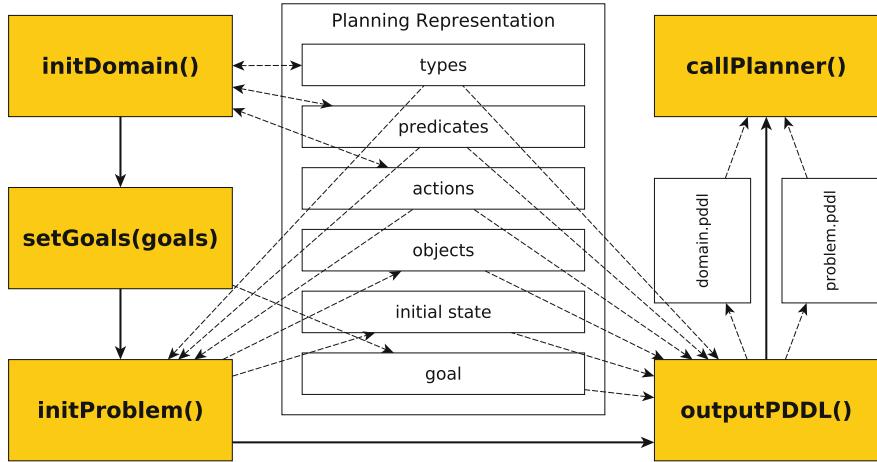


Fig. 10 An overview of the Task Planning pipeline. The inputs are the set of goals and the world scene. The output is a sequence of instantiated skills. The solid arrows represent execution flow. The dashed arrows pointing towards the data structures represent writing or modification while the dashed arrows pointing away from the data structures represent read access

case. PDDL is a well-known and popular language for writing automated planning problems and, as such, is supported by a large array of-the-shelf planners. The PDDL files created with the basic Task Planner options only use the types requirement of the original PDDL 1.2 version and are therefore suitable for use with almost all existing planners. Some extra features of the Task Planner (see Sect. 6.3) can be employed which introduce *fluent*s and *durative actions*, and therefore must be used with a PDDL 2.1 compatible temporal planner. The Task Planner is built as a plugin to SkiROS that generate generic PDDL so that the developer can insert whichever external planning algorithm they wish to use.

6.1 Overview and Usage

The Task Planner exists as a plug-in for SkiROS (`skiros_std_lib/task_planners/skiros_task_planner_plugin`) that creates a PDDL problem using the interface provided in `skiros_task/pddl.h`. The Task Planner contains two const's; `robotParameterName`, and `robotTypeName` which default to `Robot` and `Agent` respectively and must be consistent with the robot names used in the skill and world model definitions. Additionally, `pddl.h` contains a const bool `STAMINA` that toggles the use of specific extra features (explained in Sect. 6.3). Operation of the Task Planner is split into five main functions, as shown in Fig. 10 and described below:

- **initDomain** - This function causes the Task Planner to translate the skill information in the world model into the planning actions and predicates found in the planning domain. While this translation is based on the preconditions and postcondition defined in the skill, it is not direct, and details of the necessary modifications are given below.
- **setGoal** - This function sets the goal, or set of goal predicates, to be planned for. These can be provided as SkiROS *elements* or as PDDL strings.
- **initProblem** - This function takes no arguments and tells the Task Planner to query the world model to determine the initial state of the planning problem. This must be called after the previous two functions as it relies on them to work out which parts of the world model are relevant to the planning problem.
- **outputPDDL** - This function prints out a pddl domain file *domain.pddl* and problem file *p01.pddl* in the task planner directory.
- **callPlanner** - This function invokes an external planner that will take the previously output PDDL files as input and return a plan. This must be implemented by the user for whichever external planner they wish to use. The planner has been tested with Fast Downward⁸ for the general case and Temporal Fast Downward⁹ for the STAMINA use-case. Any plan found must then be converted to a vector of parameterised skills, so any extra parameters created for internal use by the planner must be removed at this point.

The user only needs to specify the external call to the planner in the callPlanner function. The setGoal function is the only one that takes arguments and requires a goalset comprised of SkiROS *elements* or PDDL strings. The other functions interface directly with the world model and require no arguments.

6.2 From Skills to PDDL

The design of the SkiROS skills system facilitates the translation to a searchable A.I. planning format. However, a direct translation from skills to planning actions is not possible, or even desirable. The preconditions and postconditions of skills should be definable by a non-planning expert based on the precondition and postcondition checks required for safe execution of the skill along with any expected changes to the world model. Therefore, the translation algorithm is required to do some work to generate a semantically correct planning problem. We will briefly discuss two important points; how it deals with heterogeneous robots, and implicitly defined transformations.

Heterogenous Robots There may be multiple robots in the world, each with different skill sets. For example, in the STAMINA use-case, the mobile platform is defined as

⁸<http://www.fast-downward.org/>.

⁹gki.informatik.uni-freiburg.de/tools/tfd/.

a separate robot to the robotic arm. The mobile platform has the *drive* skill while the gripper has the *pick* and *place* skills.

To ensure that each skill ‘s’ can only be performed by the relevant robot, a ‘can_s’ predicate is added as a precondition to each action, so that the action can only be performed if ‘can_s(r)’ is true for robot ‘r’. ‘can_s(r)’ is then added to the initial state of the problem for each robot ‘r’ that has skill ‘s’. This way the planner can plan for multiple robots at a time.

Implicit Transformations The skill definitions may include implicit transformation assumptions that need to be made explicit for the planner. For example, the STAMINA drive skill is implemented with the following condition:

```
1 addPostcondition("AtTarget", newCondition("RobotAtLocation", true, "↔
  Robot", "TargetLocation"));
```

That is, only a single postcondition check, that the robot is at the location it was meant to drive to. For updating the SkiROS world model, setting the location of the robot to the ‘TargetLocation’ will automatically remove it from its previous location. However, the planner needs to explicitly encode the deletion of the previous location otherwise the robot will end up in two places at once in its representation.

Of course, it is possible to add the relevant conditions to the skill definition. However, this is not ideal as it would mean that the robot performs verification that it is not at the previous location at the end of the drive skill. This prohibits the robot from driving from its location to the same location as the new postcondition check would fail. Whether this is a problem or not, allowing the Task Planner to automatically include explicit updates reduces the pressure on the skill writer to produce both a skill definition that is correct in terms of both the robot and the internal planning representation.

The transformation is performed in a general manner that works for all spatial relations. The skills in the planning library are iterated over and checked against the spatial relations defined in SkiROS. If spatial relations are found to be missing, in either the preconditions or delete effects of the action (i.e., no predicate with matching relation and subject as in the case of the drive skill), then a new predicate of the same spatial relation and the same object, but a new subject variable, is created and added to the preconditions and delete effects of the action. If a related spatial relation exists in just one of the preconditions and delete effects then it is added (with the same subject) to the other. More details of the planning transformation algorithm can be found in [25].

6.3 Additional Features

The task planner contains additional features used in the STAMINA problem instance that can either be enabled or disabled depending on user preference. These extra features include sequence numbers for ordering navigation through the warehouse, for

which the planner employs numeric fluents and temporal actions, and abstract objects for which the planner must add internal parameters to ensure correct execution.

Sequence Numbers In the usecase for STAMINA, the robot navigates around a warehouse following a strict path. This is enforced following the previous setup to ensure that human workers will always exit in the same order they entered, therefore preserving the output order of the kits they are creating. The locations that it is possible to navigate to are given a sequence number (by a human operator) and these numbers are used to determine the shortest path based on the particular parts in the current order.

Abstract Objects In the world model for STAMINA, parts are not instantiated until they are actually picked up. The pick skill is called on an abstract object because it is not known before execution which of the possibly numerous objects in a container will be picked up. On the other hand, the place skill is often called for an instantiated object, as, at the time of execution, it is a particular instance of an object that is in the gripper (Fig. 11).

7 Application Example

7.1 Overview

As an example application, we focus on a logistic operation and specifically consider the automation of an industrial kitting operation. Such operation is common in a variety of manufacturing processes since it involves the navigation of a mobile platform to various containers from which the objects are picked and placed in their



Fig. 11 The two different hardware setups that have been used for evaluation

corresponding compartments of a kitting box. Thus, in order to achieve this task we have developed three skills, namely the **drive**, **pick** and **place** which consist of a combination of primitives such:

- **locate** - roughly localize an object on a flat surface using a camera
- **object registration** - precisely localize an object using a camera
- **arm motion** - move the arm to a desired joint state or end-effector pose
- **kitting box registration** - localization of the kitting-box using a camera
- **gripper ctrl** - open and close the gripper

Ontology The ontology that represents our specific kitting domain has been defined starting from the general ontology presented in Fig. 8. We extended the ontology with the types of manipulatable objects (starter, alternator, compressor, etc.) and boxes (pallet, box, etc.). Second, the following set of conditions has been defined: FitsIn, EmptyHanded, LocationEmpty, ObjectAtLocation, Carrying, Holding, RobotAtLocation.

Learning primitives Learning primitives are needed to extend the robot's knowledge base with some important information about the environment. The learning primitives, in our case, are:

- **object train** - record a snapshot and associate it to an object's type specified from the user
- **grasping pose learn** - learn a grasping pose w.r.t. an object's snapshot
- **placing pose learn** - learn a placing pose w.r.t. a container
- **driving pose learn** - learn a driving pose w.r.t. a container

The primitives are executed from the GUI module tab during an initial setup phase of the robot. The snapshot and the poses taught during this phase are then used for all subsequent skills' execution.

7.2 Skills

The **Drive Skill** is the simplest skill. It is based on the standard ROS navigation interface, therefore the execution consist of an action call based on the 'move_base_msgs::MoveBaseGoal' message. The details about navigation's implementation are out of the scope of this chapter, but more implementation details can be found in [26].

The **Picking Skill** pipeline is organized in several stages, it 1. detects the container (pallet or box) using one of the workspace cameras, 2. moves the wrist camera over the detected container to detect and localize parts, and 3. picks up a part using predefined grasps. Low cycle times of roughly 40–50 s are achieved by using particularly efficient perception components and pre-computing paths between common paths to save motion planning time. Figure 15 shows examples of part picking using different mobile manipulators in different environments. The picking skill distinguishes two

types of storage containers: pallets in which parts are well separated and boxes in which parts are stored in unorganized piles. Internally, the two cases are handled by two different pipelines which, however, follow the same three-step procedure as mentioned above. In case of pallets, we first detect and locate the horizontal support surface of the pallet and then segment and approach the objects on top of the pallet for further object recognition, localization and grasping [3]. For boxes, we first locate the top rectangular edges of the box and then approach an observation pose above the box center to take a closer look inside and to localize and grasp the objects in the box [27]. In the following, we will provide further details about these two variants of the picking pipeline and how the involved components are implemented as a set of primitives in the SkiROS framework. An example of this three-step procedure for grasping a part from a transport box is shown in Fig. 12.

The Placing Skill is responsible for reliable and accurate kitting of industrial parts in confined compartments [28]. It consists of two main modules the *arm motion* and the *kit locate*. The first is responsible for reliable planning and execution of collision-free trajectories subject to task-depended constraints, while the *kit locate* is responsible for the derivation of the kitting-box pose. The high precision and reliability of both is crucial for a successful manipulation of the objects in the confined compartments of the kitting box.

7.3 Primitives

The locate primitive is one of the perception components in the picking pipeline and locates the horizontal support surfaces of pallets. In addition it segments the objects on top of this support surface, selects the object to grasp (object candidate being

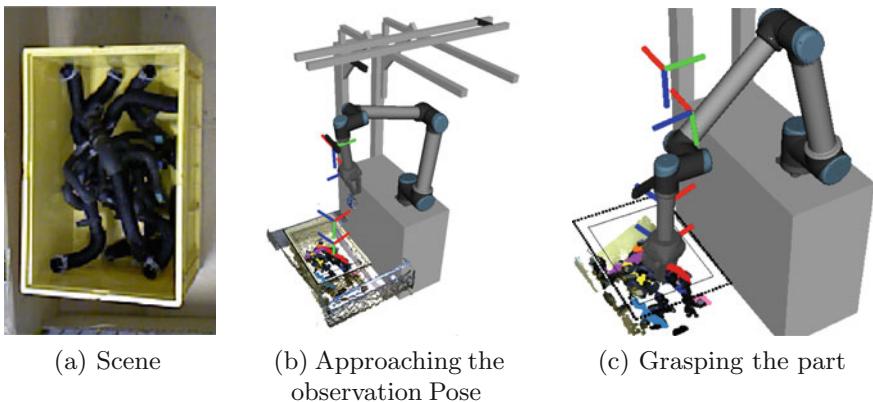


Fig. 12 Example of grasping a tube connector from a transport box (a): after detecting the box and approaching the observation pose (b), the part is successfully grasped (c)

closest to the pallet center) and computes an observation pose to take a closer look at the object for recognition and localization. The detection of the horizontal support surface is based on very fast methods for computing local surface normals, extracting points on horizontal surfaces and fitting planes to the extracted points [3]. In order to find potential object candidates, we then select the most dominant support plane, compute both convex hull and minimum area bounding box, and select all RGB-D measurements lying within these polygons and above the extracted support plane. We slightly shrink the limiting polygons in order to neglect measurements caused by the exterior walls of the pallet. The selected points are clustered (to obtain object candidates), and the cluster being closest to the center of the pallet is selected to be approached first.

After approaching the selected object candidate with the end effector, the same procedure is repeated with the wrist camera in order to separate potential objects from the support surface. Using the centroid of the extracted cluster as well as the main axes (as derived from principal component analysis), we obtain a rough initial guess of the object pose. With the subsequent registration stage, it does not matter when objects are not well segmented (connected in a single cluster) or when the initial pose estimate is inaccurate.

The **registration primitive** accurately localized the part and verifies whether the found object is the correct part or not. The initial part detection only provides a rough estimate of the position of the object candidate. In order to accurately determine both position and orientation of the part, we apply a dense registration of the extracted object cluster against a pre-trained model of the part. We use multi-resolution surfel maps (MRSMAPs) as a concise dense representation of the RGB-D measurements on an object [29]. In a training phase, we collect one to several views on the object whose view poses can be optimized using pose graph optimization techniques. The final pose refinement approach is then based on a soft-assignment surfel registration. Instead of considering each point individually, we map the RGB-D image acquired by the wrist camera into an MRSMAP and match surfels. This needs several orders of magnitudes less map elements for registration. Optimization of the surfel matches (and the underlying joint data-likelihood) yields the rigid 6 degree-of-freedom (DoF) transformation from scene to model, i.e., the pose of the object in the coordinate frame of the camera.

After pose refinement, we verify that the observed segment fits to the object model for the estimated pose. We can thus find wrong registration results, e.g., if the observed object and the known object model do not match or if a wrong object has been placed on the pallet. In such cases the robot stops immediately and reports to the operator (a special requirement of the end-user). For the actual verification, we establish surfel associations between segment and object model map, and determine the observation likelihood similar as in the object pose refinement. In addition to the surfel observation likelihood, we also consider occlusions by model surfels of the observed RGB-D image as highly unlikely. Such occlusions can be efficiently determined by projecting model surfels into the RGB-D image given the estimated alignment pose and determining the difference in depth at the projected pixel position.

The resulting segment observation likelihood is compared with a baseline likelihood of observing the model MRSMAP by itself. We determine a detection confidence from the re-scaled ratio of both log likelihoods thresholded between 0 and 1.

The arm motion primitive exploits many capabilities of MoveIt software¹⁰ such as the Open Motion Planning Library (OMPL), a voxel representation of the planning scene and interfaces with the move-group node. In order to achieve motions that are able to successfully place an object in compartments which in many cases are very confined, we have developed a planning pipeline by introducing two deterministic features which could be anticipated as planning reflexes.

The need for that addition arises due to the stochasticity of the OMPL planners which makes them unstable as presented in preliminary benchmarking tests [28]. The Probabilistic Road-maps (PRM), Expansive-Spaces Tree (EST) and Rapidly exploring Random Tree (RRT) algorithms were evaluated. Based on the results PRM performs better on the kitting task. However, its success rate is not desirable for industrial applications.

Another deterrent factor on motion planning is the Inverse Kinematics (IK) solutions that derive from MoveIt. Although that there exist multiple IK solutions for a given pose, MoveIt functions provide only one which is not always the optimal i.e. the closest to the initial joint configuration. We deal with this problem by sampling multiple solutions from the IK solver with different seeds. The IK solution whose joint configuration is closer to the starting joint configuration of the trajectory is used for planning.

The developed planning pipeline achieves repeatable and precise planning by introducing two planning reflexes, the joint and operational space linear interpolations. The first ensures that the robot's joints will rotate as less as possible and can be anticipated as an energy minimization planner. This happens by linearly interpolating, in the joint space of the robot, between the starting and the final configurations. Additionally, the operational space interpolation results to a linear motion of the end-effector in its operational space. This is achieved by performing a linear interpolation between the starting and final poses. Furthermore, the spherical linear interpolation (slerp) is used for interpolating between orientations. This linear motion is desirable for going in the narrow compartments of the kitting box.

The path that is created from the two reflexes is evaluated for collisions, constraint violations and singularities. If any of those happen then the pipeline employs the more sophisticated PRM algorithm for solving the planning problem.

The kitting box registration primitive is responsible to locate the kitting box in which compartments the grasped objects have to be placed. Usually, the pose estimation of the kitting box has to be executed whenever a new kitting box arrives in the placing area of the robot, however, due to slight changes in its position or orientation occurring during the placing task, the kitting box registration can be used as part of any skill, e.g. in the beginning of the placing skill.

¹⁰<http://moveit.ros.org>.

For the pose estimation of the box, an additional workspace camera with a top view on the kitting area is used. Due to the pose of the camera and objects which are already placed in the kitting box, most of it is not visible except for the top edges. Additionally, parts of the box edges are distorted or missing in the 3D data caused by the noise of the camera. The kitting box registration therefore implements an approach based on a 2D edge detection on a cleaned and denoised depth image. It applies a pixelwise temporal voting scheme to generate potential points belonging to the kitting box. After mapping these back to 3D space, a standard ICP algorithm is utilized to find the kitting box pose (cmp. Fig. 13). A more detailed description of the algorithm and a performance evaluation is presented in [28].

7.4 Results

We have evaluated the presented architecture with two robotic platforms that operate on different environments. A stationery Universal Robotics UR10 robot that operates within a lab environment and a Fanuc M-20iA which is mounted on a mobile platform and operates within an industrial environment. Both robotic manipulators

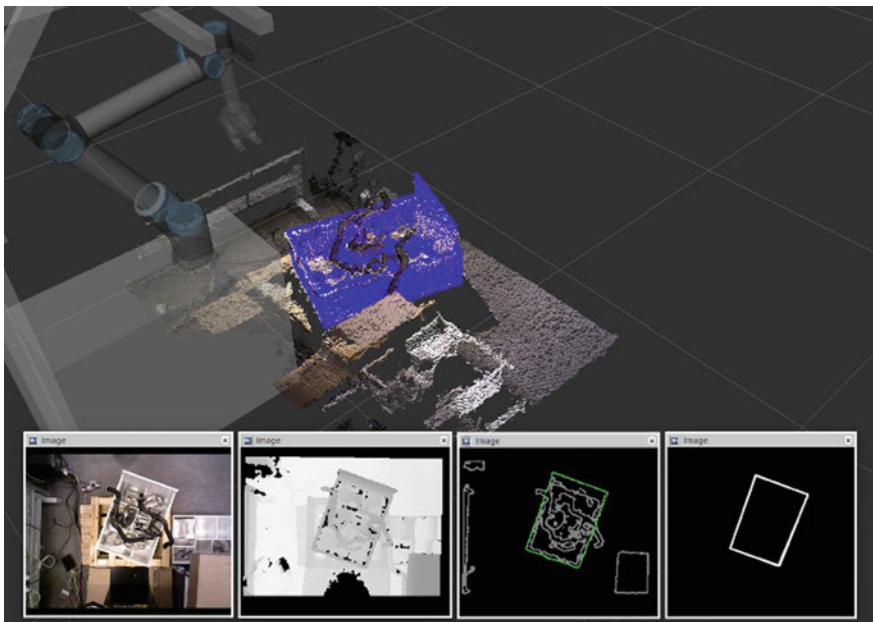


Fig. 13 The 3D scene shows a highly cluttered kitting box with the final registered kitting box (*blue*) overlaid. The images in the *bottom (left to right)* show different steps of the registration process: **a** raw RGB image **b** normalized depth image **c** after edge extraction with box candidates overlaid in green **d** final voting space for box pixels

are equipped with a Robotiq 3-Finger Adaptive Gripper and a set of RGB-D sensors. The drivers that have been used for control of both robotic manipulators and the gripper are available from the ROS-Industrial project.¹¹

Kitting Operation with UR10 We evaluate the whole kitting task on the UR10 robot with various manipulated objects. The task was planned using the Graphical User Interface presented in Sect. 4 and is a concatenation of the pick and place skills. Figure 14 illustrates the key steps of the kitting tasks. Detailed results on the performance of the kitting operation can be found in [2, 28].

Kitting Operation with Fanuc on mobile platform Additionally to the UR10 test case, where the robot is stationary and operates in a lab environment, we have applied the presented architecture on a Fanuc robot which is mounted on a mobile base. In this case the kitting operation consists of three skills, the drive, pick and place. Thus, the robot navigates at the location of the requested object, picks it and then places it in the kitting box. This sequence is illustrated in Fig. 15. Using the presented architecture the mobile manipulator is able to perform kitting operations with multiple objects that are located in various spots.

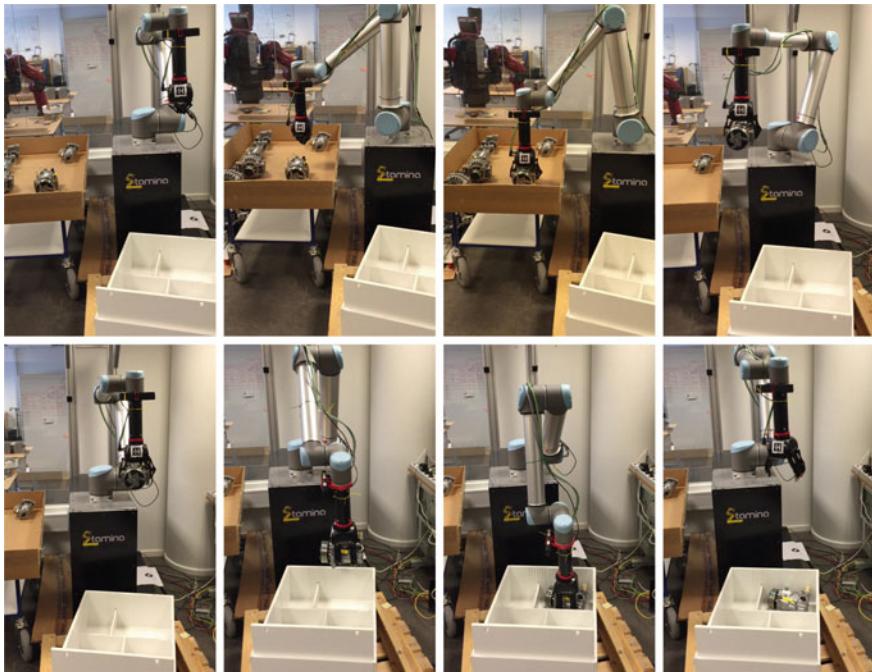


Fig. 14 Sequence of a kitting operation in the lab environment. The *top row* illustrates the execution of the pick skill and the *bottom row* the execution of placing skill

¹¹<http://rosindustrial.org/>.

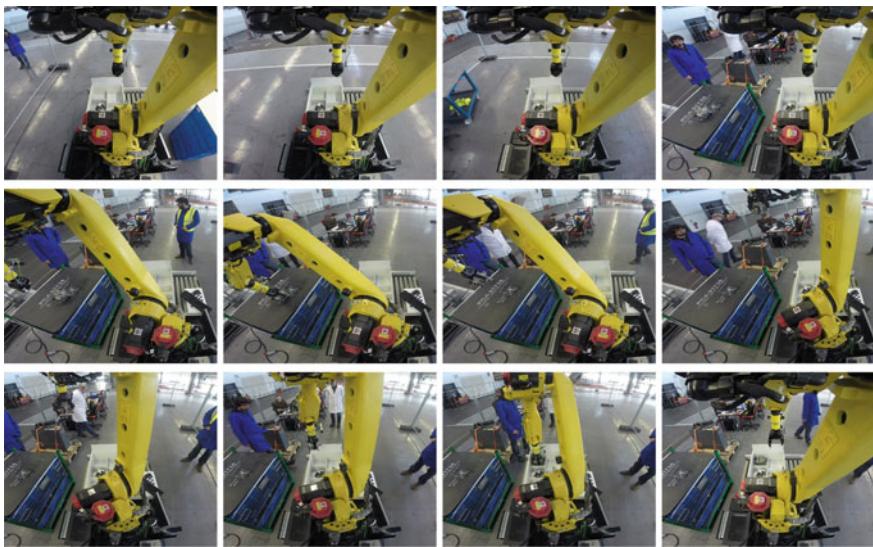


Fig. 15 Sequence of a kiting operation in an industrial environment. The *top row* illustrates the drive skill, the *middle row* the picking skill and the *bottom row* the placing skill

8 Conclusions

In this research chapter we presented SkiROS, a skill-based platform to develop and deploy software for autonomous robots, and an application example on a real industrial use-case. The platform eases the software integration and increases the robot reasoning capabilities with the support of a knowledge integration framework. The developer can split complex procedures into ‘‘skills’’, that get composed automatically at run-time to solve goal oriented missions. We presented an application example where a mobile manipulator navigated in the warehouse, picked parts from pallets and boxes, and placed them in kitting boxes. Experiments conducted in two laboratory environments, and at the industrial end-user site gave a proof-of-concept of our approach. ROS joined with SkiROS allowed for porting the pipelines on several heterogeneous mobile manipulator platforms. We believe that using SkiROS, the pipelines can integrate easily with other skills for other use-cases, e.g. for assembly operations. The code released with the chapter allows any ROS user to try out the platform and plan with a fake version of the drive, pick and place skills. The user can then add their own skill definitions and pipelines to the platform and use SkiROS to help implement and manage their own robotics systems.

References

1. Pedersen, M.R., L. Nalpantidis, R.S. Andersen, C. Schou, S. Bøgh, V. Krüger, and O. Madsen. 2015. Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*. Available online.
2. Holz, D., A. Topalidou-Kyniazopoulou, F. Rovida, M.R. Pedersen, V. Krüger, and S. Behnke. 2015. A skill-based system for object perception and manipulation for automating kitting tasks. In *Proceedings of the IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*.
3. Holz, D., A. Topalidou-Kyniazopoulou, J. Stückler, and S. Behnke. 2015. Real-time object detection, localization and verification for fast robotic depalletizing. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany*, 1459–1466.
4. McDermott, D. 2000. The 1998 ai planning systems competition. *Artifical Intelligence Magazine* 21 (2): 35–55.
5. Kortenkamp, D., and R. Simmons. 2007. Robotic systems architectures and programming. In *Springer Handbook of Robotics*, ed. B. Siciliano, and O. Khatib, 187–206. Heidelberg: Springer.
6. Arkin, R.C. 1998. *Behavior-based Robotics*, 1st ed. Cambridge: MIT Press.
7. Brooks, R.A. 1986. A robust layered control system for a mobile robot. *Journal of Robotics and Automation* 2 (1): 14–23.
8. Firby, R.J. 1989. Adaptive Execution in Complex Dynamic Worlds. Ph.D. thesis, Yale University, USA.
9. Gat, E. 1998. On three-layer architectures. In *Artificial Intelligence and Mobile Robots*, MIT Press.
10. Ferrein, A., and G. Lakemeyer. 2008. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems* 56 (11): 980–991.
11. Bensalem, S., and M. Gallien. 2009. Toward a more dependable software architecture for autonomous robots. *IEEE Robotics and Automation Magazine* 1–11.
12. Magnenat, S. 2010. Software integration in mobile robotics, a science to scale up machine intelligence. Ph.D. thesis, École polytechnique fédérale de Lausanne, Switzerland.
13. Vernon, D., C. von Hofsten, and L. Fadiga. 2010. *A Roadmap for Cognitive Development in Humanoid Robots*. Heidelberg: Springer.
14. Balakirsky, S., Z. Kootbally, T. Kramer, A. Pietromartire, C. Schlenoff, and S. Gupta. 2013. Knowledge driven robotics for kitting applications. Volume 61., Elsevier B.V. 1205–1214
15. Björkelund, A., J. Malec, K. Nilsson, P. Nugues, and H. Bruyninckx. 2012. Knowledge for Intelligent Industrial Robots. In *AAAI Spring Symposium on Designing Intelligent Robots: Reintegrating AI*.
16. Stenmark, M., and J. Malec. 2013. Knowledge-based industrial robotics. In *Scandinavian Conference on Artificial Intelligence*.
17. Tenorth, M., and M. Beetz. 2013. KnowRob: A knowledge processing infrastructure for cognition-enabled robots. *The International Journal of Robotics Research* 32 (5): 566–590.
18. Beetz, M., L. Mösenlechner, and M. Tenorth. 2010. CRAM - A Cognitive Robot Abstract Machine for everyday manipulation in human environments. In *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, 1012–1017.
19. Rovida, F., and V. Krüger. 2015. Design and development of a software architecture for autonomous mobile manipulators in industrial environments. In *2015 IEEE International Conference on Industrial Technology (ICIT)*.
20. Huckaby, J. 2014. Knowledge Transfer in Robot Manipulation Tasks. Ph.D. thesis, Georgia Institute of Technology, USA.
21. Bøgh, S., O.S. Nielsen, M.R. Pedersen, V. Krüger, and O. Madsen. 2012. Does your robot have skills? In *The 43rd International Symposium of Robotics (ISR)*.

22. Bechhofer, S., F. van Harmelen, J. Hendler, I. Horrocks, D.L. McGuinness, P.F. Patel-Schneider, and L.A. Stein. 2004. OWL Web Ontology Language reference, 10 Feb 2004. <http://www.w3.org/TR/owl-ref/>.
23. Lortal, G., S. Dhouib, and S. Gérard. 2011. Integrating ontological domain knowledge into a robotic DSL. In *Models in Software Engineering*, ed. J. Dingel, and A. Solberg, 401–414. Heidelberg: Springer.
24. Krüger, V., A. Chazoule, M. Crosby, A. Lasnier, M.R. Pedersen, F. Rovida, L. Nalpantidis, R.P.A. Petrick, C. Toscano, and G. Veiga. 2016. A vertical and cyber-physical integration of cognitive robots in manufacturing. *Proceedings of the IEEE* 104 (5): 1114–1127.
25. Crosby, M., F. Rovida, M. Pedersen, R. Petrick, and V. Krueger. 2016. Planning for robots with skills. In *Planning and Robotics (PlanRob) workshop at the International Conference on Automated Planning and Scheduling (ICAPS)*.
26. Sprunk, C., J. Rowekamper, G. Parent, L. Spinello, G.D. Tipaldi, W. Burgard, and M. Jalobeanu. 2014. An experimental protocol for benchmarking robotic indoor navigation. In *ISER*.
27. Holz, D., and S. Behnke. 2016. Fast edge-based detection and localization of transport boxes and pallets in rgb-d images for mobile robot bin picking. In *Proceedings of the 47th International Symposium on Robotics (ISR), Munich, Germany*.
28. Polydoros, A.S., B. Großmann, F. Rovida, L. Nalpantidis, and V. Krüger. 2016. Accurate and versatile automation of industrial kitting operations with skiros. In *17th Conference Towards Autonomous Robotic Systems (TAROS), (Sheffield, UK)*.
29. Stückler, J., and S. Behnke. 2014. Multi-resolution surfel maps for efficient dense 3D modeling and tracking. *Journal of Visual Communication and Image Representation* 25 (1): 137–147.

Author Biographies

Francesco Rovida is a Ph.D. student at the Robotics, Vision and Machine Intelligence Lab (RVMI), Aalborg University Copenhagen, Denmark. He holds a Bachelor's degree in Computer Science Engineering (2011), and a Master's degree in Robotic Engineering (2013) from the University of Genoa (Italy). He did his Master's thesis at the Istituto Italiano di Tecnologia (IIT, Genoa, Italy) on the development of an active head with motion compensation for the HyQ robot. His research interests include knowledge representation and software integration for the development of autonomous robots.

Matthew Crosby is a Postdoctoral Research Associate currently working at Heriot Watt University on high-level planning for robotics on the EU STAMINA project. His background is in multiagent planning (PhD, Edinburgh) and Mathematics and Philosophy (MSci, Bristol). More details can be found at mdcrosby.com.

Dirk Holz received a diploma in Computer Science from the University of Applied Sciences Cologne in 2006 and a M.Sc. degree in Autonomous Systems from the University of Applied Sciences Bonn-Rhein-Sieg in 2009. He is currently pursuing the Ph.D. degree at the University of Bonn. His research interests include perceiving, extracting and modeling semantic information using 3D sensors as well as simultaneous localization and mapping (SLAM).

Athanasis S. Polydoros received a Diploma in Production Engineering from Democritus University of Thrace in Greece and a M.Sc. degree with Distinction in Artificial Intelligence from the University of Edinburgh, Scotland. He is currently a Ph.D. student at the Robotics, Vision and Machine Intelligence (RVMI) Lab, Aalborg University Copenhagen, Denmark. His research interests are focused on machine learning for robot control and cognition and model learning.

Bjarne Großmann graduated with a dual M.Sc. degree in Computer Science in Media from the University of Applied Sciences Wedel (Germany) in collaboration with the Aalborg University Copenhagen (Denmark) in 2012. He is currently working as a Ph.D. student at the Robotics, Vision and Machine Intelligence (RVMI) Lab in the Aalborg University Copenhagen. The main focus of his work is related to Robot Perception - from Human-Robot-Interaction over 3D object recognition and pose estimation to camera calibration techniques.

Ronald Petrick is a Research Fellow in the School of Informatics at the University of Edinburgh. He received an MMath degree in Computer Science from the University of Waterloo and a PhD in Computer Science from the University of Toronto. His research interests include planning with incomplete information and sensing, cognitive robotics, knowledge representation and reasoning, and applications of planning to human-robot interaction. His recent work has focused on the application of automated planning to task-based action and social interaction on robot platforms deployed in real-world environments. Dr. Petrick has participated in a number of EU-funded research projects under FP6 (PACOPLUS) and FP7 (XPERIENCE and STAMINA). He was also the Scientific Coordinator of the FP7 JAMES project.

Volker Krüger is a Professor at Aalborg University, Denmark where he has worked since 2002. His teaching and research interests are in the area of cognitive robotics for manufacturing and industrial automation. Since 2007, he has headed the Robotics, Vision and Machine Intelligence group (RVMI). Dr. Krueger has participated in a number of EU-funded research projects under FP4, FP5, and FP6, coordinated the FP7 project GISA (under ECHORD), and participated in the EU projects TAPAS, PACO-PLUS, and CARLoS. He is presently coordinating the FP7 project STAMINA. Dr. Krueger has recently completed an executive education at Harvard Business School related to academic industrial collaborations and knowledge-exchange.

Control of Mobile Robots Using ActionLib

**Higor Barbosa Santos, Marco Antônio Simões Teixeira,
André Schneider de Oliveira, Lúcia Valéria Ramos de Arruda
and Flávio Neves Jr.**

Abstract Mobile robots are very complex systems and involve the integration of various structures (mechanical, software and electronics). The robot control system must integrate these structures so that it can perform its tasks properly. Mobile robots use control strategies for many reasons, like velocity control of wheels, position control and path tracking. These controllers require the use of preemptive structures. Therefore, this tutorial chapter aims to clarify the design of controllers for mobile robots based on ROS ActionLib. Each controller is designed in an individual ROS node to allow parallel processing by operating system. To exemplify the controller design using ActionLib, this chapter will demonstrate the implementation of two different types of controllers (PID and Fuzzy) for position control of a servo motor. These controllers will be available on GitHub. Also, a case study of scheduled fuzzy controllers based on ROS ActionLib for a magnetic climber robot used in the inspection of spherical tanks will be shown.

Keywords ROS · Mobile robots · Control · ActionLib

1 Introduction

Mobile robots have great versatility because they're free to run around their application environment. However, this is only possible because this kind of robot carries a great variety of exteroceptive and interoceptive sensors to measure its motion and

H.B. Santos (✉) · M.A.S. Teixeira · A.S. de Oliveira · L.V.R. de Arruda · F. Neves Jr.
Federal University of Technology—Parana, Av. Sete de Setembro, 3165 Curitiba, Brazil
e-mail: higorsantos@alunos.utfpr.edu.br

M.A.S. Teixeira
e-mail: marcoteixeira@alunos.utfpr.edu.br

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

L.V.R. de Arruda
e-mail: lvrarruda@utfpr.edu.br

F. Neves Jr.
e-mail: neves@utfpr.edu.br

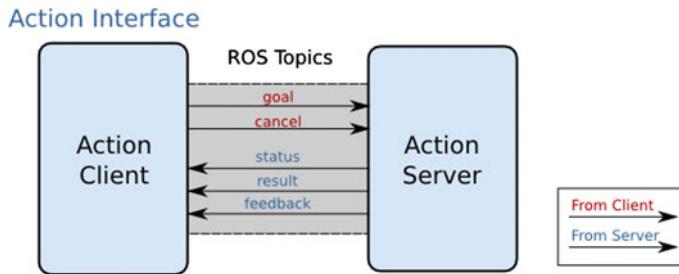


Fig. 1 Interface of ActionLib. *Source [1]*

interact with environment around it. Several of these information are used to robot's odometry or environment mapping. Thus, these signals are the robot's sense about its motion and its way to correct it.

Robot control is a complex and essential task which must be performed during all navigation. Several kinds of controllers can be applied (like proportional-integral-derivative, predictive, robust, adaptive and fuzzy). The implementation of the robot control is very similar and can be developed with the use of ROS Action Protocol by ActionLib. ActionLib provides a structure to create servers that execute long-running tasks and interacts with clients by specific messages [1].

The proposed chapter aims to explain how to create ROS controllers using the ActionLib structure. This chapter is structured in five sections.

In the first section, we will carefully discuss the ActionLib structure. This section introduces the development of preemptive tasks with ROS. The ActionLib works with three main messages, as can be seen in Fig. 1. Goal message is the desired value for controller, like its target or objective. Feedback message is the measure of controlled variable which usually is updated by means of a robot sensor. Result message is a flag that indicates when the controller reaches its goal.

The second section will demonstrate the initial requirements for the creation of a controller using ActionLib package.

The third section will present an implementation of a classic Proportional-Derivative-Integrative (PID) control strategy with the use of ActionLib. This section aims to introduce a simple (but powerful) control structure that can be applied to many different purposes, like position control, velocity control, flight control, adhesion control and among others. It'll be shown how to create your own package, set the action message, structure the server/client code, compile the created package and, finally, show the experimental results of the PID controller.

In the fourth section will be present a design of a fuzzy controller. The controller is implemented using ActionLib and an open-source fuzzy library. The fuzzy logic enables the implementation of a control without the knowledge of the system dynamics model.

Finally, last section will show a study case of an ActionLib based control for the second-generation of a climbing robot with four steerable magnetic wheels [2], called

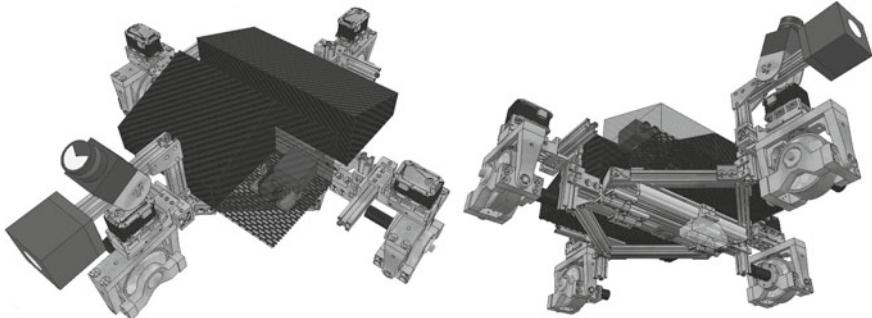


Fig. 2 Autonomous inspection robot (AIR-2)

as Autonomous Inspection Robot 2nd generation (AIR-2), as shown in Fig. 2. AIR-2 is a robot fully compatible with ROS and it has a mechanical structure designed to provide high mobility when climbs on industrial storage tanks. The scheduled fuzzy controllers were designed to manage the speed of AIR-2.

2 ActionLib

Mobile robots are very complex, they have many sensors and actuators that help them get around and locate in an unknown environment. The control of the robot isn't an easy task and it should have a parallel processing. The mobile robot must handle several tasks at same time, so the preemption is an important feature to robot control. Often, the robot control is multivariable, that's means multiple-input multiple-output systems (MIMO). Therefore, developing an algorithm with these characteristics is hard.

Robot control covers various functions of a robot. For example, obstacle avoidance is very important for the autonomous mobile robots. Therefore, [3] proposed a fuzzy intelligent obstacle avoidance controller for a wheeled mobile robot. Balancing robot is another relevant aspect in robotics, said that, [4] designed a cascaded PID controller for movement control of a two wheel robot. On the other hand, [5] presented an adhesion force control for a magnetic climbing robot used in the inspection of storage tanks. Therefore, the robot control is essential to ensure its operation, either navigation or obstacle avoidance.

The ROS has libraries that help in the implementation of control, like *ros_control*. Other library is ActionLib that enables to create servers to execute long-running tasks and clients that interact with servers. Given these features, the development of a controller using this library becomes easy. On the other hand, *ros_control* package is hard to be used, it presents a control structure that requires many configurations for implementation of a specific controller, for example, a fuzzy controller.

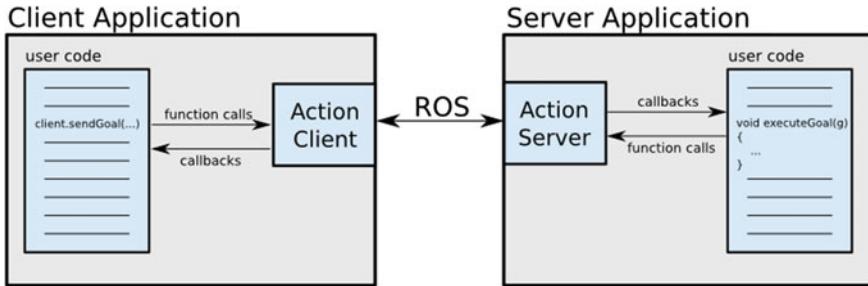


Fig. 3 Client-server interaction. *Source* [1]

The Actionlib provides a simple application for client sends goals and server executes goals. The server executes long-running goals that can be preempted. Client-server interaction using ROS Action Protocol is shown in Fig. 3.

The client-server interaction in ActionLib is provided by messages that are displayed in Fig. 1. The messages are:

- **goal**: client sends goal to the server;
- **cancel**: client sends the cancellation of the goal to the server;
- **status**: server notifies the status of the goal for the client;
- **feedback**: server sends goal information to the client;
- **result**: server notifies the client when the goal was achieved.

Thus, the ActionLib package is a powerful tool for the design of controllers in robotics. In the next section, the initial configuration of ROS Workspace will be presented for the use of ActionLib package.

3 ROS Workspace Configuration

For implementation of the controller it's necessary that ROS Indigo is properly installed. It's available at:

<http://wiki.ros.org/indigo/Installation/Ubuntu>

The next step is the ROS Workspace configuration. If it isn't configured on your machine, it'll be necessary create it:

```

1 $ mkdir -p /home/user/catkin_ws/src
2 $ cd /home/user/catkin_ws/src
3 $ catkin_init_workspace

```

The *catkin_init_workspace* command sets the *catkin_ws* folder as your workspace. After, you must build the workspace. For this, you need to navigate to your workspace folder and then type the command *catkin_make*, as shown below.

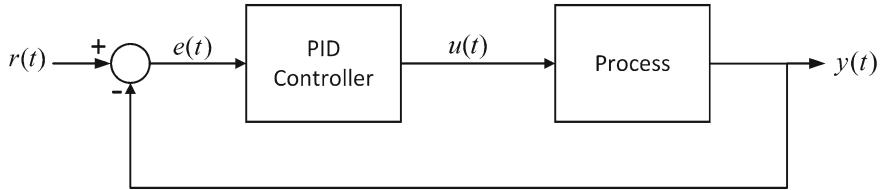


Fig. 4 PID control

```

1 $ cd /home/user/catkin_ws/
2 $ catkin_make
  
```

To add the workspace to your ROS environment, you need to source the generated setup file:

```

1 $ source /home/user/catkin_ws/devel/setup.bash
  
```

After workspace configuration, we can start creating a PID controller using ActionLib, which it'll be shown in the next section.

4 Creating a PID Controller Using ActionLib

There're various types of algorithm used to robot control. But the PID control is more used, due to its good performance for linearized systems and easy implementation. The PID controller is a control loop feedback widely used in various applications, in Fig. 4 can be seen its diagram. The Eq. 1 shows the PID equation:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{d}{dt} e(t) \quad (1)$$

$$e(t) = r(t) - y(t) \quad (2)$$

where $u(t)$ is output, K_p is proportional gain, $e(t)$ is error (difference between set-point $r(t)$ and process output $y(t)$, as shown in Eq. 2), K_i is integral gain and K_d is derivative gain. The controller calculates the error and by adjusting the gains (K_p , K_i and K_d), the PID seeks to minimize it.

In this section, it'll be shown the PID control implementation using ActionLib. The PID will control the angle of a servo motor. The servo motor was simulated in robot simulator V-REP. The V-REP is simulator based on distributed control architecture, it allows the modeling of robotic systems similar to the reality [6].

The controller has been implemented in accordance with Fig. 4, in which the setpoint is the desired angle (goal) and the feedback is provided by encoder servo.

The PID controller is available on GitHub and can be installed on your workspace:

```

1 $ source /opt/ros/indigo/setup.bash
2 $ cd /home/user/catkin_ws/src
3 $ git clone https://github.com/air-lasca/tutorial_controller
4 $ cd ..
5 $ catkin_make
6 $ source /home/user/catkin_ws/devel/setup.bash

```

The following will be shown its creation step-by-step.

4.1 Steps to Create the Controller

1st step: Creating the ActionLib package Once the workspace is created and configured, let's create package using ActionLib:

```

1 $ cd /home/user/catkin_ws/src/
2 $ catkin_create_pkg tutorial_controller actionlib
    message_generation roscpp rospy std_msgs actionlib_msgs

```

The `catkin_create_package` command creates a package named `tutorial_controller` which depends on `actionlib`, `message_generation`, `roscpp`, `rospy`, `std_msgs` and `actionlib_msgs`. Posteriorly, if you need other dependencies just add them in `CMakeList.txt`. This will be detailed in fifth step.

After creating the package, we need to define the message that is sent between the server and client.

2nd step: Creating the action messages Continuing steps to create the controller, you must set the action messages. The action file has three parts: goal, result and feedback. Each section of action file is separated by 3 hyphens (---).

The goal message is the setpoint of controller, it is sent from the client to the server. Yet, the result message is sent from the server to the client, it tells us when the server completed the goal. It would be a flag to indicate that the controller has reached the goal, but for the control has no purpose. While, the feedback message is sent by the server to inform the goal of incremental progress for the client. Feedback would be the information from the sensor used in control.

Then, to create the action messages, you must create a folder called `action` in your package.

```

1 $ cd /home/user/catkin_ws/src/tutorial_controller
2 $ mkdir action

```

After creating the folder, you must create an `.action` file (`Tutorial.action`) in action's folder of your package. The first letter of the action name should be upper-case. This information is placed in the `.action` file:

```

1 #Define the goal
2 float64 position
3 ---

```

```

4 #Define the result
5 bool ok
6 ---
7 #Define a feedback message
8 float64 position

```

The goal and feedback are defined as *float64*. The goal will receive the desired position of a servo motor and feedback will be used to send the information acquired from the encoder servo. The result is defined as *bool*, but it won't be used in control. This action file will be used in the controllers examples shown in this chapter.

The action messages are generated automatically from the .action file.

3rd step: Create the action client In src folder of your package, create *ControllerClient.cpp*, it'll be client of ActionLib. Firstly, we'll include the necessary libraries of ROS, action message and action client.

```

1 #include <ros/ros.h>
2 #include <tutorial_controller/TutorialAction.h>
3 #include <actionlib/client/simple_action_client.h>
4 #include "std_msgs/Float64.h"

```

The *tutorial_controller/TutorialAction.h* is the action message library. It'll access the messages created in the .action file.

The *actionlib/server/simple_action_client.h* is the action library used from implementing simple action client. If necessary, you can include other libraries.

Continuing the code, the client class must be set. The action client constructor defines the topic to publish the messages. So, you need specific the same topic name of your server, in this example, *pid_control* was used.

```

6 class ControllerClient
7 {
8     public:
9         ControllerClient(std::string name):
10             //Set up the client. It's publishing to topic "
11             //      pid_control", and is set to auto-spin
12             ac("pid_control", true),
13
14             //Stores the name
15             action_name(name)
16         {
17             //Get connection to a server
18             ROS_INFO("%s Waiting For Server...", action_name.c_str());
19
20             //Wait for the connection to be valid
21             ac.waitForServer();
22
23             ROS_INFO("%s Got a Server...", action_name.c_str());
24
25             goalsub = n.subscribe("/cmd_pos", 100, &ControllerClient::
26             GoalCallback, this);
27     }

```

The *ac.waitForServer()* causes the client waits for the server to start before continuing. Once the server has started, the client informs that established communication with it. Then, define a subscriber (*goalsub*) to provide the setpoint of the control.

The *doneCb* function is called every time that goal completes. It provides state of action server and result message. It'll only be called if the server has not preempted, because the controller must run continuously.

```
28 void doneCb(const actionlib::SimpleClientGoalState& state,
29   const tutorial_controller::TutorialResultConstPtr& result){
30   ROS_INFO("Finished in state [%s]", state.toString().c_str());
31   ROS_INFO("Result: %i", result->ok);
32 }
```

The *activeCb* is called every time the goal message is active, in other words, it's called each new goal received by client.

```
33 void activeCb(){
34   ROS_INFO("Goal just went active...");
35 }
```

The *feedbackCb* is called every time the server sends the feedback message to the action client.

```
38 void feedbackCb(const tutorial_controller::
39   TutorialFeedbackConstPtr& feedback){
40   ROS_INFO("Got Feedback of Progress to Goal: position: %f",
41   feedback->position);
42 }
```

GoalCallback is a function that transmits the goal of the topic */cmd_goal* to the action server.

```
42 void GoalCallback(const std_msgs::Float64& msg){
43   goal.position = msg.data;
44
45   ac.sendGoal(goal, boost::bind(&ControllerClient::doneCb, this
46     , _1, _2),
47   boost::bind(&ControllerClient::activeCb, this),
48   boost::bind(&ControllerClient::feedbackCb, this, _1));
49 }
```

The private variables of action client: *n* is a NodeHandle, *ac* is an action client object, *action_name* is a string to set the client name, *goal* is the message that is used to publish goal to server (set in .action file) and *goalsub* is a subscriber to get the goal and pass to the action server.

```
50 private:
51   actionlib::SimpleActionClient<tutorial_controller::
52     TutorialAction> ac;
53   std::string action_name;
54   tutorial_controller::TutorialGoal goal;
55   ros::Subscriber goalsub;
```

```
55     ros::NodeHandle n;
56 }
```

Begin action client:

```
58 int main (int argc, char **argv){
59     ros::init(argc, argv, "pid_client");
60
61     // create the action client
62     // true causes the client to spin its own thread
63     ControllerClient client(ros::this_node::getName());
64
65     ros::spin();
66
67     //exit
68     return 0;
69 }
```

4th step: Create the action server Therefore, the action client is finished. Now, create action server named *ControllerServer.cpp* in your src folder. Initially, it's necessary include the libraries of ROS, action message and action server. Procedure similar will be made at the client.

```
1 #include <ros/ros.h>
2 #include <tutorial_controller/TutorialAction.h>
3 #include <actionlib/server/simple_action_server.h>
4 #include "std_msgs/Float64.h"
5 #include "geometry_msgs/Vector3.h"
6 #include "sensor_msgs/JointState.h"
7 #include <math.h>
```

So, we need to set the server class. The action server constructor starts the server. Also, it defines subscriber (feedback loop's control), publisher (PID output), PID limits and initiates the control variables.

```
9 class ControllerServer{
10 public:
11     ControllerServer(std::string name):
12         as(n, "pid_control", boost::bind(&ControllerServer::
13             executeCB, this, _1), false),
14         action_name(name)
15     {
16         as.registerPreemptCallback(boost::bind(&ControllerServer
17             ::preemptCB, this));
18
19         //Start the server
20         as.start();
21
22         //Subscriber current positon of servo
23         sensorsub = n2.subscribe("/sensor/encoder/servo", 1, &
24             ControllerServer::SensorCallBack, this);
25
26         //Publisher setpoint, current position and error of
27         //control
```

```

24     error_controlpub = n2.advertise<geometry_msgs::Vector3>("control/error", 1);
25
26     //Publisher PID output in servo
27     controlpub = n2.advertise<std_msgs::Float64>("/motor/servo", 1);
28
29     //Max e Min Output PID Controller
30     float max = M_PI;
31     float min = -M_PI;
32
33     //Initializing PID Controller
34     Initialize(min,max);
35 }
```

In the action constructor, an action server is created. A sensor subscriber (*sensor_sub*) and a controller output publisher (*controlpub*) are created to the control loop.

The *preemptCB* informs that the current goal has been canceled by sending a new goal or action client canceled the request.

```

37 void preemptCB(){
38     ROS_INFO("%s got preempted!", action_name.c_str());
39     result.ok = 0;
40     as.setPreempted(result, "I got Preempted!");
41 }
```

A pointer to the goal message is passed in *executeCB* function. This function defines the rate of the controller. You can set the frequency in the *rate* argument of your control. Inside the *while*, the function of the controller should be called, as shown in line 60. It's passed to the controller setpoint (*goal->position*) and the feedback sensor (*position_encoder*).

```

43 void executeCB(const tutorial_controller::TutorialGoalConstPtr& goal){
44     prevTime = ros::Time::now();
45
46     //If the server has been killed, don't process
47     if(!as.isActive()||as.isPreemptRequested()) return;
48
49     //Run the processing at 100Hz
50     ros::Rate rate(100);
51
52     //Setup some local variables
53     bool success = true;
54
55     //Loop control
56     while(1){
57         std_msgs::Float64 msg_pos;
58
59         //PID Controller
60         msg_pos.data = PIDController(goal->position,
61             position_encoder);
```

```

63 //Publishing PID output in servo
64 controlpub.publish(msg_pos);
65
66 //Auxiliary Message
67 geometry_msgs::Vector3 msg_error;
68
69 msg_error.x = goal->position;
70 msg_error.y = position_encoder;
71 msg_error.z = goal->position - position_encoder;
72
73 //Publishing setpoint, feedback and error control
74 error_controlpub.publish(msg_error);
75
76 feedback.position = position_encoder;
77
78 //Publish feedback to action client
79 as.publishFeedback(feedback);
80
81 //Check for ROS kill
82 if(!ros::ok()){
83     success = false;
84     ROS_INFO("%s Shutting Down", action_name.c_str());
85     break;
86 }
87
88 //If the server has been killed/preempted, stop processing
89 if(!as.isActive()||as.isPreemptRequested()) return;
90
91 //Sleep for rate time
92 rate.sleep();
93 }

94
95 //Publish the result if the goal wasn't preempted
96 if(success){
97     result.ok = 1;
98     as.setSucceeded(result);
99 }
100 else{
101     result.ok = 0;
102     as.setAborted(result,"I Failed!");
103 }
104 }
```

Initialize is a function that sets the initial parameters of a controller. It defines the PID controller output limits and gains.

```

105 void Initialize( float min, float max){
106     setOutputLimits(min, max);
107     lastError = 0;
108     errSum = 0;
109
110     kp = 1.5;
111     ki = 0.1;
112     kd = 0;
```

113 }

The *setOutputLimits* function sets the control limits.

```
115 void setOutputLimits(float min, float max){
116     if (min > max) return;
117     minLimit = min;
118     maxLimit = max;
119 }
```

The *Controller* function implements the PID equation (Eq. 1). It can be used to design your control algorithm.

```
121 float PIDController(float setpoint, float PV) {
122     ros::Time now = ros::Time::now();
123     ros::Duration change = now - prevTime;
124
125     float error = setpoint - PV;
126
127     errSum += error*change.toSec();
128     errSum = std::min(errSum, maxLimit);
129     errSum = std::max(errSum, minLimit);
130
131     float dErr = (error - lastError)/change.toSec();
132
133     //Do the full calculation
134     float output = (kp*error) + (ki*errSum) + (kd*dErr);
135
136     //Clamp output to bounds
137     output = std::min(output, maxLimit);
138     output = std::max(output, minLimit);
139
140     //Required values for next round
141     lastError = error;
142     prevTime = now;
143
144     return output;
145 }
```

Sensor callback is a subscriber that provides sensor information, e.g., position or wheel velocity of a robot. In this case, it receives the position of the servo motor encoder.

```
148 void SensorCallBack(const sensor_msgs::JointState& msg){
149     position_encoder = msg.position[0];
150 }
```

The protected variables of action server:

```
152 protected:
153     ros::NodeHandle n;
154     ros::NodeHandle n2;
155
156     //Subscriber
```

```

157 ros::Subscriber sensorsub;
158
159 //Publishers
160 ros::Publisher controlpub;
161 ros::Publisher error_controlpub;
162
163 //Actionlib variables
164 actionlib::SimpleActionServer<tutorial_controller::TutorialAction> as;
165 tutorial_controller::TutorialFeedback feedback;
166 tutorial_controller::TutorialResult result;
167 std::string action_name;
168
169 //Control variables
170 float position_encoder;
171 float errSum;
172 float lastError;
173 float minLimit, maxLimit;
174 ros::Time prevTime;
175 float kp;
176 float ki;
177 float kd;
178 };

```

Finally, the main function creates the action server and spins the node. The action will be running and waiting to receive goals.

```

180 int main(int argc, char** argv){
181   ros::init(argc, argv, "pid_server");
182
183   //Just a check to make sure the usage was correct
184   if(argc != 1){
185     ROS_INFO("Usage: pid_server");
186     return 1;
187   }
188
189   //Spawn the server
190   ControllerServer server(ros::this_node::getName());
191
192   ros::spin();
193
194   return 0;
195 }

```

5th step: Compile the created package To compile your controller, it'll be need to add a few things to *CMakeLists.txt*. Firstly, you need to specify the necessary libraries to compile the package. If you require any other library that wasn't mentioned when creating your package, you can add it to *find_package()* and *catkin_package()*.

```

1 find_package(catkin REQUIRED COMPONENTS
2   actionlib
3   actionlib_msgs
4   message_generation

```

```

5  roscpp
6  rospy
7  std_msgs
8 )
9
10 find_package(
11   CATKIN_DEPENDS actionlib actionlib_msgs message_generation
12     roscpp rospy std_msgs
12 )

```

Then, specify the action file to generate the messages.

```

1 add_action_files(
2   DIRECTORY action
3   FILES Tutorial.action
4 )

```

And specify the libraries that need .action.

```

1 generate_messages(
2   DEPENDENCIES
3   actionlib_msgs std_msgs
4 )

```

Include directories that your package needs.

```

1 include_directories(${catkin_INCLUDE_DIRS})

```

The *add_executable()* creates the executable of your server and client. The *target_link_libraries()* includes libraries that can be used by action server and client at build and/or execution. The macro *add_dependencies()* creates a dependency between the messages generated by the server and client with your executables.

```

1 add_executable(TutorialServer src/ControllerServer.cpp)
2 target_link_libraries(TutorialServer ${catkin_LIBRARIES})
3 add_dependencies(TutorialServer ${
4   tutorial_controller_EXPORTED_TARGETS} ${
5   catkin_EXPORTED_TARGETS})
6
7 add_executable(TutorialClient src/ControllerClient.cpp)
8 target_link_libraries(TutorialClient ${catkin_LIBRARIES})
9 add_dependencies(TutorialClient ${
10   tutorial_controller_EXPORTED_TARGETS} ${
11   catkin_EXPORTED_TARGETS})

```

Additionally, the *package.xml* file must include the following dependencies:

```

1 <build_depend>actionlib</build_depend>
2 <build_depend>actionlib_msgs</build_depend>
3 <build_depend>message_generation</build_depend>
4 <run_depend>actionlib</run_depend>
5 <run_depend>actionlib_msgs</run_depend>
6 <run_depend>message_generation</run_depend>

```

```
robo@robo:~$ rosrun tutorial_controller TutorialServer
```

Fig. 5 PID controller server

```
robo@robo:~$ rosrun tutorial_controller TutorialClient
[ INFO] [1464107346.671790163]: /pid_client Waiting For Server...
[ INFO] [1464107346.953621521]: /pid_client Got a Server...
```

Fig. 6 PID controller client

Now, just compile your workspace:

```
1 $ cd /home/user/catkin_ws/
2 $ catkin_make
```

And refresh your ROS environment:

```
1 $ source /home/user/catkin_ws/devel/setup.bash
```

So your PID controller is ready to be used.

6th step: Run the controller Once compiled, your package is ready for use. To run your package, open terminal and start ROS:

```
1 $ roscore
```

After the ROS starts, you must start the server in new terminal, as shown in Fig. 5.

So, in a new terminal, start the client, as demonstrated in Fig. 6.

PID Controller client waits for the server and notifies you when the connection is established between them.

An alternative to *rosrun* is *roslaunch*. To use *roslaunch* command, you need to create a folder named *launch* in your package.

```
1 $ cd /home/user/catkin_ws/src/tutorial_controller
2 $ mkdir launch
```

After creating the folder, create a launch file (*tutorial.launch*) in the directory *launch* of your package. In the launch file put the following commands:

```
1 <launch>
2   <node pkg="tutorial_controller" type="TutorialServer" name="TutorialServer"
      output="screen"/>
3   <node pkg="tutorial_controller" type="TutorialClient" name="TutorialClient"
      output="screen"/>
4 </launch>
```

To roslaunch works, it's necessary add roslaunch package in *find_package()* of *CMakeLists.txt*:

```

1 find_package(
2   catkin REQUIRED
3   COMPONENTS actionlib actionlib_msgs roslaunch
4 )

```

And add below line of the *find_package()* in *CMakeLists.txt*:

```

1 roslaunch_add_file_check(launch)

```

Thus, *CMakeLists.txt* would look like this:

```

1 cmake_minimum_required(VERSION 2.8.3)
2 project(tutorial_controller)
3
4 find_package(catkin REQUIRED COMPONENTS
5   actionlib
6   actionlib_msgs
7   message_generation
8   roscpp
9   rospy
10  std_msgs
11  roslaunch
12 )
13
14 roslaunch_add_file_check(launch)
15
16 add_action_files(
17   DIRECTORY action
18   FILES Tutorial.action
19 )
20
21 generate_messages(
22   DEPENDENCIES
23   actionlib_msgs std_msgs
24 )
25
26 catkin_package(
27   CATKIN_DEPENDS actionlib actionlib_msgs message_generation
28   roscpp rospy std_msgs
29 )
30
31 include_directories(${catkin_INCLUDE_DIRS})
32
33 add_executable(TutorialServer src/ControllerServer.cpp)
34 target_link_libraries(TutorialServer ${catkin_LIBRARIES})
35 add_dependencies(TutorialServer ${
36   tutorial_controller_EXPORTED_TARGETS} ${
37   catkin_EXPORTED_TARGETS})
38
39 add_executable(TutorialClient src/ControllerClient.cpp)
40 target_link_libraries(TutorialClient ${catkin_LIBRARIES})

```

```
38 add_dependencies(TutorialClient ${  
    tutorial_controller_EXPORTED_TARGETS} ${  
    catkin_EXPORTED_TARGETS})
```

Save the CMakeLists file. So, you can compile the workspace:

```
1 $ cd /home/user/catkin_ws/  
2 $ catkin_make
```

Don't forget to add the workspace to your ROS environment:

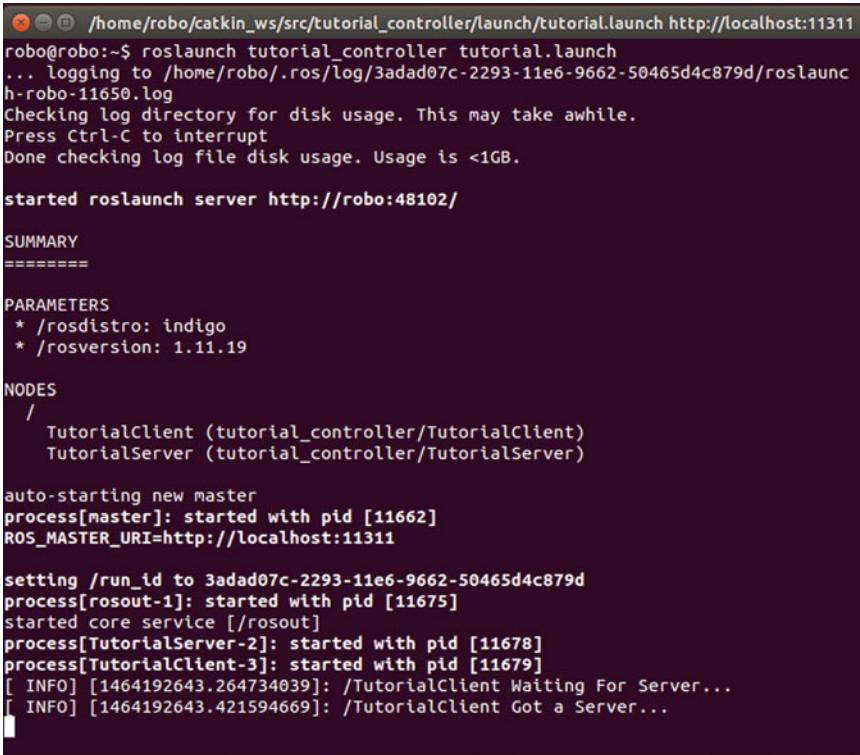
```
1 $ source catkin_ws/devel/setup.bash
```

Now, to run your package, you just need this command:

```
1 $ roslaunch tutorial_controller tutorial.launch
```

Please note that *roslaunch* starts ROS automatically, as shown in Fig. 7. Therefore, the *roscore* command isn't required before running the controller.

With *rqt_graph* command, you can see all the topics published and interaction between the nodes in ROS.



```
/home/robo/catkin_ws/src/tutorial_controller/launch/tutorial.launch http://localhost:11311
robo@robo:~$ roslaunch tutorial_controller tutorial.launch
... logging to /home/robo/.ros/log/3adad07c-2293-11e6-9662-50465d4c879d/roslaun
h-robo-11650.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://robo:48102/

SUMMARY
=====
PARAMETERS
  * /rosdistro: indigo
  * /rosversion: 1.11.19

NODES
/
  TutorialClient (tutorial_controller/TutorialClient)
  TutorialServer (tutorial_controller/TutorialServer)

auto-starting new master
process[master]: started with pid [11662]
ROS_MASTER_URI=http://localhost:11311

setting /run_id to 3adad07c-2293-11e6-9662-50465d4c879d
process[rosout-1]: started with pid [11675]
started core service [/rosout]
process[TutorialServer-2]: started with pid [11678]
process[TutorialClient-3]: started with pid [11679]
[ INFO] [1464192643.264734039]: /TutorialClient Waiting For Server...
[ INFO] [1464192643.421594669]: /TutorialClient Got a Server...
```

Fig. 7 Roslaunch running PID controller

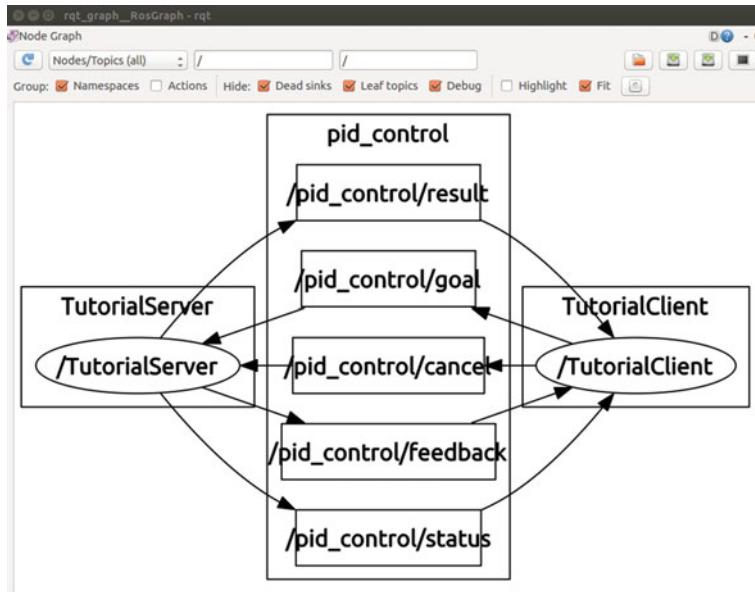


Fig. 8 Interaction between nodes of PID controller

```
$ rqt_graph
```

The `rqt_graph` package provides a GUI plugin for visualizing the ROS computation graph [7]. You can visualize the topics used for communication between the client and server. Exchanging messages between the client and server are shown in Fig. 8.

The `rqt_plot` package provides a GUI plugin to plot 2D graphics of the ROS topics [8]. Open new terminal and enter the following command:

```
rqt_plot /control/error/x /control/error/y
```

The variable x is the controller setpoint and y is feedbacked signal (sensor information). In Fig. 9 is shown the `rqt_plot` plugin.

In case you need, the variable z is the controller error and it can be added in the `rqt_plot`. Just add the topic `/control/error/z` in `rqt_plot` via command:

```
rqt_plot /control/error/x /control/error/y /control/error/z
```

Or add the error topic directly in the `rqt_plot` GUI. Just enter the desired topic, as in the Fig. 10, and click on the $+$ symbol so that it add more information to plot.

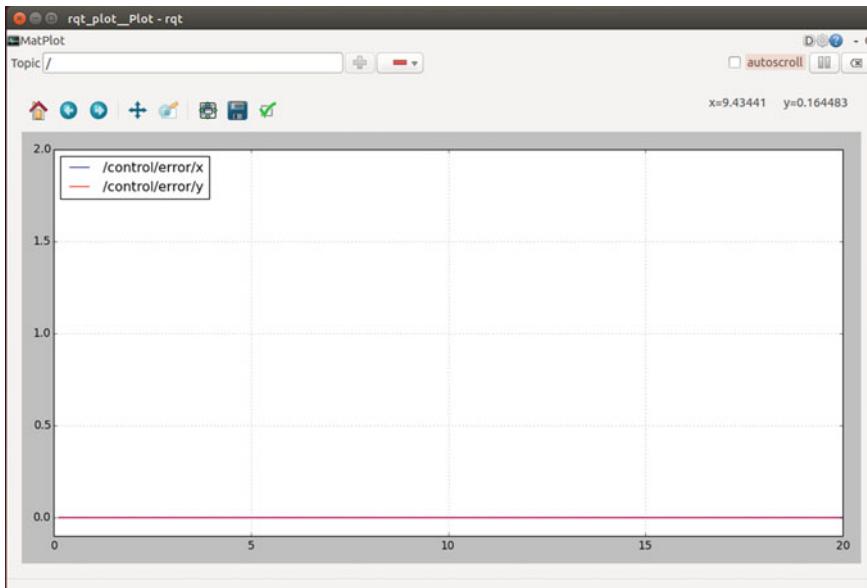


Fig. 9 rqt_plot

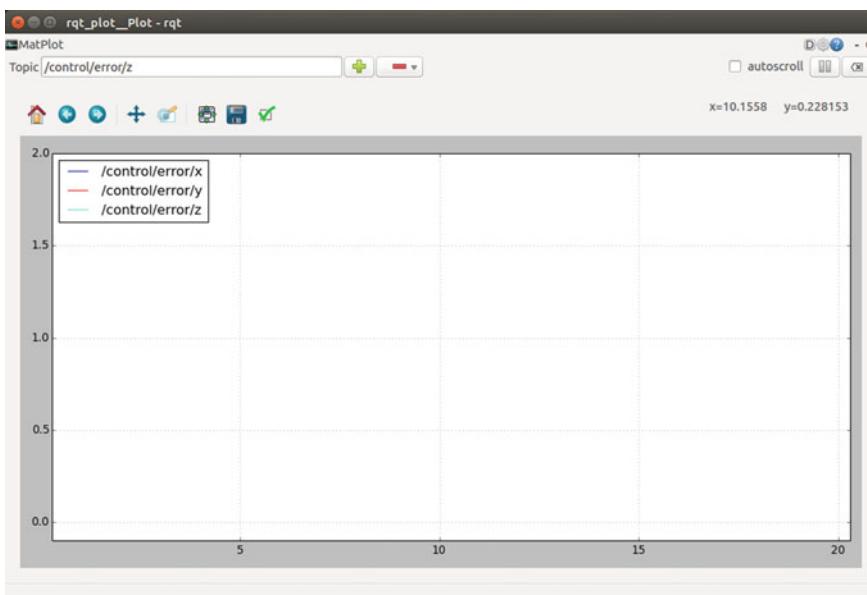


Fig. 10 Add the error topic in rqt_plot

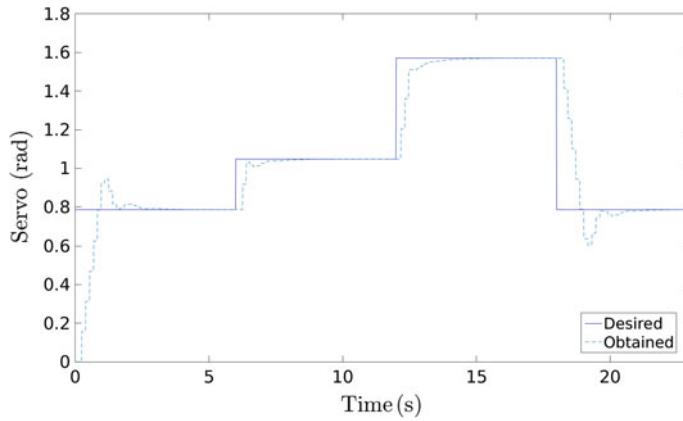


Fig. 11 PID control servo’s position

4.2 Experimental Result of PID Controller

For the controller validation, 4 different setpoints were sent to the controller.

The Fig. 11 presents the results of servo position PID control using ActionLib, the control shows a good response.

5 Creating a Fuzzy Controller Using ActionLib

In this section, the implementation of a Fuzzy control using ActionLib will be shown. The fuzzylite library was used to design the fuzzy logic control. The fuzzylite is a free and open-source fuzzy library programmed in C++ for multiple platforms (Windows, Linux, Mac, iOS, Android) [9].

QtFuzzyLite 4 is a graphic user interface for fuzzylite library, you can implement your fuzzy controller using this GUI. Its goal is to accelerate the implementation process of fuzzy logic controllers by providing a graphical user interface very useful and functional allowing you to easily create and directly interact with your controllers [9]. This GUI is available at:

<http://www.fuzzylite.com/download/fuzzylite4-linux/>.

In the Fig. 12 can be seen the graphic user interface, QtFuzzyLite 4.

In QtFuzzyLite, you can then export the C ++ code to your controller, as shown in Fig. 13.

The Fuzzy controller uses the same application example used in the PID control. In Fig. 14 the fuzzy control diagram of position servo motor can be seen, where β is setpoint, β_s is encoder servo information and u is fuzzy output.

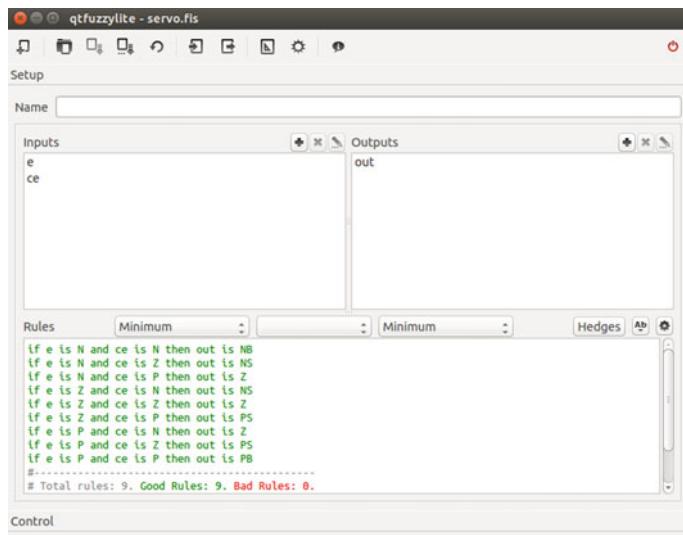


Fig. 12 QtFuzzyLite 4 GUI

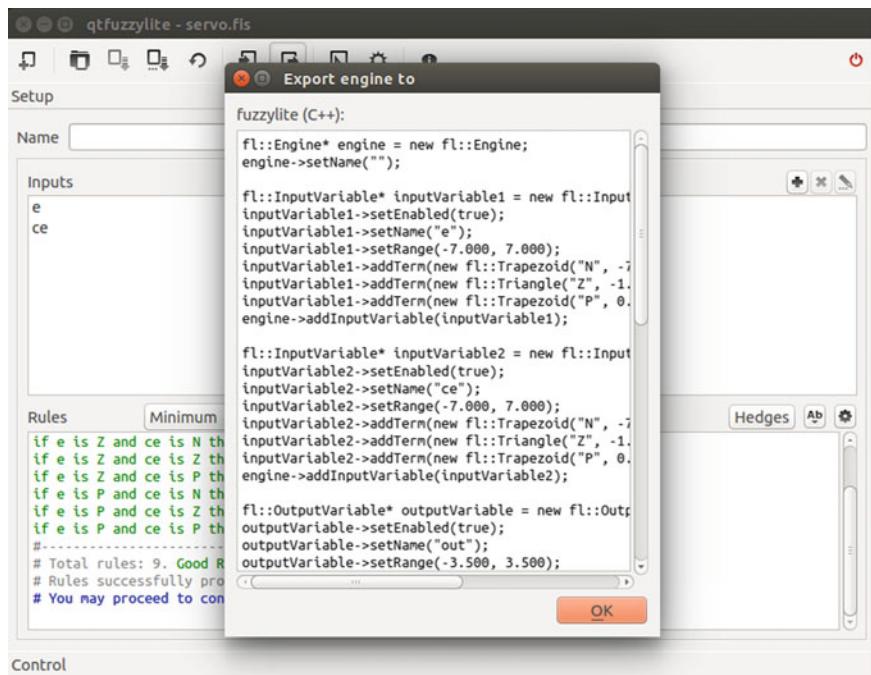


Fig. 13 Export fuzzy to C++ in QtFuzzyLite 4

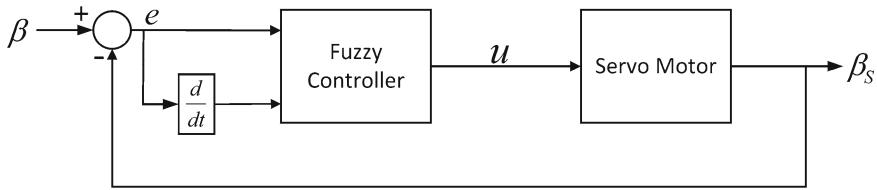


Fig. 14 Fuzzy controller for a servo motor

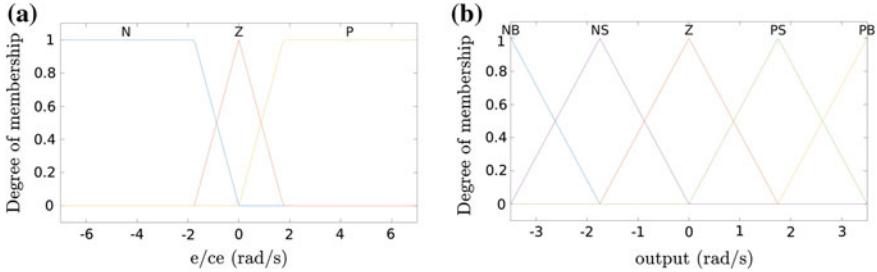


Fig. 15 Membership functions for the servo fuzzy controller of **a** error and change of error and **b** angle increment

The Servo Fuzzy Controller designed for the linear and orientation motion control is presented in Fig. 15. The inputs are ‘e’ (angle error) and ‘ce’ (angle change of error), and the output u is angle increment ($u[k] = u[k] + u[k - 1]$).

The rules for fuzzy controller are shown in Table 1.

The Fuzzy controller is available on GitHub and can be installed on your workspace:

```

1 $ source /opt/ros/indigo/setup.bash
2 $ cd /home/user/catkin_ws/src
3 $ git clone https://github.com/air-lasca/tutorial2_controller
  
```

Table 1 Rule table

e ce	N	Z	P
N	NB	NS	Z
Z	NS	Z	PS
P	Z	PS	PB

5.1 Steps to Create the Controller

The creation of Fuzzy controller follows the same steps of the PID. But it has some peculiarities that will be presented below.

1st step: Creating the ActionLib package Create the package:

```
1 $ cd /home/user/catkin_ws/src/
2 $ catkin_create_pkg tutorial2_controller actionlib
    message_generation roscpp rospy std_msgs actionlib_msgs
```

2nd step: Creating the action messages The action file will be the same used by PID controller, but the action's name will be different: *FuzzyControl.action*.

3rd step: Create action client The structure of the client will be the same PID client, it will be changed the client's name (*FuzzyClient.cpp*), action (*FuzzyControl.action*), topic (*fuzzy_control*) and package (*tutorial2_controller*).

4th step: Create the action server The server will be named *FuzzyServer.cpp*, but you will need to include the fuzzylite library.

```
1 #include <ros/ros.h>
2 #include <tutorial2_controller/FuzzyControlAction.h>
3 #include <actionlib/server/simple_action_server.h>
4
5 #include "std_msgs/Float64.h"
6 #include "geometry_msgs/Vector3.h"
7 #include "sensor_msgs/JointState.h"
8 #include <math.h>
9
10 //Fuzzylite library
11 #include "fl/Headers.h"
12 using namespace fl;
```

The changes mentioned in creating the Fuzzy client should also be made. And the control algorithm will also change, just copy it in the *FuzzyServer.cpp* available on GitHub.

5th step: Compile the created package Before you compile the fuzzy controller package, you must copy the *libfuzzylite* library to your */usr/lib/*. Then, add the fuzzylite's source files (*fl* folder) in the *include* directory of your package. The *libfuzzylite.so* file and *fl* folder can be downloaded from the link:

https://github.com/air-lasca/tutorial2_controller.

The *CMakeLists.txt* and *package.xml* follow the instructions specified in the PID controller. Only, in the *CMakeLists.txt*, you'll need to add the *include* folder and add the *libfuzzylite* library, because the server needs to be built.

```
1 include_directories(${catkin_INCLUDE_DIRS} include)
2
3 target_link_libraries(FuzzyServer ${catkin_LIBRARIES}
    libfuzzylite.so)
```

Then, you can compile the package.

```
robo@robo:~$ rosrun tutorial2_controller FuzzyServer
```

Fig. 16 Fuzzy controller server

```
robo@robo:~$ rosrun tutorial2_controller FuzzyClient
[ INFO] [1464099516.975965773]: /fuzzy_client Waiting For Server...
[ INFO] [1464099517.176285388]: /fuzzy_client Got a Server...
```

Fig. 17 Fuzzy controller client

```
robo@robo:~$ rostopic info /fuzzy_control/goal
Type: tutorial2_controller/FuzzyControlActionGoal

Publishers:
 * /fuzzy_client (http://robo:48930/)

Subscribers:
 * /fuzzy_server (http://robo:53594/)

robo@robo:~$
```

Fig. 18 Getting fuzzy goal information

```
1 $ cd /home/user/catkin_ws/
2 $ catkin_make
3 $ source /home/user/catkin_ws/devel/setup.bash
```

6th step: Run controller To run the package, open a terminal and start the ROS.

```
1 $ roscore
```

Start the server in new terminal. In the Fig. 16, Fuzzy Controller server waits the client.

And start the client in new terminal (Fig. 17).

In Fig. 18 the goal information can be seen: type of message, publisher and subscriber.

The Fig. 19 shows the exchange of messages between the server and client.

You can also use the *roslaunch* to start the controller.

```
1 $ roslaunch tutorial2_controller fuzzycontrol.launch
```

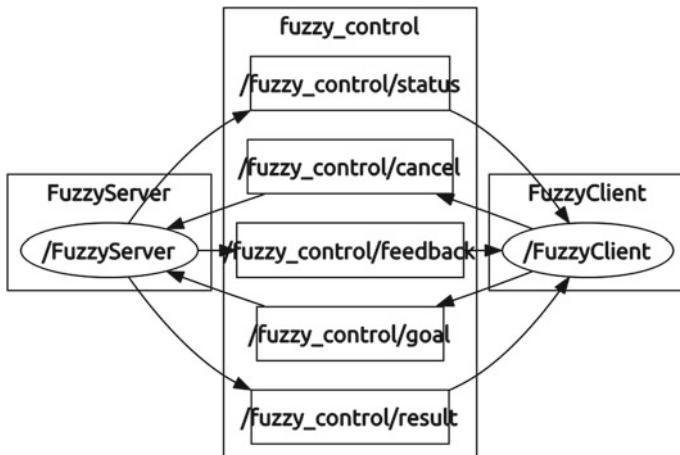


Fig. 19 Interaction between nodes of fuzzy controller

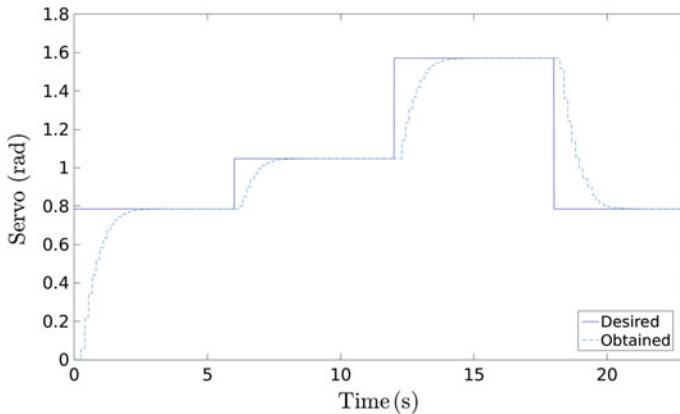


Fig. 20 Fuzzy control servo's position

5.2 Experimental Results of Fuzzy Controller

The results of servo position Fuzzy control are demonstrated in Fig. 20. The fuzzy controller didn't present overshoot in its response curve, even though it had a considerable response time due to the delay of the encoder.

6 Scheduled Fuzzy Controllers for Omnidirectional Motion of an Autonomous Inspection Robot

The scheduled fuzzy controller of AIR-2 is based on the linear velocities \dot{x}_G and \dot{y}_G and angular velocity $\dot{\theta}_G$, as presented in Eq. 3. According to the inputs, switcher (MUX) will choose which controller should be activated. If inputs are the linear velocities, linear motion controller will be activated. Already, when input is only angular velocity, the orientation controller will be enabled. And when inputs are linear and angular velocities, the free motion controller will be activated. The scheduled controllers can be seen in Fig. 21. The experimental results were simulated using V-REP. The control was implemented with ActionLib and fuzzylite library.

$$\xi_G = \begin{bmatrix} \dot{x}_G \\ \dot{y}_G \\ \dot{\theta}_G \end{bmatrix} \quad (3)$$

The feedback loop of each controller is related to each control variable. The linear velocities \dot{x}_R and \dot{y}_R of AIR-2 give feedback to linear motion and the angular velocity $\dot{\theta}_R$ of AIR-2 provides feedback to orientation motion. While, linear velocities of AIR-2 and angular velocity $\dot{\beta}_R$ of servo motors give feedback to free motion, due to side slip constraint, AIR-2 can't reorient while moving.

Each motion controller (linear, orientation and free motion) is composed of 8 Fuzzy controllers, in which 4 controllers perform velocity control of brushless motors and other 4 controllers are responsible by angle control of servo motors.

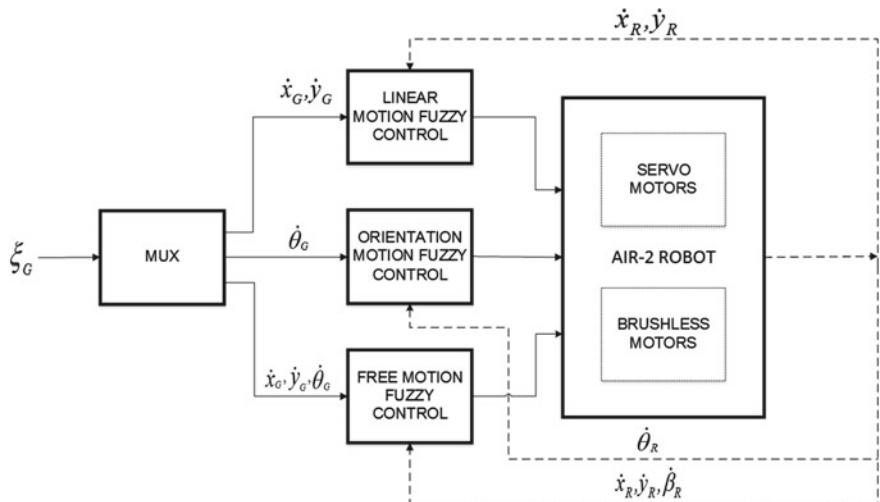


Fig. 21 Scheduled fuzzy controllers of AIR-2

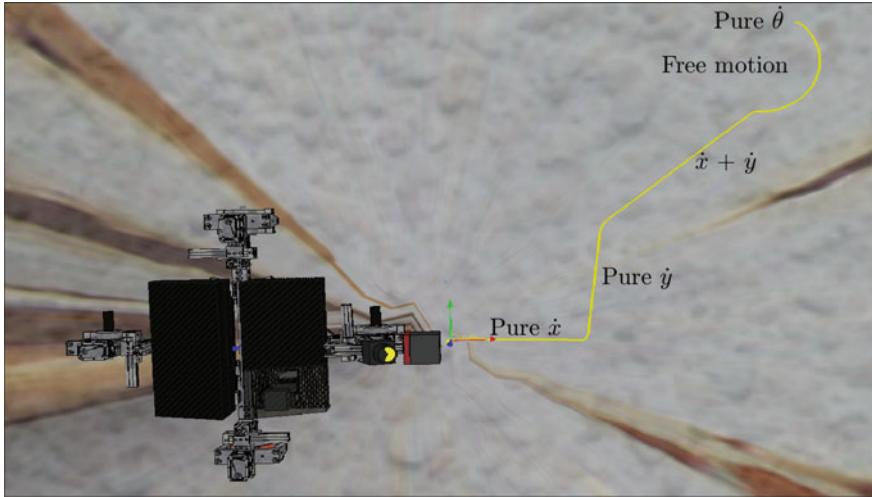


Fig. 22 AIR-2 path in the LPG sphere

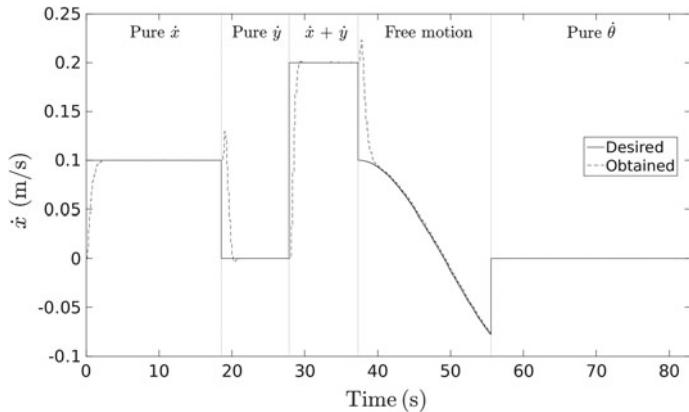


Fig. 23 Desired and obtained \dot{x}

A path with five different setpoints is generated for experimental results, which can be seen in Fig. 22.

In first, second and third setpoint, the AIR-2 has been set with linear motion, the setpoint was, respectively, \dot{x} , \dot{y} and $\dot{x} + \dot{y}$. The fourth was a free motion with \dot{x} and $\dot{\theta}$. And the fifth setpoint was orientation motion, that means only $\dot{\theta}$.

The low response time of brushless and servo motors produce overshoots that can be seen Figs. 23 and 24. It's caused by sampling frequency of encoders in V-REP.

The Fig. 25 shows the response controller to the $\dot{\theta}$. It has a small oscillation, due data provided by the IMU, even filtered, these data present a great noise. Even so, the control of angular velocity features a good response. In free motion, the high delay of

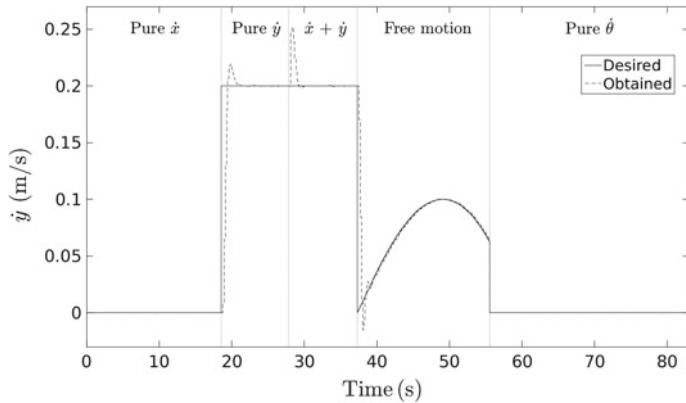


Fig. 24 Desired and obtained \dot{y}

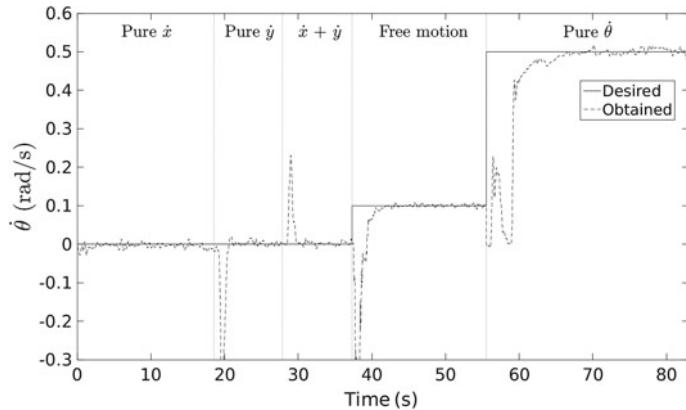


Fig. 25 Desired and obtained $\dot{\theta}$

controller is caused by reorientation wheels. It's necessary to stop brushless motors and servo motors orientate. The servo motors of front and rear are positioned at 90 degrees and left and right are positioned at zero degrees. So, the brushless motors are actuated to control the angular velocity of the AIR-2.

The overshoots and the delay times presented in the speed control don't influence the inspection, since inspection robot operating velocities are low.

The experimental results can be seen in a YouTube video available at the link below:

<https://youtu.be/46EKARDyP0w>.

7 Conclusion

The ActionLib has proved that the package can be used to implement controllers, exhibited good results as shown in the examples. Easy design of the package allows you to make any adjustments to your control, it allows even implement new control algorithms. The main disadvantage of ActionLib is non-real time, but its preemptive features allow almost periodic execution of the controller.

References

1. Wiki, R. 2016. Actionlib. <http://wiki.ros.org/actionlib/>.
2. Veiga, R., A.S. de Oliveira, L.V.R. Arruda, and F.N. Junior. 2015. Localization and navigation of a climbing robot inside a LPG spherical tank based on dual-lidar scanning of weld beads. In *Springer Book on Robot Operating System (ROS): The Complete Reference*. New York: Springer.
3. Ren, L., W. Wang, and Z. Du. 2012. A new fuzzy intelligent obstacle avoidance control strategy for wheeled mobile robot. In *2012 IEEE International Conference on Mechatronics and Automation*, 1732–1737.
4. Pratama, D., E.H. Binugroho, and F. Ardilla. 2015. Movement control of two wheels balancing robot using cascaded PID controller. In *International Electronics Symposium (IES)*, 94–99.
5. de Oliveira, A., L. de Arruda, F. Neves, R. Espinoza, and J. Nadas. 2012. Adhesion force control and active gravitational compensation for autonomous inspection in lpg storage spheres. In *Robotics Symposium and Latin American Robotics Symposium (SBR-LARS), 2012 Brazilian*, 232–238.
6. Robotics, C. 2016. Coppelia robotics v-rep: Create. compose. simulate. any robot. <http://www.coppeliarobotics.com/>.
7. Wiki, R. 2016. rqt_graph. http://wiki.ros.org/rqt_graph.
8. Wiki, R. 2016. rqt_plot. http://wiki.ros.org/rqt_plot.
9. Rada-Vilela, J. 2014. Fuzzylite: a fuzzy logic control library. <http://www.fuzzylite.com>.

Parametric Identification of the Dynamics of Mobile Robots and Its Application to the Tuning of Controllers in ROS

Walter Fetter Lages

Abstract This tutorial chapter explains the identification of dynamic parameters of the dynamic model of wheeled mobile robots. Those parameters depend on the mass and inertia parameters of the parts of the robot and even with the help of modern CAD systems it is difficult to determine them with a precision as the designed robot is not built with 100% accuracy; the actual materials have not exactly the same properties as modeled in the CAD system; there is cabling which density changes over time due to robot motion and many other problems due to differences between the CAD model and the real robot. To overcome these difficulties and still have a good representation of the dynamics of the robot, this work proposes the identification of the parameters of the model. After an introduction to the recursive least-squares identification method, it is shown that the dynamic model of a mobile robot is a cascade between its kinematic model, which considers velocities as inputs, and its dynamics, which considers torques as inputs and then that the dynamics can be written as a set of equations linearly parameterized in the unknown parameters, enabling the use of the recursive least-squares identification. Although the example is a differential-drive robot, the proposed method can be applied to any robot model that can be parameterized as the product of a vector of parameters and a vector of regressors. The proposed parameter identification method is implemented in a ROS package and can be used with actual robots or robots simulated in Gazebo. The package for the Indigo version of ROS is available at <http://www.ece.ufrgs.br/twil/indigo-twil.tgz>. The chapter concludes with a full example of identification and the presentation of the dynamic model of a mobile robot and its use for the design of a controller. The controller is based on three feedback loops. The first one linearizes the dynamics of the robot by using feedback linearization, the second one uses a set of PI controllers to control the dynamics of the robot, and the last one uses a non-linear controller to control the pose of the robot.

W.F. Lages (✉)

Federal University of Rio Grande do Sul, Av. Osvaldo Aranha, 103,

Porto Alegre RS 90035-190, Brazil

email: fetter@ece.ufrgs.br

URL: <http://www.ece.ufrgs.br/~fetter>

Keywords Parametric identification · Dynamic model · Recursive least-squares · Controller tuning · Feedback linearization · Non-smooth controller

1 Introduction

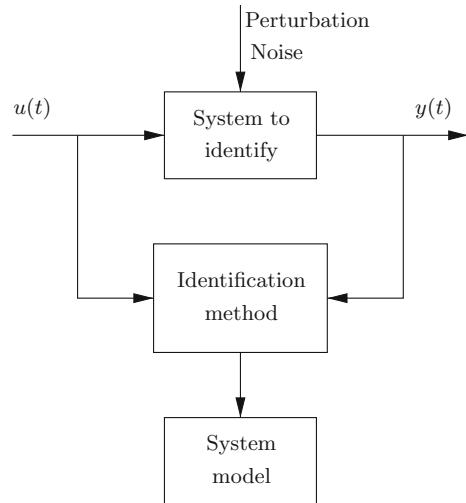
Like any robot, a wheeled mobile robot is subject to kinematics and dynamics. Also, its kinematic model depends only on geometric parameters, while the dynamic model depends on geometric parameters and mass and inertia moments. However, different from manipulator robots, the kinematic model of a wheeled mobile robot is represented by differential equations. The output of the kinematic model of a mobile robot is its pose (position and orientation) while its inputs are velocities. Depending on how the model is formulated, its inputs may be the velocity on each wheels or angular and linear velocity of the robot, or any other variable which is homogeneous to velocity [9].

Based on those properties of the kinematic model of mobile robots, many controllers for mobile robots command velocities to the robot, under the assumption that those commanded velocities are instantaneously imposed on the robot. Of course, that assumption is only valid if the actuators are powerful enough regarding the mass and inertia moments of the robot. That is indeed the case for small robots or robots using servo motors, which are commanded in velocity, as actuators. In this case, the controller can be designed based on the kinematic model alone, whose parameters are usually well-known. Note that, as the kinematic model of a mobile robot is given by a differential equation, it is often called a dynamic model (because its described by dynamic, i.e. differential equations) although it does not effectively model the dynamics of the robot. In this chapter, that model is referred to as a kinematic model. The term dynamic model is reserved for models that describe the dynamics of the robot.

However, for larger robots, or robots in which actuators are not powerful enough to impose very fast changes in velocities, it is necessary to consider the dynamics of the robot in the design of the controller. Then, a more sophisticated model of the robot, including its kinematics and its dynamics, should be used. The output of this model is the robot pose, as well as in the kinematic model, but its inputs are the torques on the wheels. The parameters of this model depends on the mass and inertia parameters of the parts of the robot and even with the help of modern CAD systems it is difficult to know with good precision some of those values as the designed robot is not built with 100% accuracy; the actual materials have not exactly the same properties as modeled in the CAD system; there is cabling which density changes over time due to motion and many other problems.

To overcome the difficulties in considering all constructive details in a mathematical model and still have a good representation of the dynamics of the robot, it is possible to obtain a model by observing the system output given proper inputs as shown in Fig. 1. This procedure is called system identification [10] or model learning [18].

Fig. 1 Basic block diagram for system identification



It is shown that the dynamic model of a mobile robot can be properly parameterized such that the recursive least-squares method [10] can be used for parameter identification. The proposed parameter identification method is implemented in a ROS package and can be used with actual robots or robots simulated in Gazebo. The package for the Indigo version of ROS can be downloaded from <http://www.ece.ufrgs.br/twil/indigo-twil.tgz>. See Sect. 3 for details on how to install it.

The identified parameters and the respective diagonal of the covariance matrix are published as ROS topics to be used in the off-line design of controllers or even used online to implement adaptive controllers. The diagonal of the covariance matrix is a measure of confidence on the parameter estimation and hence can be used to decide if identified parameters are good enough. In the case of an adaptive controller, it can be used to decide if the adaptation should be shut-off or not.

The chapter concludes with a complete example of identification and controller design. Note that although the example and the general method is developed for differential-drive mobile robots, it can be applied to any robot, as long as it is possible to write the model in a way such that the unknown parameters are linearly related to the measured variables, as shown in Sect. 2.1.

More specifically, the remainder of this chapter will cover the following topics:

- a background on identification
- a background on modeling of mobile robots
- installing the required packages
- testing the installed packages
- description of the package for identification of mobile robots.

2 Background

2.1 Parametric Identification

In order to design a control system it is generally necessary to have a model of the plant (the system to be controlled). In many cases, those models can be obtained by analyzing how the system works and using the laws of Physics to write the set of equations describing it. This is called the white-box approach. However, sometimes it is not possible to obtain the model using this approach, due to complexity of the system or uncertainty about its parameters or operating conditions. In those cases, it might be possible to obtain a model through the observation of the system behavior as shown in Fig. 1, which is known as the black-box approach and formally called system identification.

In this chapter, the focus is on identification methods which can be used online, because they are more convenient for computational implementation and can be readily used for implementing adaptive controllers. When the parameter estimation is performed online, it is necessary to obtain a new updated estimate in the period between two successive samples. Hence, it is highly desirable for the estimation algorithm to be simple and easily implementable. A particularly interesting class of online algorithms are those in which the current estimate $\theta(t)$ is computed as a function of the former estimates, and then it is possible to compute the estimates recursively.

Let a single-input, single-output (SISO) system represented by its ARX¹ model:

$$\begin{aligned} y(t+1) = & a_1 y(t) + \cdots + a_p y(t-p+1) \\ & + b_1 u(t) + \cdots + b_q u(t-q+1) + \omega(t+1) \end{aligned} \quad (1)$$

where t is the sampling period index,² $y \in \mathbb{R}$ is the system output, $u \in \mathbb{R}$ is the system input, $a_i, i = 1, 2, \dots, p$ and $b_i, j = 1, 2, \dots, q$ are the system parameters and $\omega(t+1)$ is a Gaussian noise representing the uncertainty in the model.

The model (1) can be rewritten as:

$$y(t+1) = \phi^T(t)\theta + \omega(t+1) \quad (2)$$

¹AutoRegressive with eXogenous inputs.

²Note that in system identification theory it is common to use t as the independent variable even though the model is a discrete time one.

with

$$\theta = \begin{bmatrix} a_1 \\ \vdots \\ a_p \\ b_1 \\ \vdots \\ b_1 \end{bmatrix}, \text{ the vector of parameters} \quad (3)$$

and

$$\phi(t) = \begin{bmatrix} y(t) \\ \vdots \\ y(t-p+1) \\ u(t) \\ \vdots \\ u(t-q+1) \end{bmatrix}, \text{ the regression vector.} \quad (4)$$

The identification problem consists in determining θ based on the information (measurements) about $y(t+1)$ and $\phi(t)$ for $t = 0, 1, \dots, n$. To solve this problem, it can be formulated as an optimization problem with the cost to minimize:

$$J(n, \theta) = \frac{1}{n} \sum_{t=0}^{n-1} (y(t+1) - \phi^T(t)\theta)^2 \quad (5)$$

where $y(t+1) - \phi^T(t)\theta$ is the prediction error.

More formally:

$$\hat{\theta}(n) = \arg \min_{\theta} J(n, \theta) \quad (6)$$

Figure 2 shows a block diagram of the identification system implementing (6).

In order to solve the minimization (6) it is convenient to write it as:

$$\hat{\theta}(n) = \arg \min_{\theta} ((Y(n) - \Phi(n)\theta)^T (Y(n) - \Phi(n)\theta)) \quad (7)$$

with

$$Y(n) = \begin{bmatrix} y(1) \\ y(2) \\ \vdots \\ y(n) \end{bmatrix} \quad (8)$$

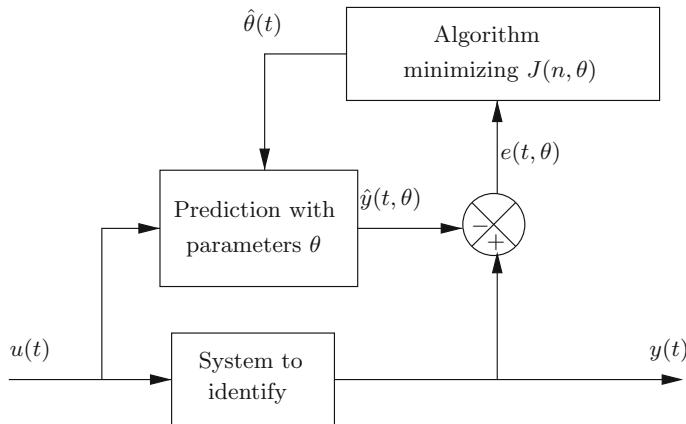


Fig. 2 Block diagram of system identification

and

$$\Phi(n) = \begin{bmatrix} \phi^T(0) \\ \phi^T(1) \\ \vdots \\ \phi^T(n-1) \end{bmatrix} \quad (9)$$

Then, (6) can be solved by making the differential of $J(n, \theta)$ with respect to θ equal to zero:

$$\frac{\partial J(n, \theta)}{\partial \theta} \Big|_{\theta=\hat{\theta}(n)} = 0 = -2\Phi^T(n)Y(n) + 2\Phi^T(n)\Phi(n)\hat{\theta}(n) \quad (10)$$

Hence,

$$\hat{\theta}(n) = (\Phi^T(n)\Phi(n))^{-1}\Phi^T(n)Y(n) \quad (11)$$

or

$$\hat{\theta}(n) = \left(\sum_{t=0}^{n-1} \phi(t)\phi^T(t) \right)^{-1} \sum_{t=0}^{n-1} \phi(t)y(t+1) \quad (12)$$

Expression (12) is the solution of (6) and can be used to compute an estimate $\hat{\theta}$ for the vector of parameters θ at time instant n . However, this expression is not in a recursive form and is not practical for online computing because it requires the inversion of a matrix of dimension $(n-1) \times (n-1)$ for each update of the estimate. Furthermore, n keeps increasing without bound, thus increasing computation time and memory requirements.

For online computation it is convenient to have a recursive form of (12), such that at each update time, the new data can be assimilated without the need to compute everything again. To obtain such a recursive form define:

$$P(n) = \left(\sum_{t=0}^n \phi(t)\phi^T(t) \right)^{-1} \quad (13)$$

then, from (12):

$$\hat{\theta}(n+1) = P(n) \sum_{t=0}^n \phi(t)y(t+1) \quad (14)$$

On the other hand:

$$P^{-1}(n) = \sum_{t=0}^n \phi(t)\phi^T(t) \quad (15)$$

$$= \sum_{t=0}^{n-1} \phi(t)\phi^T(t) + \phi(n)\phi^T(n) \quad (16)$$

$$= P^{-1}(n-1) + \phi(n)\phi^T(n) \quad (17)$$

or

$$P(n) = (P^{-1}(n-1) + \phi(n)\phi^T(n))^{-1} \quad (18)$$

$$= P(n-1) - P(n-1)\phi(n)(\phi^T(n)P(n-1)\phi(n) + 1)^{-1}\phi^T(n)P(n-1) \quad (19)$$

By using the Matrix Inversion Lemma³ [3] with $A = P^{-1}(n-1)$, $B = \phi(n)$, $C = 1$ e $D = \phi^T(n)$, it is possible to compute:

$$P(n) = P(n-1) - \frac{P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \quad (20)$$

which, replaced in (14) results:

$$\hat{\theta}(n+1) = P(n) \sum_{t=0}^n \phi(t)y(t+1) \quad (21)$$

$$= P(n) \left(\sum_{t=0}^{n-1} \phi(t)y(t+1) + \phi(n)y(n+1) \right) \quad (22)$$

³Matrix Inversion Lemma: $(A + BCD)^{-1} = A^{-1} - A^{-1}B(C^{-1} + DA^{-1}B)^{-1}DA^{-1}$.

$$= \left(P(n-1) - \frac{P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \right) \\ \left(\sum_{t=0}^{n-1} \phi(t)y(t+1) + \phi(n)y(n+1) \right) \quad (23)$$

By expanding the product:

$$\hat{\theta}(n+1) = P(n-1) \sum_{t=0}^{n-1} \phi(t)y(t+1) + P(n-1)\phi(n)y(n+1) \\ - \frac{P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \sum_{t=0}^{n-1} \phi(t)y(t+1) \\ - \frac{P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \phi(n)y(n+1) \quad (24)$$

Then by delaying (14) a sampling period and replacing in (24):

$$\hat{\theta}(n+1) = \hat{\theta}(n) + P(n-1)\phi(n)y(n+1) \\ - \frac{P(n-1)\phi(n)\phi^T(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \hat{\theta}(n) \\ - \frac{P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \phi(n)y(n+1) \quad (25)$$

and by grouping together the terms in $\phi(n)y(n+1)$:

$$\hat{\theta}(n+1) = \hat{\theta}(n) + \\ + \frac{P(n-1) + P(n-1)\phi^T(n)P(n-1)\phi(n) - P(n-1)\phi(n)\phi^T(n)P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \phi(n)y(n+1) \\ - \frac{P(n-1)\phi(n)\phi^T(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \hat{\theta}(n) \quad (26)$$

or

$$\hat{\theta}(n+1) = \hat{\theta}(n) + \frac{P(n-1)}{1 + \phi^T(n)P(n-1)\phi(n)} \phi(n)y(n+1) \\ - \frac{P(n-1)\phi(n)\phi^T(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \hat{\theta}(n) \quad (27)$$

which can be rewritten as:

$$\hat{\theta}(n+1) = \hat{\theta}(n) + \frac{P(n-1)\phi(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \left(y(n+1) - \phi^T(n)\hat{\theta}(n) \right) \quad (28)$$

The term multiplying the error can be regarded as the optimal gain of the identification algorithm. Hence, the solution for the problem (6) in a recursive form is given by:

$$\hat{\theta}(n+1) = \hat{\theta}(n) + K(n) \left(y(n+1) - \phi^T(n)\hat{\theta}(n) \right) \quad (29)$$

$$K(n) = \frac{P(n-1)\phi(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \quad (30)$$

$$P(n) = (I - K(n)\phi^T(n)) P(n-1) \quad (31)$$

Expression (29) is an update of the previous parameter estimate $\hat{\theta}(n)$ by an optimal gain $K(n)$, from (30), multiplied by the prediction error, $y(n+1) - \hat{y}(n+1)$. Note that $\hat{y}(n+1) = \phi^T(n)\hat{\theta}(n)$ is the prediction of the system output. It can be shown that $P(n)$ as computed by (31) is the covariance of the prediction error and hence it is a measure of the confidence in the parameter estimates.

The Algorithm 1 details the procedure for parameter identification:

Algorithm 1 Recursive Least-Squares.

Initialize $\phi(0), \hat{\theta}(0), \text{e } P(-1) = cI$.

At sampling time $n+1$:

1. Read system output $y(n+1)$ from sensors
2. Compute the prediction of system output $\hat{y}(n+1)$:

$$\hat{y}(n+1) = \phi^T(n)\hat{\theta}(n) \quad (32)$$

3. Compute the gain $K(n)$:

$$K(n) = \frac{P(n-1)\phi(n)}{1 + \phi^T(n)P(n-1)\phi(n)} \quad (33)$$

4. Update the parameter vector estimate:

$$\hat{\theta}(n+1) = \hat{\theta}(n) + K(n) (y(n+1) - \hat{y}(n+1)) \quad (34)$$

5. Store $\hat{\theta}(n+1)$ for use, if necessary
6. Update the covariance matrix:

$$P(n) = (I - K(n)\phi^T(n)) P(n-1) \quad (35)$$

7. Wait for the next sampling time
 8. Increment n and return to step 1
-

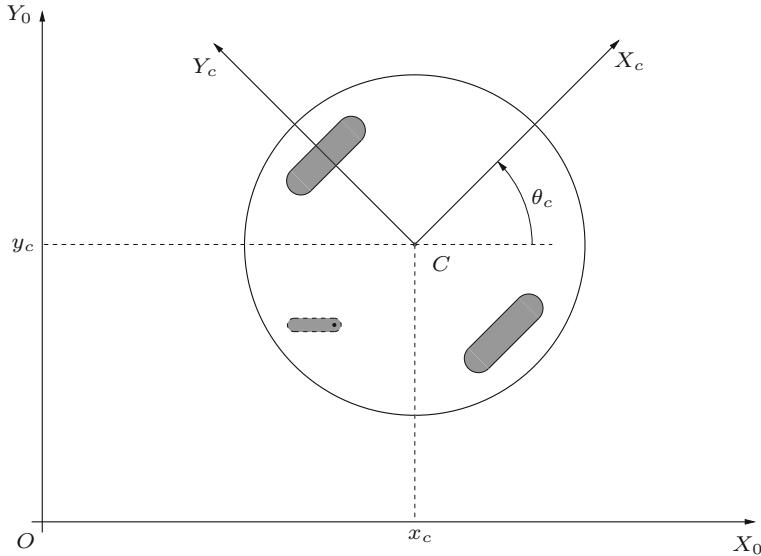


Fig. 3 Coordinate systems

2.2 Mobile Robot Model

The model of the mobile robot used in this chapter is described in this section. Figure 3 shows the coordinate systems used to describe the mobile robot model, where X_c and Y_c are the axes of the coordinate system attached to the robot and X_0 and Y_0 form the inertial coordinate system.

The pose (position and orientation) of the robot is represented by $x = [x_c \ y_c \ \theta_c]^T$. The mobile robot dynamic model can be obtained based on the Lagrange-Euler formulation [9] and is given by:

$$\begin{cases} \dot{x} = B(x)u \\ H(\beta)\dot{u} + f(\beta, u) = F(\beta)\tau \end{cases} \quad (36)$$

where β is the vector of the angles of the caster wheels, $u = [v \ \omega]^T$ is the vector of the linear and angular velocities of the robot and τ is the vector of input torques on the wheels. $B(x)$ is a matrix whose structure depends on the kinematic (geometric) properties of the robot, while $H(\beta)$, $f(\beta, u)$ and $F(\beta)$ depend on the kinematic and dynamic (mass and inertia) parameters of the robot. Although this chapter is based on a differential-drive mobile robot, the model (36) is valid for any type of wheeled mobile robot. See [9] for details and examples for other types of wheeled mobile robots.

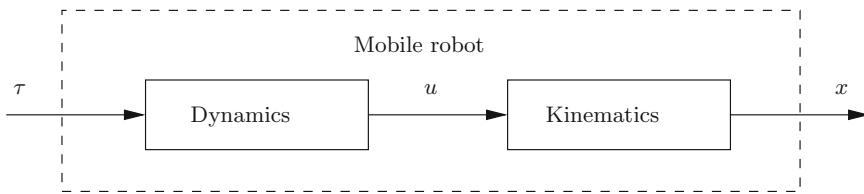


Fig. 4 Cascade between dynamics and the kinematic model

Fig. 5 The Twil mobile robot



Note that the dynamic model of the robot is a cascade between its kinematic model (the first expression of (36), with velocities as inputs) and its dynamics (the second expression of (36), with torques as inputs), as shown in Fig. 4.

This chapter is based on the Twil mobile robot (see Fig. 5), which is a differential-drive mobile robot, but the results and the ROS package for parameter identification can be used directly for any other differential-drive mobile robot, as it does not depend on Twil characteristics. For other types of wheeled mobile robots, the model has the same form as (36) and given its particular characteristics such as the location of the wheels with respect to the robot reference frame, the model (36) can be customized and rewritten in a form similar to the one used here for differential-drive robots. Then, the same procedure could be used for parameter estimation.

The matrices of the model (36) customized for a differential-drive robot such as Twil are:

$$B(x) = \begin{bmatrix} \cos \theta_c & 0 \\ \sin \theta_c & 0 \\ 0 & 1 \end{bmatrix} \quad (37)$$

$$H(\beta) = I \quad (38)$$

$$f(\beta, u) = - \begin{bmatrix} 0 & K_5 \\ K_6 & 0 \end{bmatrix} \begin{bmatrix} u_1 u_2 \\ u_2^2 \end{bmatrix} = -f(u) \quad (39)$$

$$F(\beta) = \begin{bmatrix} K_7 & K_7 \\ K_8 & -K_8 \end{bmatrix} = F \quad (40)$$

where I is the identity matrix and K_5 , K_6 , K_7 and K_8 are constants depending only on the geometric and inertia parameters of the robot. Note that for this robot $H(\beta)$, $f(\beta, u)$ and $F(\beta)$ do not actually depend on β .

Note that only the dynamics of the robot depends on mass and inertia parameters, which are difficult to know with precision. Furthermore, u is a vector with linear and angular velocity of the robot, which can be easily measured, while x are the robot pose, which are more difficult to be obtained. However, the parameter of the kinematics depends on the geometry of the robot and can be obtained with good precision by calibration. Therefore, only the part of the model regarding the dynamics is used for parameter estimation, taking u as output and τ as input.

In the following it is shown that the dynamics of the robot can be written as a set of equations in the form of $y(k+1) = \phi^T(k)\theta(k)$, where y is the acceleration (measured or estimated from velocities), ϕ is the vector of regressors (measured velocities and applied torques) and θ is the vector of unknown parameters to be identified. Then, it is possible to obtain an estimate $\hat{\theta}$ for θ by using the recursive least squares algorithm [10] described in Sect. 2.1.

The parameters K_5 , K_6 , K_7 and K_8 depend on the geometric and mass properties of the robot in a very complex way. Even for a robot simulated in Gazebo, the same problem arises, as the model (36) is more simple than a typical robot described in URDF, which typically include more constructive details, for a realistic and good looking animation. On the other hand, the model described in URDF is not available in a closed form as (36), whose structure can be explored in the design of a controller. Also, it is not trivial to obtain a model in the form of (36) equivalent to an URDF description.

To overcome the difficulties in considering all constructive details in an algebraic model such as (36) and still have a good representation of the dynamics of the robot, the parameters of the model are identified.

In obtaining a model in a form suitable to be identified by the recursive least squares algorithm described in Sect. 2.1 it is important to note that only the second expression of (36) depends on the unknown parameters. Furthermore, u is a vector with linear and angular velocity of the robot, which can be easily measured, while x are the robot pose, which are more difficult to be obtained. Therefore, only the second expression of (36) will be used for parameter estimation, taking u as output and τ as input.

By using (37)–(40), the second expression of (36) for the Twil robot can be written as:

$$\dot{u} = \begin{bmatrix} 0 & K_5 \\ K_6 & 0 \end{bmatrix} \begin{bmatrix} u_1 u_2 \\ u_2^2 \end{bmatrix} + \begin{bmatrix} K_7 & K_7 \\ K_8 & -K_8 \end{bmatrix} \tau \quad (41)$$

Although (41) seems somewhat cryptic, its physical meaning can be understood as $\dot{u} = [\dot{v} \dot{\omega}]^T$ is the vector of linear and angular acceleration of the robot. Hence, the term $K_5 u_2^2$ represents the centrifugal, the term $K_6 u_1 u_2$ represents the Coriolis acceleration. Also, as the linear acceleration of the robot is proportional to the average of the torques applied to the left and right wheels, $1/K_7$ represents the robot mass and as the angular acceleration of the robot is proportional to the difference of torques, $1/K_8$ represents the moment of inertia of the robot.

For the purpose of identifying K_5 , K_6 , K_7 and K_8 it is convenient to write (41) as two scalar expressions:

$$\dot{u}_1 = K_5 u_2^2 + K_7 (\tau_1 + \tau_2) \quad (42)$$

$$\dot{u}_2 = K_6 u_1 u_2 + K_8 (\tau_1 - \tau_2) \quad (43)$$

Then, by discretizing (42)–(43) it is possible to obtain two recursive models: one linearly parameterized in K_5 and K_7 and another linearly parameterized in K_6 and K_8 :

$$y_1(k+1) = \dot{u}_1(k) \simeq \frac{u_1(k+1) - u_1(k)}{T} \quad (44)$$

$$= K_5 u_2^2(k) + K_7(\tau_1(k) + \tau_2(k)) \quad (45)$$

$$y_1(k+1) = \begin{bmatrix} u_2^2(k) \\ \tau_1(k) + \tau_2(k) \end{bmatrix}^T \begin{bmatrix} K_5 \\ K_7 \end{bmatrix} \quad (46)$$

$$y_2(k+1) = \dot{u}_2(k) \simeq \frac{u_2(k+1) - u_2(k)}{T} \quad (47)$$

$$= K_6 u_1(k) u_2(k) + K_8(\tau_1(k) - \tau_2(k)) \quad (48)$$

$$y_2(k+1) = \begin{bmatrix} u_1(k) u_2(k) \\ \tau_1(k) - \tau_2(k) \end{bmatrix}^T \begin{bmatrix} K_6 \\ K_8 \end{bmatrix} \quad (49)$$

Note that it is easier and more convenient to identify two models depending on two parameters each one than to identify a single model depending on four parameters.

Then, by defining:

$$\phi_1(k) = \begin{bmatrix} u_2^2(k) \\ \tau_1(k) + \tau_2(k) \end{bmatrix} \quad (50)$$

$$\theta_1(k) = \begin{bmatrix} K_5 \\ K_7 \end{bmatrix} \quad (51)$$

$$\phi_2(k) = \begin{bmatrix} u_1(k)u_2(k) \\ \tau_1(k) - \tau_2(k) \end{bmatrix} \quad (52)$$

$$\theta_2(k) = \begin{bmatrix} K_6 \\ K_8 \end{bmatrix} \quad (53)$$

it is possible to write (46) and (49) as:

$$y_1(k+1) = \phi_1^T(k)\theta_1(k) \quad (54)$$

$$y_2(k+1) = \phi_2^T(k)\theta_2(k) \quad (55)$$

and then, it is possible to obtain an estimate $\hat{\theta}_i$ for θ_i by using a standard recursive least squares algorithm such as described in Sect. 2.1:

$$\hat{y}_i(n+1) = \phi_i^T(n)\hat{\theta}_i(n) \quad (56)$$

$$K_i(n) = \frac{P_i(n-1)\phi_i(n)}{1 + \phi_i^T(n)P(n-1)\phi_i(n)} \quad (57)$$

$$\hat{\theta}_i(n+1) = \hat{\theta}_i(n) + K_i(n)(y_i(n+1) - \hat{y}_i(n-1)) \quad (58)$$

$$P_i(n) = (I - K_i(n)\phi_i^T(n))P_i(n-1) \quad (59)$$

where $\hat{y}_i(n+1)$ are estimates for $y_i(n+1)$, $K_i(n)$ are the gains and $P_i(n)$ are the covariance matrices.

3 ROS Packages for Identification of Robot Model

This section describes the installation of some packages useful for the implementation of the identification procedure described in Sect. 2. Some of them are not present in a standard installation of ROS and should be installed. Also, some custom packages with our implementation of the identification should be installed.

3.1 Setting up a Catkin Workspace

The packages to be installed for implementing ROS controllers assume an existing catkin workspace. If it does not exist, it can be created with the following commands (assuming a ROS Indigo version):

```
source /opt/ros/indigo/setup.bash
mkdir -p ~/catkin_ws/src
cd ~/catkin_ws/src
catkin_init_workspace
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

3.2 ***ros_control***

The `ros_control` meta-package includes a set of packages to implement generic controllers. It is a rewrite of the `pr2_mechanism` packages to be used with all robots and no just with PR2. This package implements the base architecture of ROS controllers and hence is required for setting up controllers for the robot. In particular, for the identification method proposed here, it is necessary to actuate the robot directly, that is, without any controller, neither an open-loop nor a closed-loop controller. This can be done in ROS by configuring a forward controller, a controller that just replicates its input in its output.

This meta-package includes the following packages:

`control_toolbox`: contains classes that are useful for all controllers, such as PID controllers.

`controller_interface`: implements a base class for interfacing with controllers, the `Controller` class.

`controller_manager`: implements the `ControllerManager` class, which loads, unloads, starts and stops controllers.

`hardware_interface`: base class for implementing the hardware interface, the `RobotHW` and `JointHandle` classes.

`joint_limits_interface`: base class for implementing the joint limits.

`transmission_interface`: base class for implementing the transmission interface.

`realtime_tools`: contains a set of tool that can be used from a hard real-time thread.

The `ros_control` meta-package is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-ros-control
```

3.3 ***ros_controllers***

This meta-package is necessary to make the `forward_command_controller` and `forward_command_controller` controllers available. The robot is put in

open-loop by using the `forward_command_controller` controller, and then the desired input signal can be applied for identification. The `joint_state_controller` controller, which by its name seems an state-space controller in the joint space, but actually it is just a publisher for the values of the position and velocities of the joints. Then, that topic is used to obtain the output of the robot system to be used in the identification.

More specifically, `ros_controllers` includes the following:

`forward_command_controller`: just a bypass from the reference to the control action as they are the same physical variable.

`effort_controllers`: implements effort controllers, that is, SISO controllers in which the control action is the torque (or an equivalent physical variable) applied to the robot joint. There are three types of `effort_controllers`, depending on the type of the reference and controlled variable:

`effort_controllers/joint_effort_controller`: just a bypass from the reference to the control action as they are the same physical variable.

`effort_controllers/joint_position_controller`: a controller in which the reference is joint position and the control action is torque. The PID control law is used.

`effort_controllers/joint_velocity_controller`: a controller in which the reference is joint velocity and the control action is torque. The PID control law is used.

`position_controllers`: implements SISO controllers in which the control action is the position (or an equivalent physical variable) applied to the robot joint.

Currently, there is just one type of `position_controllers`:

`position_controllers/joint_position_controller`: just a bypass from the reference to the control action as they are the same physical variable.

`velocity_controllers`: implements SISO controllers in which the control action is the velocity (or an equivalent physical variable) applied to the robot joint.

Currently, there is just one type of `velocity_controllers`:

`velocity_controllers/joint_velocity_controller`: just a bypass from the reference to the control action as they are the same physical variable.

`joint_state_controller`: implements a sensor which publishes the joint state as a `sensor_msgs/JointState` message, the `JointStateController` class.

The `ros_controllers` meta-package is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-ros-controllers
```

3.4 *gazebo_ros_pkgs*

This is a collection of ROS packages for integrating the `ros_control` controller architecture with the Gazebo simulator [12], containing the following:

`gazebo_ros_control`: Gazebo plugin that instantiates the `RobotHW` class in a `DefaultRobotHWSim` class, which interfaces with a robot simulated in Gazebo. It also implements the `GazeboRosControlPlugin` class.

The `gazebo_ros_pkgs` meta-package is not included in the standard ROS desktop installation, hence it should be installed. On Ubuntu, it can be installed from Debian packages with the command:

```
sudo apt-get install ros-indigo-gazebo-ros-pkgs ros-indigo-gazebo-ros-control
```

3.5 *twil*

This is a meta-package with the package for identification of the Twil robot. It contains an URDF description of the Twil mobile robot [4] and the implementation of the identification and some controllers used for identification and using the identified parameters. More specifically it includes the following packages:

`twil_description`: URDF description of the Twil mobile robot.

`twil_controllers`: implementation of a forward controller, a PID controller and a linearizing controller for the Twil mobile robot.

`twil_ident`: ROS node implementing the recursive least-squares for identification of the parameters of a differential-drive mobile robot.

The `twil` meta-package can be downloaded and installed in the ROS catkin workspace with the commands:

```
cd ~/catkin_ws/src
wget http://www.ece.ufrgs.br/twil/indigo-twil.tgz
tar -xzf indigo-twil.tgz
cd ~/catkin_ws
catkin_make
source ~/catkin_ws/devel/setup.bash
```

4 Testing the Installed Packages

A simple test for the installation of the packages described in Sect. 3 is performed here.

The installation of the ROS packages can be done by loading the Twil model in Gazebo and launching the computed torque controller with the commands:

```
source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
rosrun twil_controllers joint_effort.launch
```

The robot should appear in Gazebo as shown in Fig. 6.

Then, start the simulation by clicking in the play button in the Gazebo panel, open a new terminal and issue the following commands to move the robot.

```
source /opt/ros/indigo/setup.bash
source ~/catkin_ws/devel/setup.bash
rosrun twil_controllers test_openloop.sh
```

If everything is right, the Twil robot should move for some seconds and then stop, as shown in Fig. 7.

In this simulation, the Twil mobile robot is driven by standard ROS controllers implementing a *bypass* from its reference to its output. This is the equivalent to drive the robot in open-loop, that means, without any controller.

The `effort_controllers/JointEffortController` controller implements just a *bypass* from its input to its output as its input is effort and its output is effort, as well. The example uses one of such controllers in each wheel of the Twil mobile robot, effectively keeping it in open-loop. Hence, the reference applied to the controllers is directly the torque applied to each wheel. Figure 8 shows the computation graph for this example.

The controllers themselves are not shown because they are plugins loaded by the controllers manager and hence they are not individual ROS nodes. The right wheel controller receives its reference through the `/twil/right_wheel_joint_effort_controller_command` topic and the left wheel controller receives its through the `/twil/left_wheel_joint_effort_controller_command`

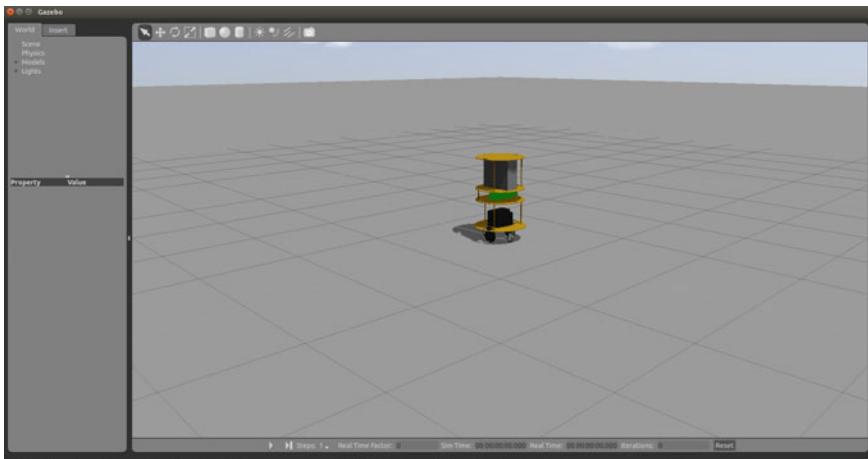


Fig. 6 Twil mobile Robot in Gazebo

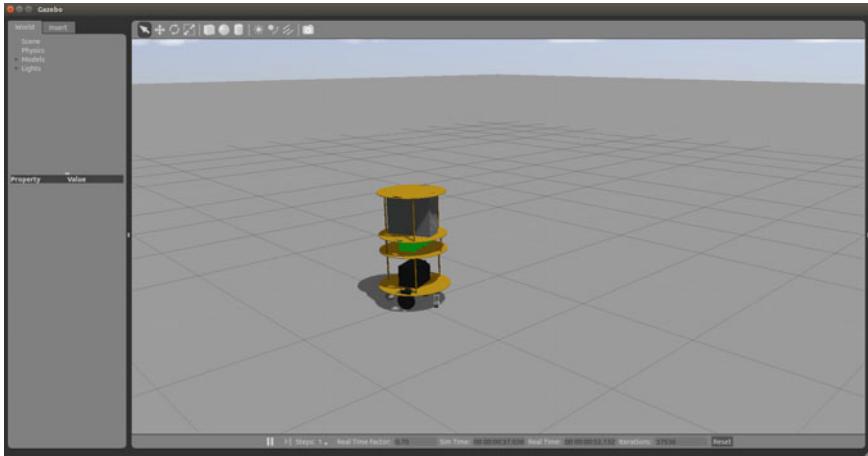


Fig. 7 Gazebo with Twil robot after test motion

topic. The `/joint_states` topic is where the state of the joints (wheels) are published by the `JointStateController` controller. In the next Sections the data published in this topic, will be used to identify the parameters of the Twil mobile robot. For a good identification, adequate signals, as detailed in Sect. 5.3, will be applied to the `/twil/right_wheel_joint_effort_controller_command` and `/twil/left_wheel_joint_effort_controller_command` topics.

The `test_openloop.sh` is a script with an example of how to set the reference for the controllers, in this case, the torque on right and left wheels of the Twil robot. The script just publishes the required values by using the `rostopic` command. In a real application, probably with a more sophisticated controller, those references would be generated by a planning package, such as MoveIt! [24] or a robot navigation package, such as the Navigation Stack [15, 16]. In the case of an identification task, as discussed here, the references for the controllers are generated by a node implementing the identification algorithm.

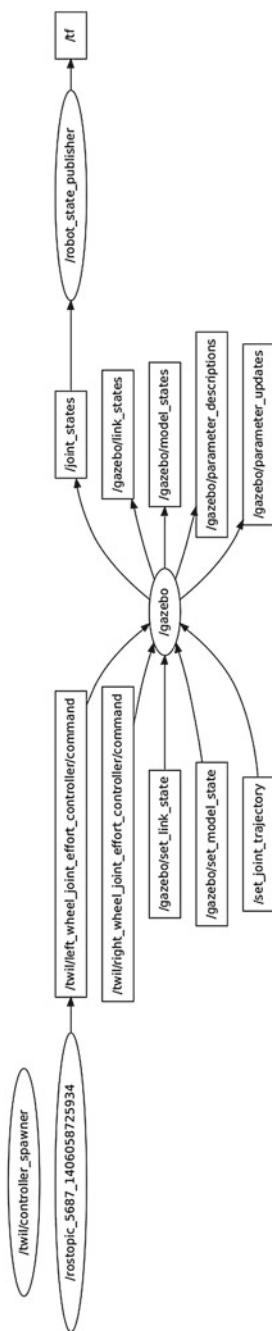


Fig. 8 Computation graph for Twil in open loop

5 Implementation of Parametric Identification in ROS

In this section, the `twil` ROS meta-package is detailed. This meta-package consists of an URDF description of the Twil mobile robot (`twil_description`), the implementation of some controllers for Twil (`twil_controllers`) and a ROS node for implementing the parametric identification (`twil_ident`). Although the parametric identification launch file is configured for Twil, the source code for the identification is generic and should work directly with any differential-drive mobile robot and with minor modifications for any wheeled mobile robot. Hence, in most cases, the package can be used with any robot by just adapting the launch file.

5.1 `twil_description` Package

The `twil_description` package has the URDF description of the Twil robot. Files in the `xacro` directory contains the files describing the geometric and mass parameters of the many bodies used to compose the Twil robot, while the `meshes` directory holds the STereoLithography (STL) files describing the shapes of the bodies. The files in the `launch` directory are used to load the Twil model in the ROS parameter server. The `twil.launch` file just loads the Twil model in the parameter server and is intended to be used with the actual robot, while the `twil_sim.launch` file loads the Twil model in the parameter server and launches the Gazebo simulator.

It is beyond the scope of this chapter to discuss the modeling of robots in URDF. The reader is directed to the introductory ROS references for learning the details about URDF modeling in general. However, one key point for simulating ROS controllers in Gazebo is to tell it to load the plugin for connecting with `ros_control`. In the `twil_description` package this is done in the top level URDF file, within the `<gazebo>` tag, as shown in Listing 1. See [13] for a detailed description of the plugin configuration.

Listing 1 Plugin description in `twil.urdf.xacro`.

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so" >
    <robotNamespace>/twil</robotNamespace>
    <controlPeriod>0.001</controlPeriod>
  </plugin>
</gazebo>
```

5.2 `twil_controllers` Package

The `twil_controllers` package implements the controllers for Twil. In particular, a Cartesian linearizing controller is implemented as an example of using

the results of the parametric identification. The files in the `config` directory specify the parameters for the controllers, such as the joints of the robot associated to the controller, its gains and sampling rate. The `script` directory has some useful scripts for setting the reference for the controllers and can be used for testing them. Note that although only the `CartLinearizingController` is implemented in this package, the Twil can use other controllers, as those implemented in the `ros_controllers` package. In particular, the controller used for identification (`effort_controllers/JointEffortController`) comes from the `ros_controllers` package.

The file in the `src` directory are the implementation of controllers for Twil, in particular `CartLinearizingController`, derived from the `Controller` class, while the `include` directory holds the files with the declarations of those classes. The `twil_controllers_plugins.xml` file specifies that the classes implementing the controllers are plugins for the ROS controller manager. The files in the `launch` directory are used to load and start the controllers with the respective configuration files. The detailed description of the implementation of controllers is not the scope of this chapter. See [13] for a detailed discussion about the implementation of controllers in ROS.

5.3 `twil_ident` Package

The `twil_ident` package contains a ROS node implementing the parameter identification procedure described in Sect. 2.1. Again, the source code is in the `src` directory and there is a launch file in the `launch` directory which is used to load and start the node.

The identification node can be launched through the command:

```
roslaunch twil_ident ident.launch
```

The launch file is shown in Listing 2. Initially, there are remaps of the topic names used as reference for the controllers for the right and left wheels, then, another launch file, which loads Gazebo with the Twil simulation is called. The next step is the loading of controller parameters from the configuration file `effort_control.yaml` in the parameter server and the controller manager node is spawn for loading the controllers for both wheels and the `joint_state_controller` to publish the robot state. Finally the identification node is loaded and the identification procedure starts.

Listing 2 Launch file `ident.launch`.

```
<launch>
  <remap from="/twil/left_wheel_joint_effort_controller/command"
        to="/twil/left_wheel_command"/>
  <remap from="/twil/right_wheel_joint_effort_controller/command"
        to="/twil/right_wheel_command"/>

  <include file="$(find twil_description)/launch/twil_sim.launch"/>
```

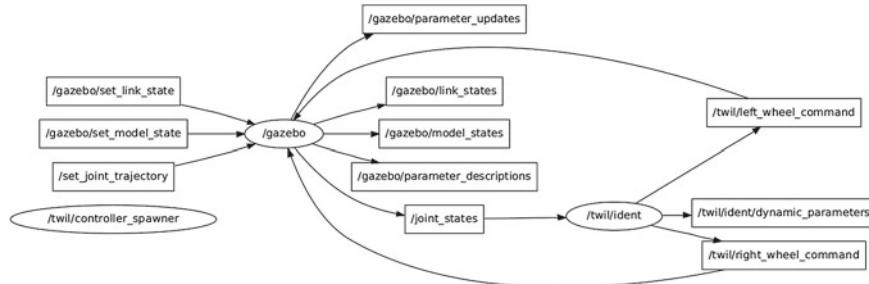


Fig. 9 Computation graph for parameter identification

```

<rosparam file="$(find twil_controllers)/config/effort_control.yaml" command="load"/>

<node name="controller_spawner" pkg="controller_manager" type="spawner" respawn="false"
      output="screen" ns="/twil"
      args="joint_state_controller left_wheel_joint_effort_controller \
            right_wheel_joint_effort_controller"/>

<node name="ident" ns="/twil" pkg="twil_ident" type="ident" output="screen">
  <remap from="ident/left_wheel_command" to="left_wheel_command"/>
  <remap from="ident/right_wheel_command" to="right_wheel_command"/>
</node>
</launch>
  
```

Figure 9 shows the computation graph used for the identification. The difference with respect to the computation graph in Fig. 8 is that torques to be applied to the wheels are published on the `/twil/right_wheel_command` and `/twil/left_wheel_command` by the `ident` node. These topics are the same described in Sect. 4, but had their names changed to force the connection between the `/gazebo` and the `/twil/ident` nodes.

The `/twil/ident` node implements the parameter identification and subscribes to the `/joint_states` topic in order to obtain the joint (wheel) velocities necessary for the identification algorithm and publishes the torque commands for the wheels in the `/twil/right_wheel_command` and `/twil/left_wheel_command` topics. The torques follow a Pseudo Random Binary Sequence (PRBS) pattern [21] in order to ensure the persistent excitation for a good identification of the parameters.

The identified parameters and the respective diagonal of the covariance matrix are published on the `/twil/ident/dynamic_parameters` topic. Those values can be used offline by controllers, for the implementation of non-adaptive controllers or can be used online to implement adaptive controllers. In this case, the controller can subscribe to the `/twil/ident/dynamic_parameters` topic to receive updates of the identified parameters and the respective diagonal of the covariance matrix, which is a measure of confidence on the parameter estimation and hence can be used to decide if the adaptation should be shut-off or not. This way, the mass and inertia parameters of the robot would be adjusted on-line for variations due to changes

in workload, for example. In order to ensure the reliability of the identified values, parameters would be changed only if the associated covariance is small enough.

The Ident class is shown in Listing 3. The private variable members of the Ident class are:

```

node_: the ROS node identifier
jointStateSubscriber: ROS topic subscriber to receive the joint state
dynParamPublisher: ROS topic to publish the identified parameters
leftWheelCommandPublisher: ROS topic to publish the reference for the
    left wheel controller
dynParamPublisher: ROS topic to publish the reference for the right wheel
    controller
nJoints: the number of joints (wheels) of the robot
u: vector of joint velocities
thetaEst1: vector of estimated parameters  $\hat{\theta}_1$ 
thetaEst2: vector of estimated parameter  $\hat{\theta}_2$ 
P1: covariance of the error in estimates  $\hat{\theta}_1$ 
P2: covariance of the error in estimates  $\hat{\theta}_2$ 
prbs: vector of PRBS sequences used as input to the robot for identification
lastTime: time for the last identification iteration.

```

Listing 3 Ident class.

```

1  class Ident
2  {
3      public:
4          Ident(ros::NodeHandle node);
5          ~Ident(void);
6          void setCommand(void);
7
8      private:
9          ros::NodeHandle node_;
10
11         ros::Subscriber jointStatesSubscriber;
12         ros::Publisher dynParamPublisher;
13         ros::Publisher leftWheelCommandPublisher;
14         ros::Publisher rightWheelCommandPublisher;
15
16         const int nJoints;
17
18         Eigen::VectorXd u;
19         Eigen::VectorXd thetaEst1;
20         Eigen::VectorXd thetaEst2;
21         Eigen::MatrixXd P1;
22         Eigen::MatrixXd P2;
23
24         std::vector<Prbs> prbs;
25
26         ros::Time lastTime;
27
28         void jointStatesCB(const sensor_msgs::JointState::ConstPtr &jointStates);
29         void resetCovariance(void);
30     };

```

The jointStatesCB() function is the callback for receiving the state of the robot and running the identifier iteration, as shown in Listing 4.

Listing 4 JointStatesCB() function.

```

1 void Ident::jointStatesCB(const sensor_msgs::JointState::ConstPtr &jointStates)
2 {
3     ros::Duration dt=jointStates->header.stamp-lastTime;
4     lastTime=jointStates->header.stamp;
5
6     Eigen::VectorXd y=-u;      //y(k+1)=(u(k+1)-u(k))/dt
7
8     Eigen::VectorXd Phi1(nJoints);
9     Eigen::VectorXd Phi2(nJoints);
10    Phi1[0]=u[1];           // u2^2(k)
11    Phi2[0]=u[0]*u[1];      // u1(k)*u2(k)
12
13    Eigen::VectorXd torque(nJoints);
14    for(int i=0;i < nJoints;i++)
15    {
16        u[i]=jointStates->velocity[i];           // u(k+1)
17        torque[i]=jointStates->effort[i];        // torque(k)
18    }
19
20    y+=u;
21    y/=dt.toSec();
22
23    Phi1[1]=torque[0]+torque[1];
24    Phi2[1]=torque[0]-torque[1];
25
26    double yEst1=Phi1.transpose()*thetaEst1;
27    Eigen::VectorXd K1=P1*Phi1/(1+Phi1.transpose()*P1*Phi1);
28    thetaEst1+=K1*(y[0]-yEst1);
29    P1-=K1*Phi1.transpose()*P1;
30
31    double yEst2=Phi2.transpose()*thetaEst2;
32    Eigen::VectorXd K2=P2*Phi2/(1+Phi2.transpose()*P2*Phi2);
33    thetaEst2+=K2*(y[1]-yEst2);
34    P2-=K2*Phi2.transpose()*P2;
35
36    std_msgs::Float64MultiArray dynParam;
37    for(int i=0;i < nJoints;i++)
38    {
39        dynParam.data.push_back(thetaEst1[i]);
40        dynParam.data.push_back(thetaEst2[i]);
41    }
42    for(int i=0;i < nJoints;i++)
43    {
44        dynParam.data.push_back(P1(i,i));
45        dynParam.data.push_back(P2(i,i));
46    }
47    dynParamPublisher.publish(dynParam);
48 }
```

In the callback, first of all the time interval since last call is computed (`dt`), then the $\phi_1(t)$ and $\phi_2(t)$ vectors are assembled in variables `Phi1` and `Phi2` and the system output $y(t + 1)$ is assembled. Then, the parameter estimates and their covariances are computed from (56)–(59) and finally, the parameter estimates and the covariance matrix diagonal are published in `dynParamPublisher`.

The `main()` function of the `ident` node is shown in Listing 5. It is just a loop running at 100 Hz publishing torques for the robot wheels through the `setCommand()` function.

Listing 5 ident node main() function.

```

1 int main(int argc,char* argv[])
2 {
3     ros::init(argc,argv,"twil_ident");
4     ros::NodeHandle node;
5
6     Ident ident(node);
7
8     ros::Rate loop(100);
9     while(ros::ok())
10    {
11        ident.setCommand();
12
13        ros::spinOnce();
14        loop.sleep();
15    }
16    return 0;
17 }
```

Torques to be applied to the robot wheels are published by the `setCommand()` function shown in Fig. 6. A PRBS signal with amplitude switching between -5 and 5 Nm is applied to each wheel.

Listing 6 setCommand() function.

```

1 void Ident::setCommand(void)
2 {
3     std_msgs::Float64 leftCommand;
4     std_msgs::Float64 rightCommand;
5     leftCommand.data=10.0*prbs[0]-5.0;
6     rightCommand.data=10.0*prbs[1]-5.0;
7     leftWheelCommandPublisher.publish(leftCommand);
8     rightWheelCommandPublisher.publish(rightCommand);
9 }
```

While the identification procedure is running, the estimates of parameters K_5 , K_6 , K_7 and K_8 and their associated covariances are published as a vector in the `/twil/ident/dynamic_parameters` topic. Hence, the results of estimation can be observed by monitoring this topic with the command:

```
rostopic echo /twil/ident/dynamic_parameters
```

The results for the estimates of parameters K_5 , K_6 , K_7 and K_8 can be viewed in Figs. 10, 11, 12 and 13, respectively. Note that for a better visualization the time horizon for Figs. 10 and 11 is not the same in all figures.

Figures 14, 15, 16 and 17 show the evolution of the diagonal of the covariance matrix related to the K_5 , K_6 , K_7 and K_8 parameters, respectively.

Although the identified values remain changing over time due to noise, it is possible to consider that they converge to an average value and stop the identification algorithm. The resulting values are shown in Table 1, with the respective diagonal of the covariance matrix. Those values were used for the implementation of the feedback linearization controller.

Given the results in Table 1 and recalling the model (36) and (37)–(40), the identified model of the Twil mobile robot is:

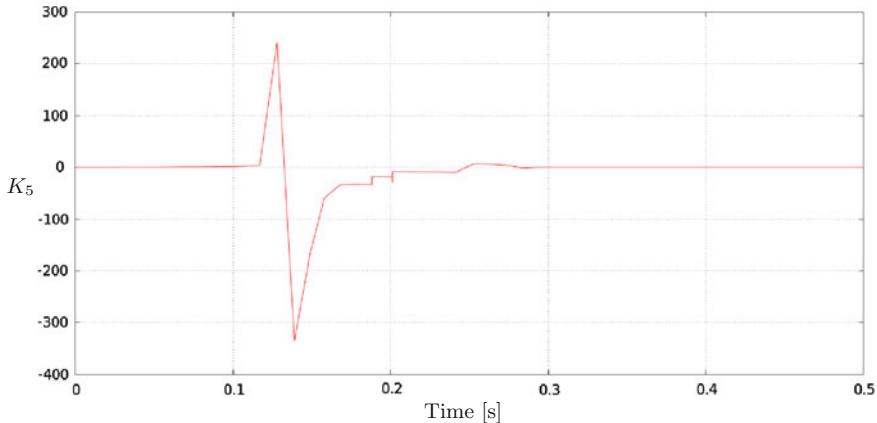


Fig. 10 Evolution of the estimate of the K_5 parameter

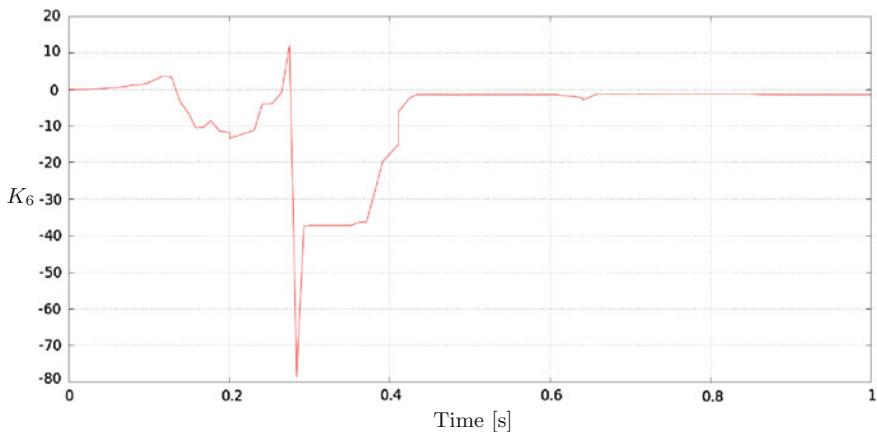


Fig. 11 Evolution of the estimate of the K_6 parameter

$$\begin{cases} \dot{x} = \begin{bmatrix} \cos \theta_c & 0 \\ \sin \theta_c & 0 \\ 0 & 1 \end{bmatrix} u \\ \dot{u} = \begin{bmatrix} 0 & 0.00431 \\ 0.18510 & 0 \end{bmatrix} \begin{bmatrix} u_1 u_2 \\ u_2^2 \end{bmatrix} + \begin{bmatrix} 18.7807 & 18.7807 \\ -14.3839 & 14.3839 \end{bmatrix} \tau \end{cases} \quad (60)$$

In principle, it sounds pointless to identify the parameters of a simulated robot. However, the simulation performed by Gazebo is based on the Open Dynamics Engine (ODE) library [23] with parameters derived from a URDF description of the robot, which is more detailed than the model used for identification and control. Due to the model non-linearities and the richness of details of the URDF description

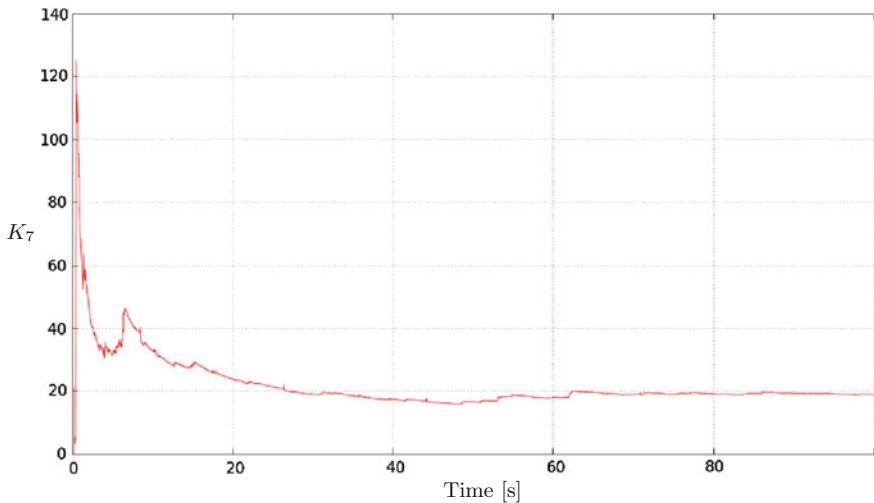


Fig. 12 Evolution of the estimate of the K_7 parameter

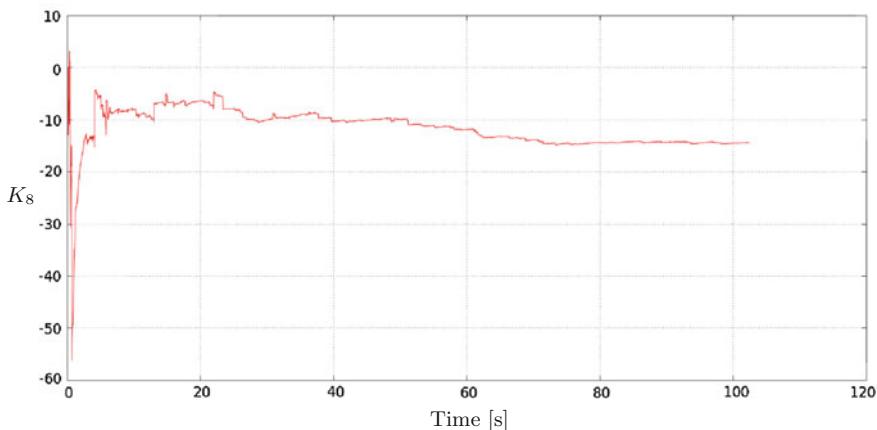


Fig. 13 Evolution of the estimate of the K_8 parameter

it is not easy to compute the equivalent parameters to be used in a closed form model useful for control design. Hence, those parameters are identified. Note that this situation is analogous to a real robot, where the actual parameters are not the same as the theoretical ones due to many details not being modeled.

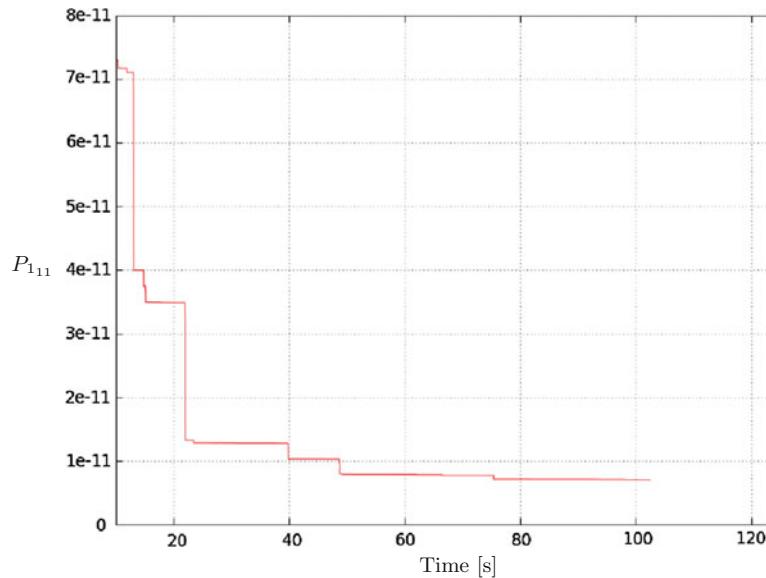


Fig. 14 Diagonal of the covariance matrix related to the K_5 parameter

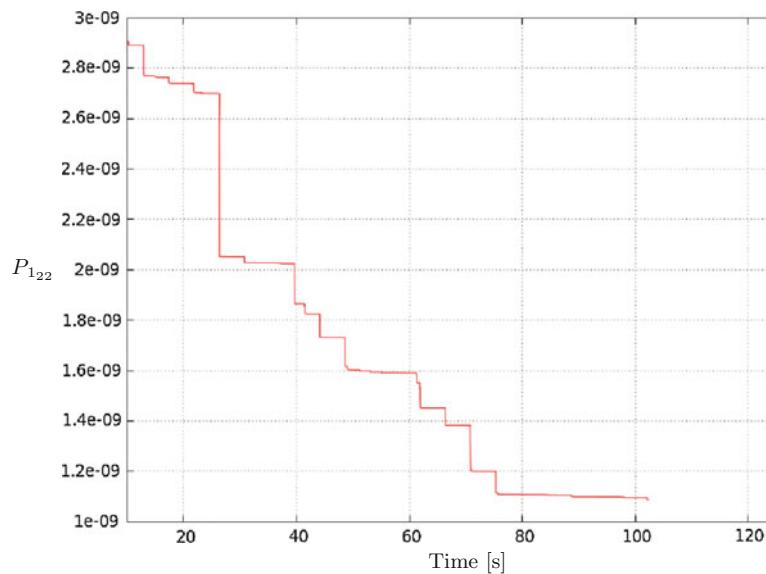


Fig. 15 Diagonal of the covariance matrix related to the K_6 parameter

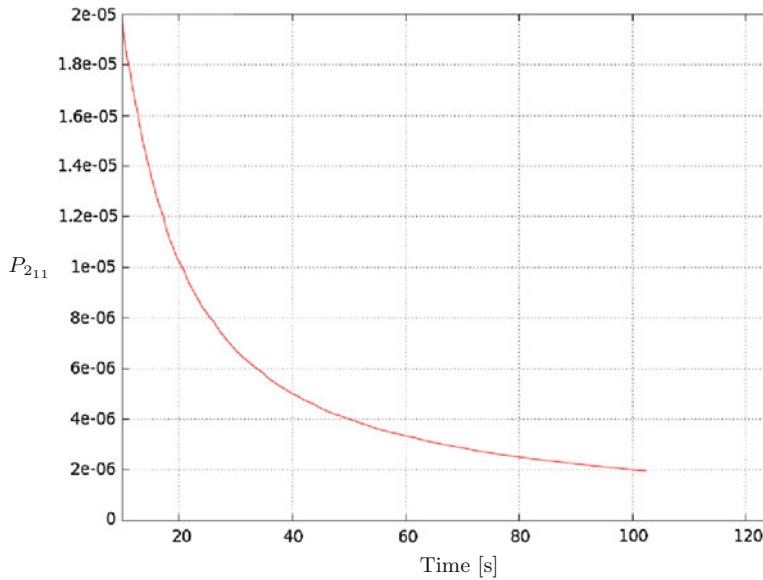


Fig. 16 Diagonal of the covariance matrix related to the K_7 parameter

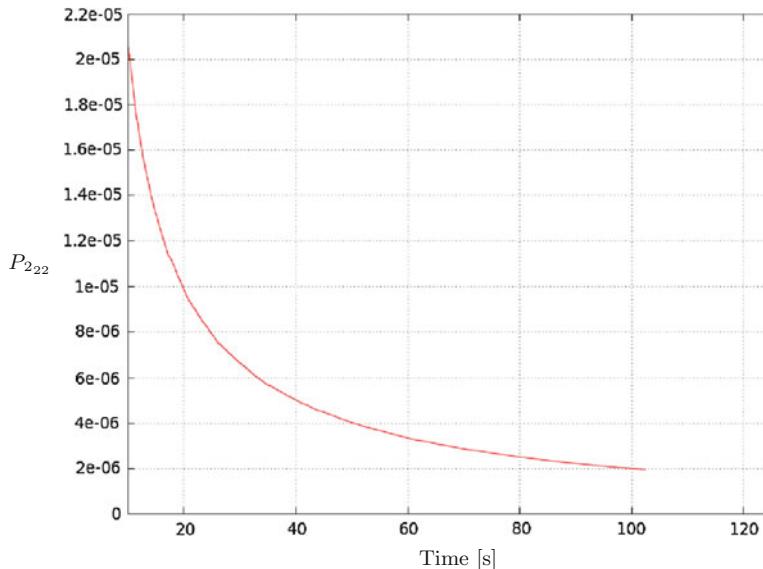


Fig. 17 Diagonal of the covariance matrix related to the K_8 parameter

Table 1 Twil parameters obtained by identification

Parameter	Value	Covariance diagonal
K_5	0.00431	7.0428×10^{-12}
K_6	0.18510	1.0870×10^{-09}
K_7	18.7807	1.9617×10^{-06}
K_8	-14.3839	1.9497×10^{-06}

6 Controller Design

The model (60) can be used for the design of controllers. Although now all parameters are known, it is still a non-linear model and a cascade of the dynamics and the kinematics as shown in Fig. 4. Also, as discussed in Sect. 1 there are in the literature many publications dealing with the control of mobile robots using only the kinematic model. Furthermore, the non-holonomic constraints associated to mobile robots, with exception of the omnidirectional ones, are associated to the first expression of (60), while the second expression is a holonomic system. For this reason, most difficulties in designing a controller for a mobile robot are related to its kinematic model and not to its dynamic model.

In order to build-up on the many methods developed to control mobile robots based on the kinematic model alone, the control strategy proposed here considers the kinematics and the dynamics of the robot in two independent steps. See [6, 7] for a control approach dealing with the complete model of the robot in a single step.

The dynamics of the robot is described by the second expression of (60) and has the form:

$$\dot{u} = f(u) + F\tau \quad (61)$$

and a state feedback linearization [11, 13] with the control law:

$$\tau = F^{-1}(\nu - f(u)) \quad (62)$$

where ν is a new control input, leads to:

$$\dot{u} = \nu \quad (63)$$

which is a linear, decoupled system. That means that each element of u is driven by a single element of ν or $\dot{u}_i = \nu_i$.

For differential-drive mobile robot such as Twil the elements of $u = [v \ \omega]^T$, are the linear and angular velocities of the robot. For other types of wheeled mobile robots, the number and meaning of the elements of u would not be the same, but (63) would still have the same form, eventually with larger vectors. Anyway, the transfer function for each element of (63) is:

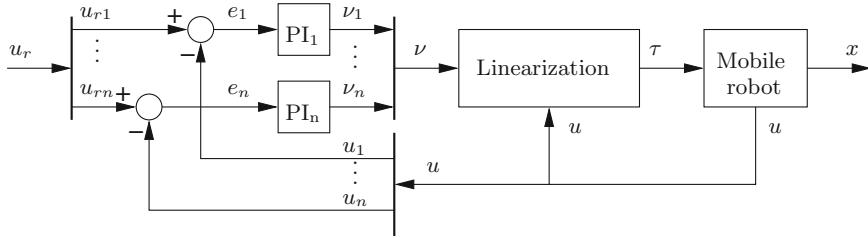


Fig. 18 Block diagram of the controller for the dynamics of the mobile robot

$$G_i(s) = \frac{U_i(s)}{V_i(s)} = \frac{1}{s} \quad (64)$$

In other words, by using the feedback (62), the system (61) is transformed in a set of independent systems, each one with a transfer function equal to $G_i(s)$. Each one of these systems can be controlled by a PI controller, then:

$$\nu_i = K_{pi}e_i + K_{ii} \int e_i dt \quad (65)$$

where $e_i = u_{ri} - u_i$, u_{ri} is the reference for the i -th element of u and K_{pi} and K_{ii} are the proportional and integral gains, respectively.

The transfer function of PI controller (65) is:

$$C_i(s) = \frac{V_i(s)}{E_i(s)} = \frac{K_{pi}s + K_{ii}}{s} \quad (66)$$

Then, by remembering that $E_i(s) = U_{ri}(s) - U_i(s)$ and using (64) and (66), it is possible to write the closed-loop transfer function as:

$$H_i(s) = \frac{U_i(s)}{U_{ri}(s)} = \frac{C_i(s)G_i(s)}{1 + C_i(s)G_i(s)} = \frac{K_{pi}s + K_{ii}}{s^2 + K_{pi}s + K_{ii}} \quad (67)$$

Figure 18 shows the block diagram of the proposed controller, which is implemented by using (62) and (65). Note that (65) can be implemented by using the `Pid` class already implemented in the `control_toolbox` ROS package, by just making the derivative gain equal to zero.

The performance of the controller is determined by the characteristic polynomial (the denominator) of (67). For canonical second order systems, the characteristic polynomial is given by:

$$s^2 + 2\xi\omega_n s + \omega_n^2 \quad (68)$$

where ξ is the damping ratio and ω_n is the natural frequency.

Hence, it is easy to see that $K_{pi} = 2\xi\omega_n$ and $K_{ii} = \omega_n^2$. Furthermore, the time it takes to the control system to settle to a precision of 1% is given by [19]:

$$T_s = \frac{-\ln 0.01}{\xi\omega_n} = \frac{4.6}{\xi\omega_n} \quad (69)$$

Therefore, by choosing the damping ration and the settling time required for each PI controller it is possible to compute K_{pi} and K_{ii} .

Again, for the Twil mobile robot and all differential-drive robots, $u = [v \omega]^T$, hence, for this type of robot there are two PI controllers: one for controlling the linear velocity and other for controlling the angular velocity. In most cases it is convenient to tune both controllers for the same performance, therefore, as the system model is the same, the controller gains would be the same. In robotics, it is usual to set $\xi = 1.0$, to avoid overshoot, and T_s is the time required for the controlled variable to converge to within 1% of error of the reference. In a first moment one may think that T_s should be set to a very small value. However, the trade-off here is the control effort. A very small T_s would require a very large ν and hence very large torques on the motors, probably above what they are able to provide. Therefore, T_s should be set to a physically sensible value. By making $T_s = 50$ ms, from (69):

$$\omega_n = \frac{4.6}{\xi T_s} = \frac{4.6}{50 \times 10^{-3}} = 92 \text{ rad/s} \quad (70)$$

and

$$K_{p1} = K_{p2} = 2\xi\omega_n = 184 \quad (71)$$

$$K_{i1} = K_{i2} = \omega_n^2 = 8464 \quad (72)$$

By using the above gains, the controls system shown in Fig. 18 ensures that u will converge to u_r in a time T_s . Then, by commanding u_r it is possible to steer the mobile robot to the desired pose. To do this in a robust way, it is necessary to have another control loop using the robot pose as feedback signal. By supposing that T_s is selected to be much faster than the pose control loop (at least five times faster), the dynamics (67) can be neglected and the resulting system (equivalent to see the system in Fig. 18 as a single block) model can be written as:

$$\dot{x} = \begin{bmatrix} \cos \theta_c & 0 \\ \sin \theta_c & 0 \\ 0 & 1 \end{bmatrix} u, \quad (73)$$

It is important to note that using (73) for the design of the pose controller is not the same as using only the kinematic model (first expression of (36)). Although the equations are the same, and then, the same control methods can be used, now there is an internal control loop (Fig. 18) that forces the commanded u_r to be effectively applied to the robot despite the dynamics of the robot.

The pose controller used here follows the one proposed in [1] and is non-linear controller based on the Lyapunov theory and a change of the robot model to polar coordinates. Also, as most controllers based on Lyapunov theory, it is assumed that the system should converge to its origin. However, as it is interesting to be able to stabilize the robot at any pose $\mathbf{x}_r = [x_{cr} \ y_{cr} \ \theta_{cr}]^T$, the following coordinate change [2] is used to move the origin of the new system to the reference pose::

$$\begin{bmatrix} \bar{x}_c \\ \bar{y}_c \\ \bar{\theta}_c \end{bmatrix} = \begin{bmatrix} \cos \theta_{cr} & \sin \theta_{cr} & 0 \\ -\sin \theta_{cr} & \cos \theta_{cr} & 0 \\ 0 & 0 & 1 \end{bmatrix} (\mathbf{x} - \mathbf{x}_r) \quad (74)$$

By using a change to polar coordinates [5] given by:

$$e = \sqrt{\bar{x}_c^2 + \bar{y}_c^2} \quad (75)$$

$$\psi = \text{atan2}(\bar{y}_c, \bar{x}_c) \quad (76)$$

$$\alpha = \bar{\theta}_c - \psi \quad (77)$$

the model (73) can be rewritten as:

$$\begin{cases} \dot{e} = \cos \alpha u_{r1} \\ \dot{\psi} = \frac{\sin \alpha}{e} u_{r1} \\ \dot{\alpha} = -\frac{\sin \alpha}{e} u_{r1} + u_{r2}. \end{cases} \quad (78)$$

which is only valid for differential-drive mobile robots. For a similar procedure for other configurations of wheeled mobile robots see [14].

Then, given a candidate to Lyapunov function:

$$V = \frac{1}{2} (\lambda_1 e^2 + \lambda_2 \alpha^2 + \lambda_3 \psi^2), \quad (79)$$

with $\lambda_i > 0$. Its time derivative is:

$$\dot{V} = \lambda_1 e \dot{e} + \lambda_2 \alpha \dot{\alpha} + \lambda_3 \psi \dot{\psi} \quad (80)$$

By replacing \dot{e} , $\dot{\alpha}$ and $\dot{\psi}$ from (78):

$$\dot{V} = \lambda_1 e \cos \alpha u_{r1} - \lambda_2 \alpha \frac{\sin \alpha}{e} u_{r1} + \lambda_2 \alpha u_{r2} + \lambda_3 \psi \frac{\sin \alpha}{e} u_{r1} \quad (81)$$

and, it can be shown that the input signal:

$$u_{r1} = -\gamma_1 e \cos \alpha \quad (82)$$

$$u_{r2} = -\gamma_2 \alpha - \gamma_1 \cos \alpha \sin \alpha + \gamma_1 \frac{\lambda_3}{\lambda_2} \cos \alpha \frac{\sin \alpha}{\alpha} \psi \quad (83)$$

leads to:

$$\dot{V} = -\gamma_1 \lambda_i e^2 \cos^2 \alpha - \gamma_2 \lambda_2 \alpha^2 \leq 0 \quad (84)$$

which, along with the continuity of V , assures the system stability. However, the convergence of system state to the origin still needs to be proved. See [22] for other choices of u_r leading to $\dot{V} \leq 0$.

Given that V is lower bounded, that V is non-increasing, as $\dot{V} \leq 0$ and that \dot{V} is uniformly continuous, as $\ddot{V} < \infty$, the Barbalat lemma [20] assures that $\dot{V} \rightarrow 0$ which implies $\alpha \rightarrow 0$ and $e \rightarrow 0$. It remains to be shown that ψ also converges to zero.

To prove that $\psi \rightarrow 0$, consider the closed loop system obtained by applying (82)–(83) to (78), given by:

$$\dot{e} = -\gamma_1 e \cos^2 \alpha \quad (85)$$

$$\dot{\psi} = -\gamma_1 \sin \alpha \cos \alpha \quad (86)$$

$$\dot{\alpha} = -\gamma_2 \alpha + \gamma_1 \frac{\lambda_3}{\lambda_2} \psi \frac{\sin \alpha}{\alpha} \cos \alpha \quad (87)$$

Given that ψ is bounded and from (86) it can be concluded that $\dot{\psi}$ is also bounded, it follows that ψ is uniformly continuous, which implies that $\dot{\alpha}$ is uniformly continuous as well, since $\ddot{\alpha} < \infty$. Then, it follows from the Barbalat lemma that $\alpha \rightarrow 0$ implies $\dot{\alpha} \rightarrow 0$. Hence, from (87) it follows that $\psi \rightarrow 0$. Therefore, (82)–(83) stabilize the system (78) at its origin.

Note that although the open loop system described by (78) has a mathematical indetermination due to the e in denominator, the closed-loop system (85)–(87) is not undetermined and hence can converge to zero. The indetermination in (78) is not due to a fundamental physical constraint as it not present in the original model (73), but was artificially created by the coordinate change (75)–(77). It is a well-known result from [8] that a non-holonomic mobile robot can not be stabilized to a pose by using a smooth, time-invariant feedback. Here, those limitations are overcome by using a discontinuous coordinate change. Also, the input signals (82)–(83) can be always computed as $\frac{\sin(\alpha)}{\alpha}$ converges to 1 as α converges to 0. Furthermore, (78) is just an intermediate theoretical step to obtain the expressions for the input signals (82)–(83). There is no need to actually compute it. If using the real robot, there is no need to use the model for simulation and if using a simulated robot, it can be simulated by the Cartesian model (73), which does not present any indetermination.

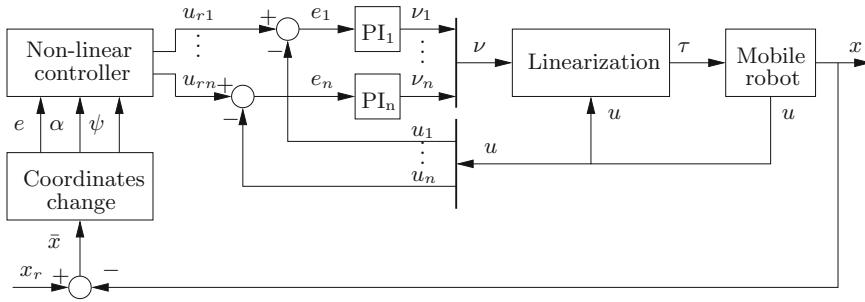


Fig. 19 Block diagram of the pose controller considering the kinematics and the dynamics of the mobile robot

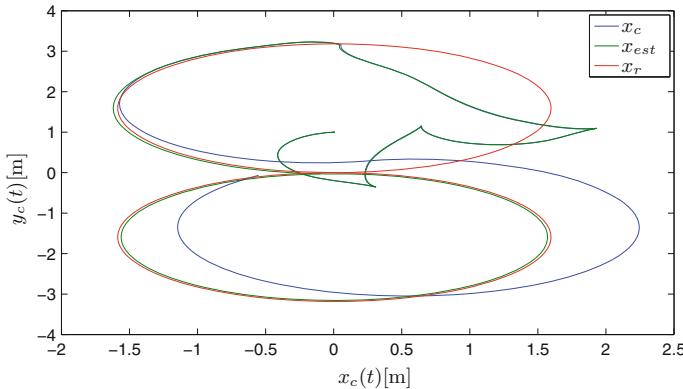


Fig. 20 Controller performance in the Cartesian plane

Figure 19 shows a block diagram of the whole pose control system, considering the kinematics and the dynamics of the robot. Note that although this control system can theoretically make the robot converge to any x_r pose departing from any pose, without a given trajectory (it is generated implicitly by the controller), in practice it is not a good idea to force x_r too far from the current robot position, as this could lead to large torque signals which can saturate the actuators. Hence, in practice x_r should be a somewhat smooth reference trajectory and the controller would force the robot to follow it.

Figure 20 shows the performance of the controller with the proposed controller while performing an 8 path. The red line is the reference path, starting at $x_r = [0 \ 0 \ 0]^T$, the blue line is the actual robot position, starting at $x_{est} = [0 \ 1 \ 0]^T$, and the read line is the robot position estimated by odometry, starting at $x_{est} = [0 \ 1 \ 0]^T$. Note that the starting position of the robot is outside the reference path and that the controller forces the convergence to the reference path and then the reference path is followed. Also note that the odometry error increases over time, but that is a pose estimation problem, which is not addressed in this chapter. The controller forces the

estimated robot position to follow the reference. Unfortunately, using just odometry, the pose estimation is not good and after some time it does not reflect the actual robot pose.

Figure 21 shows the robot orientation. Again, the red line is the reference orientation, the blue line is the actual robot orientation and the red line is the robot orientation.

An adaptive version of the controllers shown in Fig. 19 can be built, by using simultaneously the proposed controller and the identification module, as shown in Fig. 22. Then, the mass and inertia parameters of the robot would be adjusted on-line for variations due to changes in workload, for example.

Note that in this case, a PRBS pattern of torques is not necessary, as the control input generated by the controller is used. The noise and external perturbations should provide enough richness in the signal for a good identification. In extreme cases, the

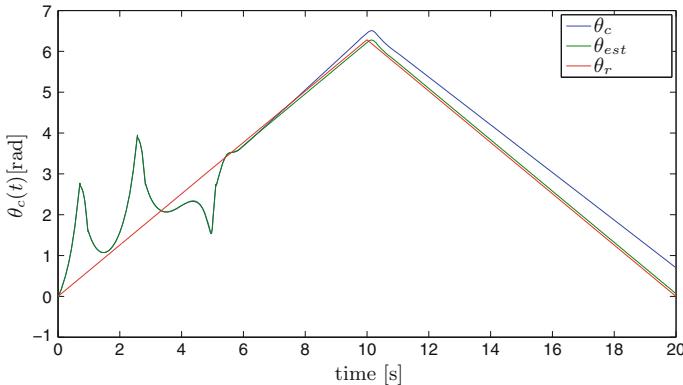


Fig. 21 Controller performance. Orientation in time

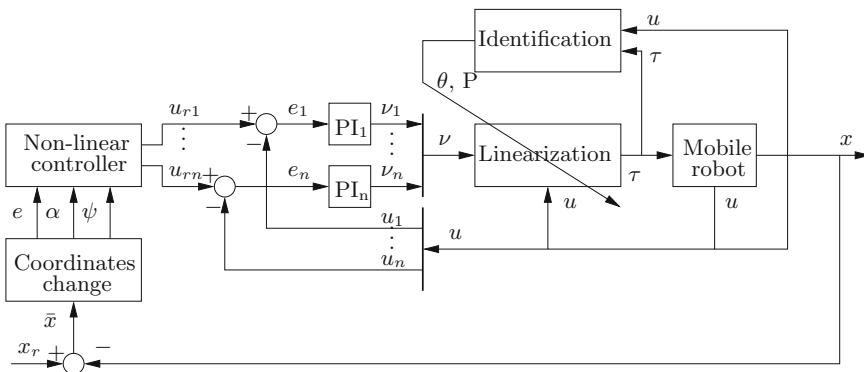


Fig. 22 Block diagram of the adaptive controller

persistence of excitation of the control signal could be tested for and the identification turned off while it is not rich enough for a good identification.

7 Conclusion

This chapter presented the identification of the dynamic model of a mobile robot in ROS. This model is the departure point for the design of advanced controllers for mobile robots. While for small robots, it is possible to neglect the dynamics and design a controller based only on the kinematic model, for larger or faster robots the controller should consider the dynamic effects. However, the theoretical determination of the parameters of the dynamic model is not easy due to the many parts of the robot and uncertainty in the assembly of the robot. Then, the online identification of those parameters enables the overcame of those difficulties.

The packages used for such identification were described and a complete example, from modeling, parameterizing of the model until the computation the of numerical values for the unknown parameters and the write down of the model with all its numerical values was shown.

The identification method was implemented as an online recursive algorithm, which enable its use in an adaptive controller, where new estimates of the parameters of the model are used to update the parameters of the controller, in a strategy known as indirect adaptive control [17].

The results of the identification procedure were used to design a controller based on the dynamics and the kinematics of the mobile robot Twil and adaptive version of that controller was proposed.

References

1. Aicardi, M., G. Casalino, A. Bicchi, and A. Balestrino. 1995. Closed loop steering of unicycle-like vehicles via lyapunov techniques. *IEEE Robotics and Automation Magazine* 2 (1): 27–35.
2. Alves, J.A.V., and W.F. Lages. 2012. Mobile robot control using a cloud of particles. In *Proceedings of 10th International IFAC Symposium on Robot Control*, pp. 417–422. International Federation of Automatic Control, Dubrovnik, Croatia. doi:[10.3182/20120905-3-HR-2030.00096](https://doi.org/10.3182/20120905-3-HR-2030.00096).
3. Åström, K.J., and B. Wittenmark. 2011. *Computer-Controlled Systems: Theory and Design*, 3rd ed., Dover Books on Electrical Engineering, Dover Publications.
4. Barrett Technology Inc. 2011. *Cambridge*. MA: WAM User Manual.
5. Barros, T.T.T., and W.F. Lages. 2012. Development of a firefighting robot for educational competitions. In *Proceedings of the 3rd International Conference on Robotics in Education*. Prague, Czech Republic.
6. Barros, T.T.T., and W.F. Lages. 2014. A backstepping non-linear controller for a mobile manipulator implemented in the ros. In *Proceedings of the 12th IEEE International Conference on Industrial Informatics*. IEEE Press, Porto Alegre, RS, Brazil.
7. Barros, T.T.T. and W.F. Lages. 2014. A mobile manipulator controller implemented in the robot operating system. In *Proceedings for the Joint Conference of 45th International Symposium*

- on Robotics and 8th German Conference on Robotics.* pp. 121–128. VDE Verlag, Munich, Germany, ISBN 978-3-8007-3601-0.
- 8. Brockett, R.W. 1982. *New Directions in Applied Mathematics*. New York: Springer.
 - 9. Campion, G., G. Bastin, and B. D'Andréa-Novel. 1996. Structural properties and classification of kinematic and dynamical models of wheeled mobile robots. *IEEE Transactions on Robotics and Automation* 12 (1): 47–62. Feb.
 - 10. Goodwin, G.C., and K.S. Sin. 1984. *Adaptive Filtering, Prediction and Control*. Prentice-Hall Information and System Sciences Series. Englewood Cliffs, NJ: Prentice-Hall Inc.
 - 11. Isidori, A. 1995. *Nonlinear Control Systems*, 3rd ed. Berlin: Springer.
 - 12. Koenig, N., and A. Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2004)*. vol. 3, pp. 2149–2154. IEEE Press, Sendai, Japan.
 - 13. Lages, W.F. 2016. Implementation of real-time joint controllers. In *Robot Operating System (ROS): The Complete Reference (Volume 1), Studies in Computational Intelligence*, vol. 625, ed. A. Koubaa, 671–702. Switzerland: Springer International Publishing.
 - 14. Lages, W.F., and E.M. Hemerly. 1998. Smooth time-invariant control of wheeled mobile robots. In *Proceedings of The XIII International Conference on Systems Science*. Technical University of Wrocław, Wrocław, Poland.
 - 15. Marder-Eppstein, E. 2016. Navigation Stack. <http://wiki.ros.org/navigation>.
 - 16. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., and K. Konolige. 2010. The office marathon: Robust navigation in an indoor office environment. In *2010 IEEE International Conference on Robotics and Automation (ICRA)*. pp. 300–307. IEEE Press, Anchorage, AK.
 - 17. Narendra, K.S., and A.M. Annaswamy. 1989. *Stable Adaptive Systems*. Englewood Cliffs, NJ: Prentice-Hall Inc.
 - 18. Nguyen-Tuong, D., and J. Peters. 2011. Model learning for robot control: a survey. *Cognitive Processing* 12(4), 319–340 (2011). <http://dx.doi.org/10.1007/s10339-011-0404-1>.
 - 19. Ogata, K. 1970. *Modern Control Engineering*. Englewood Cliffs, NJ, USA: Prentice-Hall.
 - 20. Popov, V.M. 1973. *Hyperstability of Control Systems, Die Grundlehren der mathematischen Wissenschaften*, vol. 204. Berlin: Springer.
 - 21. Press, W.H., S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing*, 2nd ed. Cambridge: Cambridge University Press.
 - 22. Secchi, H., Carelli, R., and V. Mut. 2003. An experience on stable control of mobile robots. *Latin American Applied Research* 33(4):379–385 (10 2003). http://www.scielo.org.ar/scielo.php?script=sci_arttext&pid=S0327-07932003000400003&nrm=iso.
 - 23. Smith, R. 2005. Open dynamics engine. <http://www.ode.org>.
 - 24. Sucan, I.A., and S. Chitta. 2015. MoveIt! <http://moveit.ros.org>.

Author Biography

Walter Fetter Lages graduated in Electrical Engineering at Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) in 1989 and received the M.Sc. and D.Sc. degrees in Electronics and Computer Engineering from Instituto Tecnológico de Aeronáutica (ITA) in 1993 and 1998, respectively. From 1993 to 1997 he was an assistant professor at Universidade do Vale do Paraíba (UNIVAP), from 1997 to 1999 he was an adjoint professor at Fundação Universidade Federal do Rio Grande (FURG). In 2000 he moved to the Universidade Federal do Rio Grande do Sul (UFRGS) where he is currently a full professor. In 2012/2013 he held an PostDoc position at Universität Hamburg. Dr. Lages is a member of IEEE, ACM, the Brazilian Automation Society (SBA) and the Brazilian Computer Society (SBC).

Online Trajectory Planning in ROS Under Kinodynamic Constraints with Timed-Elastic-Bands

Christoph Rösmann, Frank Hoffmann and Torsten Bertram

Abstract This tutorial chapter provides a comprehensive and extensive step-by-step guide on the ROS setup of a differential-drive as well as a car-like mobile robot with the navigation stack in conjunction with the *teb_local_planner* package. It covers the theoretical foundations of the TEB local planner, package details, customization and its integration with the navigation stack and the simulation environment. This tutorial is designated for ROS Kinetic running on Ubuntu Xenial (16.04) but the examples and code also work with Indigo, Jade and is maintained in future ROS distributions.

1 Introduction

Service robotics and autonomous transportation systems require mobile robots to navigate safely and efficiently in highly dynamic environments to accomplish their tasks. This observation poses the fundamental challenge in mobile robotics to conceive universal motion planning strategies that are applicable to different robot kinematics, environments and objectives. Online planning is preferred over offline approaches due to its immediate response to changes in a dynamic environment or perturbations of the robot motion at runtime. In addition to generating a collision free path towards the goal online trajectory optimization considers secondary objectives such as control effort, control error, clearance from obstacles, trajectory length and travel time.

The authors developed a novel, efficient online trajectory optimization scheme termed Timed-Elastic-Band (TEB) in [1, 2]. The TEB efficiently optimizes the robot trajectory w.r.t. (kino-)dynamic constraints and non-holonomic kinematics while

C. Rösmann (✉) · F. Hoffmann · T. Bertram
Institute of Control Theory and Systems Engineering, TU Dortmund University,
44227 Dortmund, Germany
e-mail: christoph.roesmann@tu-dortmund.de

F. Hoffmann
e-mail: frank.hoffmann@tu-dortmund.de

T. Bertram
e-mail: torsten.bertram@tu-dortmund.de

explicitly incorporating temporal information in order to reach the goal pose in minimal time. The approach accounts for efficiency by exploiting the sparsity structure of the underlying optimization problem formulation. In practice, due to limited computational resources online optimization usually rests upon local optimization techniques for which convergence towards the global optimal trajectory is not guaranteed. In mobile robot navigation locally optimal trajectories emerge due to the presence of obstacles. The original TEB planner is extended in [3] to a fully integrated online trajectory planning approach that combines the exploration and simultaneous optimization of multiple admissible topologically distinctive trajectories during runtime.

The complete integrated approach is implemented as an open-source package *teb_local_planner*¹ within the Robot Operating System (ROS). The package constitutes a local planner plugin for the navigation stack.² Thus, it takes advantage of the features of the established mobile navigation framework in ROS, e.g. such as sharing common interfaces for robot hardware nodes, sensor data fusion and the definition of navigation tasks (by requesting navigation goals). Furthermore, it conforms to the global planning plugins available in ROS. A video that describes the package and its utilization is available online.³ Recently, the package has been extended to accomplish navigation tasks for car-like robots (with Ackermann steering) beyond the originally considered differential-drive robots.⁴ To our best knowledge, the *teb_local_planner* is currently the only local planning package for the navigation stack that explicitly supports car-like robots with limited turning radius. The main features and highlights of the planner are:

- seamless integration with the ROS navigation stack,
- general objectives for optimal trajectory planning, such as time optimality and path following,
- explicit consideration of kino-dynamic constraints,
- applicable to general non-holonomic kinematics, such as car like robots,
- explicit exploration of distinctive topologies in case of dynamic obstacles,
- computationally efficiency for online trajectory optimization.

This chapter covers the following topics:

1. the theoretical foundations of the underlying trajectory optimization method is presented (Sect. 2),
2. description of the ROS package and its integration with the navigation stack (Sect. 3),
3. package test and parameter exploration for optimization of an example trajectory (Sect. 4),
4. modeling differential-drive and car-like robots for simulation in *stage* (Sect. 5),
5. Finally, complete navigation setup of the differential-drive robot (Sect. 6) and the car-like robot (Sect. 7).

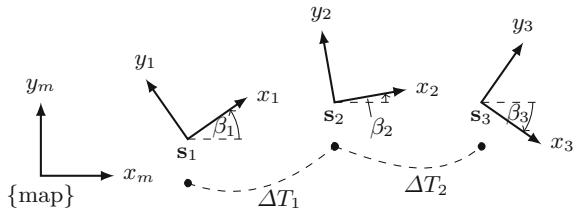
¹*teb_local_planner*, URL: http://wiki.ros.org/teb_local_planner.

²ROS navigation, URL: <http://wiki.ros.org/navigation>.

³*teb_local_planner*, online-video, URL: <https://youtu.be/e1Bw6JOgHME>.

⁴*teb_local_planner* extensions, online-video, URL: <https://youtu.be/o5wnRCzdUMo>.

Fig. 1 Discretized trajectory with $n = 3$ poses



2 Theoretical Foundations of TEB

This section introduces and explains the fundamental concepts of the TEB optimal planner. It provides the theoretical foundations for its successful utilization and customization in own applications. For a detailed description of trajectory planning with TEB the interested reader is referred to [1, 2].

2.1 Trajectory Representation and Optimization

A discretized trajectory $\mathbf{b} = [\mathbf{s}_1, \Delta T_1, \mathbf{s}_2, \Delta T_2, \dots, \Delta T_{N-1}, \mathbf{s}_N]^\top$ is represented by an ordered sequence of poses augmented with time stamps. $\mathbf{s}_k = [x_k, y_k, \beta_k]^\top \in \mathbb{R}^2 \times S^1$ with $k = 1, 2, \dots, N$ denotes the pose of the robot and $\Delta T_k \in \mathbb{R}_{>0}$ with $k = 1, 2, \dots, N - 1$ represents the time interval associated with the transition between two consecutive poses \mathbf{s}_k and \mathbf{s}_{k+1} , respectively. Figure 1 depicts an example trajectory with three poses. The reference frame of the trajectory representation and planning frame respectively is denoted as *map-frame*.⁵

Trajectory optimization seeks for a trajectory \mathbf{b}^* that constitutes a minimizer of a predefined cost function. Common cost functions capture criteria such as the total transition time, energy consumption, path length and weighted combinations of those. Admissible solutions are restricted to a feasible set for which the trajectory does not intersect with obstacles or conforms to the (kino-)dynamic constraints of the mobile robot. Improving the efficiency of solving such nonlinear programs with hard constraints has become an important research topic over the past decade. The TEB approach includes constraints as soft penalty functions into the overall cost function. The introduction of soft rather than hard constraints enables the exploitation of efficient and well studied unconstrained optimization techniques for which mature open-source implementations exist.

⁵Conventions for names of common coordinate frames in ROS are listed here: <http://www.ros.org/reps/rep-0105.html>.

The TEB optimization problem is defined such that \mathbf{b}^* minimizes a weighted and aggregated nonlinear least-squares cost function:

$$\mathbf{b}^* = \arg \min_{\mathbf{b} \setminus \{\mathbf{s}_1, \mathbf{s}_N\}} \sum_i \sigma_i f_i^2(\mathbf{b}), \quad i \in \{\mathcal{J}, \mathcal{P}\} \quad (1)$$

The terms $f_i : \mathcal{B} \rightarrow \mathbb{R}_{\geq 0}$ capture conflicting objectives and penalty functions. The set of indices associated with objectives is denoted by \mathcal{J} and the set of indices that refer to penalty functions by \mathcal{P} . The trade-off among individual terms is determined by weights σ_i . The notation $\mathbf{b} \setminus \{\mathbf{s}_1, \mathbf{s}_N\}$ indicates that start pose $\mathbf{s}_1 = \mathbf{s}_s$ and goal pose $\mathbf{s}_N = \mathbf{s}_g$ are fixed and hence not subject to optimization. In the cost function, \mathbf{s}_1 and \mathbf{s}_N are substituted by the current robot pose \mathbf{s}_s and desired goal pose \mathbf{s}_g .

The TEB optimization problem (1) is represented as a hyper-graph in which poses \mathbf{s}_k and time intervals ΔT_k denote the vertices of the graph and individual cost terms f_i define the (hyper-)edges. The term *hyper* indicates that an edge connects an arbitrary number of vertices, in particular temporal related poses and time intervals. The resulting hyper-graph is efficiently solved by utilizing the *g2o*-framework⁶ [4]. The interested reader is referred to [2] for a detailed description on how to integrate the *g2o*-framework with the TEB approach. The formulation as hyper-graph benefits from both the direct capture of the sparsity structure for its exploitation within the optimization framework and its modularity which easily allows incorporation of additional secondary objectives f_k .

Before the individual cost terms f_k are described, the approximation of constraints by penalty functions is introduced. Let \mathcal{B} denote the entire set of trajectory poses and time intervals such that $\mathbf{b} \in \mathcal{B}$. The inequality constraint $g_i(\mathbf{b}) \geq 0$ with $g_i : \mathcal{B} \rightarrow \mathbb{R}$ is approximated by a positive semi-definite penalty function $p_i : \mathcal{B} \rightarrow \mathbb{R}_{\geq 0}$ which captures the degree of violation:

$$p_i(\mathbf{b}) = \max\{\mathbf{0}, -g_i(\mathbf{b}) + \epsilon\} \quad (2)$$

The parameter ϵ adds a margin to the inequality constraint such that the cost only vanishes for $g_i(\mathbf{b}) \geq \epsilon$. Combining indices of inequality constraints g_i respectively penalty functions p_i into the set \mathcal{P} results in the overall cost function (1) by assigning $f_i(\mathbf{b}) = p_i(\mathbf{b})$, $\forall i \in \mathcal{P}$. It is assumed that the choice of $g_i(\mathbf{b})$ preserves a continuous derivative (C^1 -differentiability) of $p_i^2(\mathbf{b})$ and that $g_i(\mathbf{b})$ adheres to eligible monotonicity or convexity constraints.

In order to guarantee the true compliance of a solution with the constraint $g_i(\mathbf{b}) \geq 0$ by means of (2) the corresponding weights in the overall objective function (1) are required to tend towards infinity $\sigma_i \rightarrow \infty$, $\forall i \in \mathcal{P}$. For a comprehensive introduction to the theory of penalty methods the reader is referred to [5]. On the other hand, large weights prevent the underlying solver to converge properly as they cause the optimization problem to become numerically ill-conditioned. Hence, the

⁶*libg2o*, URL: <http://wiki.ros.org/libg2o>.

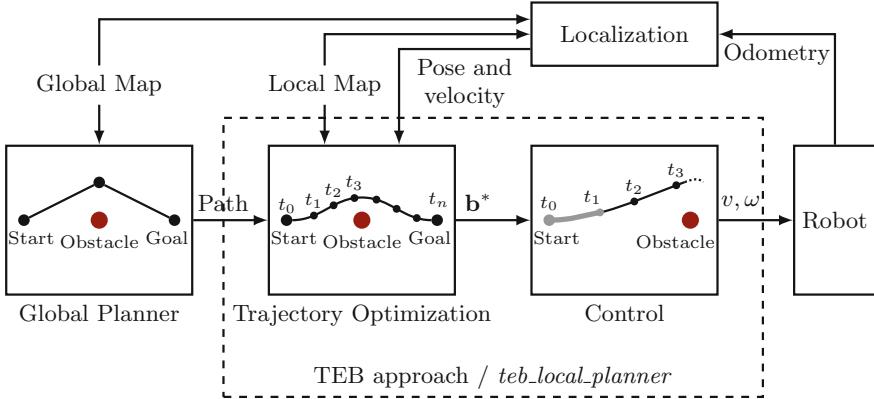


Fig. 2 System setup of a robot controlled by the TEB approach

TEB approach compensates the true minimizer with a suboptimal but computationally more efficiently obtained solution with user defined weights and the specification of an additional margin ϵ . The ROS implementation provides the parameter `penalty_epsilon` that specifies ϵ .

The TEB method utilizes multiple cost terms f_i for e.g. obstacle avoidance, compliance with (kino-)dynamic constraints of mobile robots and visiting of via-points. The list of currently implemented cost terms is described in the ROS package description in Sect. 3.3.

2.2 Closed-Loop Control

Figure 2 shows the general control architecture with the local TEB planner. The optimization scheme for (1) starts with an initial solution trajectory generated from the path provided by the global planner w.r.t. a static representation of the environment (global map). Instead of a tracking controller which regulates the motion along the planned optimal trajectory \mathbf{b}^* , a predictive control scheme is applied in order to account for dynamic environments encapsulated in the local map and to allow the refinement of the trajectory during runtime. Thus, optimization problem (1) is solved repeatedly w.r.t. the current robot pose and velocity. The current position of the robot is usually obtained from a localization scheme. Within each sampling interval⁷ only the first control action of the TEB is commanded to the robot, which is the basic idea in model predictive control [6]. As most robots are velocity controlled by their base controllers, low-level hardware interfaces accept translational and angular velocity components w.r.t. the robot base frame. These components are easily extracted from

⁷The `move_base` node (navigation stack) provides a parameter `controller_frequency` to adjust the sampling interval.

the optimal trajectory \mathbf{b}^* by investigating finite differences on the position and orientation part. Car-like robots often require the steering angle rather than angular velocity. The corresponding steering angle is calculated from the turn rate and the car-like kinematic model.

In order to improve computational efficiency the trajectory optimization pursues a warm start approach. Trajectories generated in previous iterations are reused as initial solutions in subsequent sampling intervals with updated start and goal poses. Since the time differences ΔT_k are subject to optimization the resolution of the trajectory is adjusted at each iteration according to an adaptation rule. If the resolution is too high, overly many poses increase the computational load of the optimization. On the other hand, if the resolution is too low, the finite difference approximations of quantities related to the (kino-)dynamic model of the robot are no longer accurate, causing a degradation of navigation capabilities. Therefore, the approach accounts for changing magnitudes of ΔT by regulating the resolution towards a desired temporal discretization ΔT_{ref} (ROS parameter `dt_ref`). In case of low resolution $\Delta T_k > \Delta T_{ref} + \Delta T_{hyst}$ an additional pose and time interval are filled in between \mathbf{s}_k and \mathbf{s}_{k+1} . In case of inflated resolution $\Delta T_k < \Delta T_{ref} - \Delta T_{hyst}$ pose \mathbf{s}_{k+1} is removed. The hysteresis specified by ΔT_{hyst} (ROS parameter `dt_hyst`) avoids oscillations in the number of TEB states. In case of a static goal pose this adaption implies a shrinking horizon since the overall transition time decreases as the robot advances towards to the goal.

Algorithm 1 Online TEB feedback control

```

1: procedure TEBALGORITHM( $\mathbf{b}, \mathbf{x}_s, \mathbf{x}_g, \mathcal{O}, \mathcal{V}$ )            $\triangleright$  Invoked each sampling interval
2:   Initialize or update trajectory
3:   for all Iterations 1 to  $I_{teb}$  do
4:     Adjust length  $n$  of the trajectory
5:     Build/update hyper-graph incl. association of obstacles  $\mathcal{O}$  and via-points  $\mathcal{V}$  with poses of
       the trajectory
6:      $\mathbf{b}^* \leftarrow \text{CALLOPTIMIZER}(\mathbf{b})$             $\triangleright$  solve (1), e.g. with libg2o
7:     Check feasibility
return First (sub-) optimal control inputs ( $v_1, \omega_1$ )

```

The major steps performed at each sampling interval are captured by Algorithm 1. The loop starting at line 3 is referred to as the outer optimization loop, which adjusts the length of the trajectory as described above and associates the current set of obstacles \mathcal{O} and via-points \mathcal{V} with their corresponding states \mathbf{s}_k of the current trajectory. Further information on obstacles and via-points is provided in Sects. 3.3 and 3.5. The loop is repeated I_{teb} times (ROS parameter `no_outer_iterations`). The actual solver for optimization problem (1) is invoked in line 6 which itself performs multiple solver iterations. The corresponding ROS parameter for the number of iterations of the inner optimization loop is `no_inner_iterations`. The choice of these parameters significantly influences the required computation time as well as the convergence properties. After obtaining the optimized trajectory \mathbf{b}^* a feasibility check

is performed that verifies if the first M poses actually are collision free based on their original footprint model defined in the navigation stack (note, this is not the footprint model used for optimization as presented in Sect. 3.4). The verification horizon M is represented by the ROS parameter `feasibility_check_no_poses`.

2.3 Planning in Distinctive Topologies

The previously introduced TEB approach and its closed-loop application are subject to local optimization schemes which might cause the robot to get stuck in local minima. Local minima often emerge due to the presence of obstacles. Identifying those local minima coincides with analyzing distinctive topologies between start and goal poses. For instance the robot either chooses the left or right side in order to circumnavigate an obstacle. Our TEB ROS implementation investigates the discovery and optimization of multiple trajectories in distinctive topologies and selects the best candidate for control at each sampling interval. The equivalence relation presented in [7] determines whether two trajectories share the same topology. However, the configuration and theory of this extension is beyond the scope of this tutorial. The predefined default parameters are usually appropriate for applications as presented in the following sections. For further details the reader is referred to [3].

3 The `teb_local_planner` ROS Package

This section provides an overview about the `teb_local_planner` ROS package which implements the TEB approach for online trajectory optimization as described in Sect. 2.

3.1 Prerequisites and Installation

In order to install and configure the `teb_local_planner` package for a particular application, observe the following limitations and prerequisites:

- Although online trajectory optimization approaches pursue mature computational efficiency, their application still consumes substantial CPU resources. Depending on the desired trajectory length respectively resolution as well as the number of considered obstacles, common desktop computers or modern notebooks usually cope with the computational burden. However, older systems and embedded systems might not be capable to perform trajectory optimization at a reasonable rate.

- Results and discussions on stability and optimality properties for online trajectory optimization schemes are widespread in the literature, especially in the field of model predictive control. However, since these results are often theoretically and the planner is confronted with e.g. sensor noise and dynamic environments in real applications, finding a feasible and stable trajectory in every conceivable scenario cannot be guaranteed. However, the planner tries to detect and resolve failures to generate a feasible trajectory by post-introspection of the optimized trajectory. Its ongoing algorithmic improvement is subject to further investigations.
- The package currently supports differential-drive, car-like and omnidirectional robots. Since the planner is integrated with the navigation stack as plugin it provides a `geometry_msgs/Twist` message containing the velocity commands for controlling the robots motion. Since the original navigation stack is not intended for car-like robots yet, the additional recovery behaviors must be turned off and the global planner is expected to provide appropriate plans. However, the default global planners work well for small and medium sized car-like robots as long as the environment does not contain long and narrow passages unless the length of the vehicle exceeds their width. A conversion to steering angle has to be applied in case the car-like robot only accepts a steering angle rather than the angular velocity and interprets the `geometry_msgs/Twist` or `ackermann_msgs/AckermannDriveStamped` message different from the nominal convention. The former is directly enabled (see Sect. 6) and the latter requires a dedicated conversion ROS node.
- The oldest officially supported ROS distribution is Indigo. At the time of writing the planner is also available in Jade and Kinetic. Support of future distributions is expected. The package is released for both default and ARM architectures.
- Completion of the common ROS beginner tutorials, e.g. being aware of navigating the filesystem, creating and building packages as well as dealing with `rviz`,⁸ launch files, topics, parameters and `yaml` files is essential. Experiences with the navigation stack are highly recommended. The user should be familiar with concepts and components of ROS navigation such as local and global costmaps and local and global planners (`move_base` node), coordinate transforms, odometry and localization. This tutorial outlines the configuration of a complete navigation setup. However, explanation of the underlying concepts in detail is beyond the scope of this chapter. Tutorials on ROS navigation are available at the wiki page² and [8].
- Table 1 provides an overview of currently available local planners for the ROS navigation stack and summarizes its main features.

The `teb_local_planner` is easily installed from the official ROS repositories by invoking in *terminal*:

```
$ sudo apt-get install ros-kinetic-teb-local-planner
```

⁸`rviz`, URL: <http://wiki.ros.org/rviz>.

Table 1 Comparison of available local planners in the ROS navigation stack

	EBand ^a	TEB	DWA ^b
Strategy	Force-based path deformation and path following controller	Continuous trajectory optimization resp. predictive controller	Sampling-based trajectory generation, predictive controller
Optimality	Shortest path without considering kinodynamic constraints (local solutions)	Time-optimal (or ref. path fidelity) with kinodynamic constraints (multiple local solutions, parallel optimization)	Time-sub-optimal with kinodynamic constraints, samples of trajectories with constant curvature for prediction (multiple local solutions)
Kinematics	Omnidirectional and differential-drive robots	Omnidirectional, differential-drive and car-like robots	Omnidirectional and differential-drive robots
Computational burden	Medium	High	Low/Medium

^a*eband_local_planner*, URL: http://wiki.ros.org/eband_local_planner

^b*TrajectoryPlannerROS*, URL: http://wiki.ros.org/base_local_planner

The distribution name `kinetic` might be adapted to match the currently installed one. In the following, terminal commands are usually indicated by a leading \$-sign. As an alternative to the default package installation, recent versions (albeit experimental) can be obtained and compiled from source:

```
$ cd ~/catkin_ws/src
2 $ git clone https://github.com/rst-tu-dortmund/
     teb_local_planner.git --branch kinetic-devel
$ cd ../
4 $ rosdep install --from-paths src --ignore-src --rosdistro
     kinetic -y
$ catkin_make
```

Hereby, it is assumed that `~/catkin_ws` points to the user-created *catkin* workspace.

3.2 Integration with ROS Navigation

The `teb_local_planner` package seamlessly integrates with the ROS navigation stack since it complies with the interface `nav_core::BaseLocalPlanner` specified in the `nav_core`² package. Figure 3 shows an overview of the main components that constitute the navigation stack and the `move_base` node respectively.⁹ The `move_base` node takes care about the combination of the global and local planner as well as

⁹Adopted from the `move_base` wiki page, URL: http://wiki.ros.org/move_base.

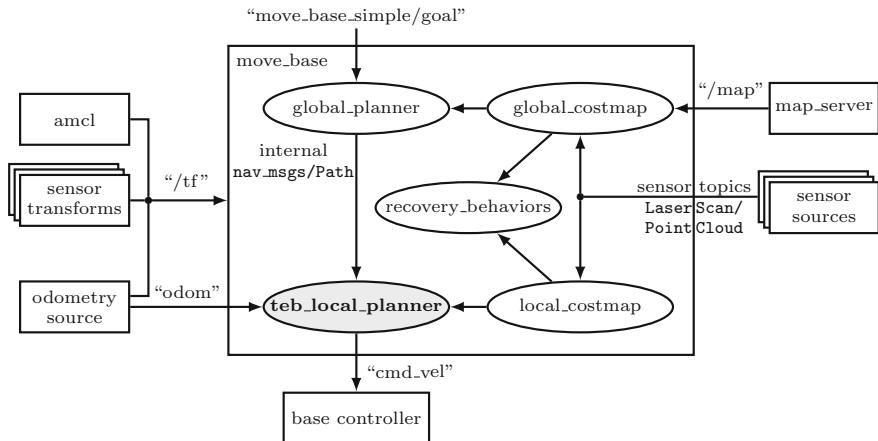


Fig. 3 ROS navigation component-view including `teb_local_planner`

handling costmaps for obstacle avoidance. The `amcl` node¹⁰ provides an adaptive monte carlo localization algorithm which corrects the accumulated odometric error and localizes the robot w.r.t. the global map. For the following tutorials the reader is expected to be familiar with the navigation stack components and the corresponding topics.

The `teb_local_planner` package comes with its own parameters which are configurable by means of the parameter server. The full list of parameters is available on the package wiki page, but many of them are presented and described in this tutorial. Parameters are set according to the relative namespace of `move_base`, e.g. `/move_base/TebLocalPlannerROS/param_name`. Most of the parameters can also be configured during runtime with `rqt_reconfigure` which is instantiated as follows (assuming a running planner instance):

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Within each local planner invocation (respectively each sampling interval) the `teb_local_planner` chooses an intermediate virtual goal within a specified lookahead distance on the current global plan. Only the local stretch of the global plan between current pose and lookahead point is subject to trajectory optimization by means of Algorithm 1. Hence the lookahead distance implies a receding horizon control strategy which transits to a shrinking horizon once the virtual goal coincides with the final goal pose of the global plan. The lookahead distance to the virtual goal is set by parameter `max_global_plan_lookahead_dist` but the virtual goal is never located beyond the boundaries of the local costmap.

¹⁰`amcl`, URL: <http://wiki.ros.org/amcl>.

3.3 Included Cost Terms: Objectives and Penalties

The *teb_local_planner* determines the current control commands in terms of minimizing the future trajectory w.r.t. a specified cost function (1) which itself consists of aggregated objectives and penalty terms as described in Sect. 2. Currently implemented cost terms f_i of the optimization problem (1) are summarized in the following overview including their corresponding ROS parameters such as the optimization weights σ_i .

Limiting translational velocity (Penalty)

Description: Constrains the translational velocity v_k to the interval $[-v_{back}, v_{max}]$. v_k is computed with s_k , s_{k+1} and ΔT_k using finite differences.

Weight parameter: weight_max_vel_x

Additional parameters: max_vel_x (v_{max}), max_vel_x_backwards (v_{back})

Limiting angular velocity (Penalty)

Description: Constrains the angular velocity to $|\omega_k| \leq \omega_{max}$ (finite differences).

Weight parameter: weight_max_vel_theta

Additional parameters: max_vel_theta (ω_{max})

Limiting translational acceleration (Penalty)

Description: Constrains the translational acceleration to $|a_k| \leq a_{max}$ (finite differences).

Weight parameter: weight_acc_lim_x

Additional parameters: acc_lim_x (a_{max})

Limiting angular acceleration (Penalty)

Description: Constrains the angular acceleration to $|\dot{\omega}_k| \leq \dot{\omega}_{max}$ (finite differences).

Weight parameter: weight_acc_lim_theta

Additional parameters: acc_lim_theta ($\dot{\omega}_{max}$)

Compliance with non-holonomic kinematics (Objective)

Description: Minimize deviations from the geometric constraint that requires two consecutive poses s_k and s_{k+1} to be located on a common arc of constant curvature. Actually, kinematic compliance is not merely an objective, but rather an equality constraint. However, since as the planner rests upon unconstrained optimization a sufficient compliance is ensured by a large weight.

Weight parameter: weight_kinematics_nh

Limiting the minimum turning radius (Penalty)

Description: Some mobile robots exhibit a non-zero turning radius (e.g. implicated by a limited steering angle). In particular car-like robots are unable to rotate in place. This penalty term enforces $r = \frac{v_k}{\omega_k} \geq r_{min}$. Differential drive and unicycle robots can turn in place $r_{min} = 0$.

Weight parameter: weight_kinematics_turning_radius

Additional parameters: min_turning_radius (r_{min})

Penalizing backwards motions (Penalty)

Description: This cost term expresses preference for forward motions independent of the actual maximum backward velocity v_{back} in terms of a bias weight. The penalty is deactivated if `min_turning_radius` is non-zero.

Weight parameter: `weight_kinematics_forward_drive`

Obstacle avoidance (Penalty)

Description: This cost term maintains a minimum separation d_{min} of the trajectory from obstacles. A dedicated robot footprint model is taken into account for distance calculation (see Sect. 3.3).

Weight parameter: `weight_obstacle`

Additional parameters: `min_obstacle_dist` (d_{min})

Via-points (Objective)

Description: This cost term minimizes the distance to via-points, e.g. located along the global plan. Each via-point defines an attractor for the planned trajectory.

Weight parameter: `weight_viapoint`

Additional parameters: `global_plan_viapoint_sep`

Arrival at the goal in minimum time (Objective)

Description: This term minimizes ΔT_k in order seek for a time-optimal trajectory.

Weight parameter: `weight_optimaltime`.

3.4 Robot Footprint for Optimization

The obstacle avoidance penalty function introduced in Sect. 3.3 depends on a dedicated robot footprint model. The reason behind not using the original footprint specified in the navigation stack resp. costmap configuration is to promote efficiency in the optimization formulation while keeping the original footprint for feasibility checks. Since the optimization scheme is subject to a large number of distance calculations between robot and obstacles, the original polygonal footprint would drastically increase the computational load, as each polygon edge has to be taken into account. However, the user might still duplicate the original footprint model for optimization, but in practice simpler approximations are often sufficient. The current package version provides four different models (see Fig. 4). Parameters for defining the footprint model (as listed below) are defined w.r.t. the robot base frame, e.g. `base_link`, such that s_k defines its origin. An example robot frame is depicted in Fig. 4d. In particular, the four models are:

- **Point model:** The most efficient representation in which the robot is modeled as a single point. The robot's radial extension is captured by inflating the minimum distance from obstacles d_{min} (`min_obstacle_dist`, see Fig. 4a) by the robot's radius.
- **Line model:** The line model is ideal for robots which dimensions differ in longitudinal and lateral directions. Start and end of the underlying line segment, $\mathbf{l}_s \in \mathbb{R}^2$

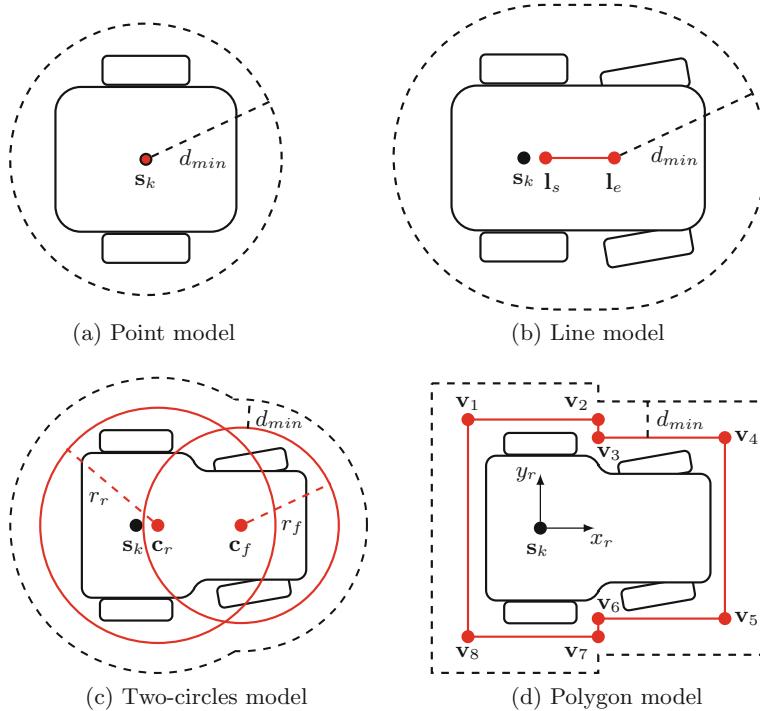


Fig. 4 Available footprint models for optimization

and $\mathbf{l}_e \in \mathbb{R}^2$ respectively, are arbitrary w.r.t. the robot's center of rotation \mathbf{s}_k as origin $(0, 0)$. The robot's radial extension is controlled similar to the point model by inflation of d_{min} (refer to Fig. 4b).

- **Two-circle model:** The two-circle model is suited for robots that exhibit a more cone-shaped footprint rather than a rectangular one (see Fig. 4c). The centers of both circles, \mathbf{c}_r and \mathbf{c}_f respectively, are restricted to be located on the robot's x -axis. Their offsets w.r.t. the center of rotation \mathbf{s}_k and their radii r_r and r_f are arbitrary.
- **Polygon model:** The polygon model is the most general one, since the number of edges is arbitrary. Figure 4d depicts a footprint defined by 8 vertices $\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_8 \in \mathbb{R}^2$. The polygon is automatically completed by adding an edge between the first and last vertex.

The following `yaml` file contains example parameter values for customizing the footprint. All parameters are defined w.r.t. the local planner namespace `TebLocalPlannerROS`:

```

TebLocalPlannerROS:
2   footprint_model:
      type: "point" # types: "point", "line", "two_circles",
      "polygon"
4     line_start: [-0.3, 0.0] # for type "line"
5     line_end: [0.3, 0.0] # for type "line"
6     front_offset: 0.2 # for type "two_circles"
7     front_radius: 0.2 # for type "two_circles"
8     rear_offset: 0.2 # for type "two_circles"
9     rear_radius: 0.2 # for type "two_circles"
10    vertices: [ [-0.1,0.2], [0.2,0.2], [0.2,-0.2], [-0.1,-0.2] ]
      # for type "polygon"

```

3.5 Obstacle Representations

The *teb_local_planner* takes the local costmap into account as already stated in Sect. 3.2. The costmap mainly consists of a grid in which each cell stores an 8-bit cost value that determines whether the cell is free (0), unknown, undesired or occupied (255). Besides, the ability to implement multiple layers and to fuse data from different sensor sources, the costmap is perfectly suited for local planners in the navigation stack due to their sampling based nature. In contrast, the TEB optimization problem (1) cannot just test discrete cell states inside its own cost function for collisions, but rather requires continuous functions based on the distance to obstacles. Therefore, our implementation extracts relevant obstacles from the current costmap at the beginning of each sampling interval and considers each occupied cell as single dimensionless point-shaped obstacle. Hence, the computation time strongly depends on the local costmap size and resolution (see Sect. 6). Additionally, custom obstacles can be provided by a specific topic. The *teb_local_planner* supports obstacle representations in terms of points, lines and closed polygons. A costmap conversion¹¹ might be activated in order to convert costmap cells into primitive types such as lines and polygons in a separate thread. However, these extensions are beyond the scope of this introductory tutorial, but the interested reader is referred to the package wiki page.

Once obstacles are extracted and cost terms (hyper-edges) according to Sect. 3.3 are constructed, obstacles are associated with the discrete poses s_k of the trajectory in order to maintain a minimal separation.

In order to speed up the time spent by the solver for computing the cost function multiple times during optimization, each pose s_k is only associated with its nearest obstacles. The association is renewed at every outer iteration (refer to Algorithm 1) in order to correct vague associations during convergence. For advanced parameters of the association strategy the reader is referred to the *teb_local_planner* ROS wiki page. Figure 5 depicts an example planning scenario in which the three

¹¹*costmap_converter*, URL: http://wiki.ros.org/costmap_converter.

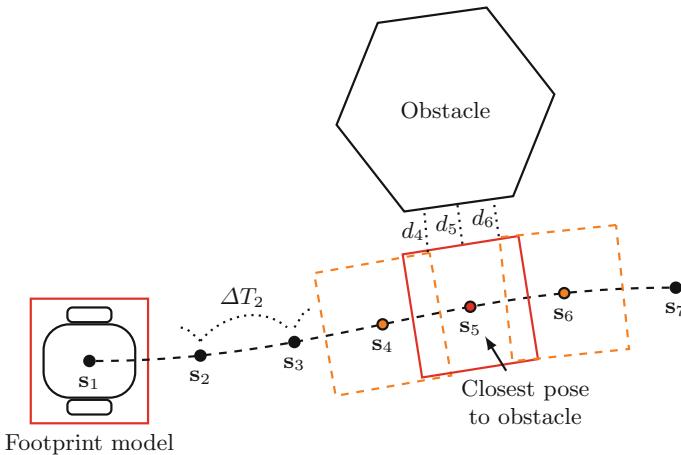


Fig. 5 Association between poses and obstacles

closest poses are associated with the polygonal obstacle. Notice, the minimum distances d_4 , d_5 and d_6 to the robot footprints located at s_4 , s_5 and s_6 are constrained to `min_obstacle_dist`. The minimum distance should account for an additional safety margin around the robot, since the penalty functions cannot guarantee its fulfillment and small violations might cause a rejection of the trajectory by the feasibility check (refer to Sect. 2).

4 Testing Trajectory Optimization

Before starting with the configuration process of the `teb_local_planner` for a particular robot and application, we recommend the reader to familiarize himself with the optimization process and furthermore to check the performance on the target hardware. The package includes a simple test node (`test_optim_node`) that optimizes a trajectory between a fixed start and goal pose. Some obstacles are included with interactive markers¹² which are conveniently moved via the GUI in `rviz`.

Launch the `test_optim_node` in combination with a preconfigured `rviz` node as follows:

```
$ roslaunch teb_local_planner test_optim_node.launch
```

An `rviz` window should open showing the trajectory and obstacles. Select the menu button *Interact* in order to move the obstacles around. An example setting is depicted in Fig. 6a. As briefly stated in Sect. 2, the package generates and optimizes trajectories in different topologies in parallel. The currently selected trajectory for navigation is

¹²*interactive_markers*, URL: http://wiki.ros.org/interactive_markers.

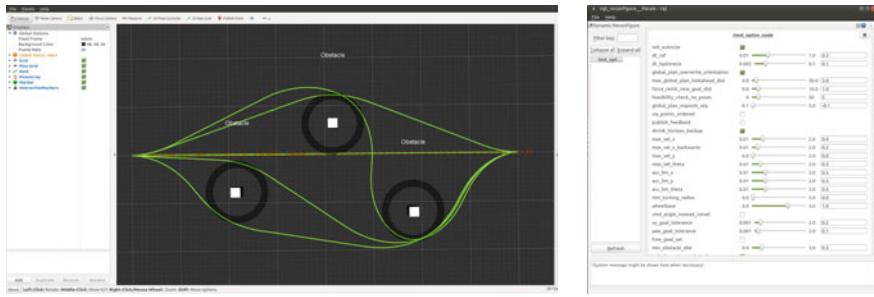


Fig. 6 Testing trajectory optimization with *test_optim_node*

augmented with red pose arrows in visualization. In order to change parameters during runtime, invoke

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

in a new terminal window and select *test_optim_node* from the list of available nodes (refer to Fig. 6b). Try to customize the optimization with different parameter settings. Since some parameters significantly influence the optimization result, adjustments should be performed slightly and in a step-by-step manner. In case you encounter a poor performance on your target system even with the default settings, try to decrease parameters *no_inner_iterations*, *no_outer_iterations* or increase *dt_ref* slightly.

5 Creating a Mobile Robot in Stage Simulator

This section introduces a minimal *stage*¹³ simulation setup with a differential-drive and a car-like robot. *stage* is chosen for this tutorial since it constitutes a package which is commonly used in ROS tutorials and is thus expected to be available in future ROS distributions as well. Furthermore, *stage* is fast and lightweight in terms of visualization which allows its execution even on slow CPUs and older graphic cards. It supports kinematic models for differential-drive, car-like and holonomic robots, but it is not intended to perform dynamic simulations such as *Gazebo*.¹⁴ However, even if the following sections refer to a stage model, the procedures and configurations are directly applicable to other simulation environments or a real mobile robot without major modifications.

¹³*stage_ros*, URL: http://wiki.ros.org/stage_ros.

¹⁴*gazebo_ros_pkgs*, URL: http://wiki.ros.org/gazebo_ros_pkgs.

Note *stage* (resp. *stage_ros*) publishes the coordinate transformation between *odom* and *base_link* and the odometry information to the *odom* topic. It subscribes to velocity commands by the topic *cmd_vel*.

Make sure to install *stage* for your particular ROS distribution:

```
$ sudo apt-get install ros-kinetic-stage-ros
```

In order to gather all configuration and launch files that will be created during the tutorial, a new package is initiated as follows:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg teb_tutorial
```

It is a good practice to add *teb_local_planner* and *stage_ros* to the run dependencies of your newly created package (check the new *package.xml* file). Now, create a *stage* and a *maps* folder inside the *teb_tutorial* package and download a predefined map called *maze.png*¹⁵ and its *yaml* configuration for the *map_server*:

```
$ roscd teb_tutorial
$ mkdir stage && mkdir maps
$ cd maps
# remove any whitespaces in the URLs below after copying
$ wget https://cdn.rawgit.com/rst-tu-dortmund/
    teb_local_planner_tutorials/rosbook/maps/maze.png
$ wget https://cdn.rawgit.com/rst-tu-dortmund/
    teb_local_planner_tutorials/rosbook/maps/maze.yaml
```

5.1 Differential-Drive Robot

Stage loads its environment from world files that define a static map and agents such like your robot (in plain text format). In the following, we add a world file to the *teb_tutorial* package which loads the map *maze.png* and spawns a differential-drive robot. The robot is assumed to be represented as a box ($0.25\text{ m} \times 0.25\text{ m} \times 0.4\text{ m}$). Whenever text files are edited, the editor *gedit* is utilized (`sudo apt-get install gedit`), but you might employ the editor of your preferred choice.

```
$ roscd teb_tutorial/stage
$ gedit maze_diff_drive.world # or use the editor of your
    choice
```

The second command creates and opens a new file with *gedit*. Add the following code and save the contents to file *maze_diff_drive.world*:

¹⁵Borrowed from the *turtlebot_stage* package: http://wiki.ros.org/turtlebot_stage.

```

## Simulation settings
1 resolution 0.02
2 interval_sim 100 # simulation timestep in milliseconds
3 ## Load a static map
4 model(
5   name "maze"
6   bitmap "../maps/maze.png"
7   size [ 10.0 10.0 2.0 ]
8   pose [ 5.0 5.0 0.0 0.0 ]
9   color "gray30"
10 )
11 ## Definition of a laser range finder
12 define mylaser ranger(
13   sensor(
14     range_max 6.5 # maximum range
15     fov 58.0 # field of view
16     samples 640 # number of samples
17   )
18   size [ 0.06 0.15 0.03 ]
19 )
20 )
21 ## Spawn robot
22 position(
23   name "robot"
24   size [ 0.25 0.25 0.40 ] # (x,y,z)
25   drive "diff" # kinematic model of a differential-drive robot
26   mylaser(pose [ -0.1 0.0 -0.11 0.0 ]) # spawn laser sensor
27   pose [ 2.0 2.0 0.0 0.0 ] # initial pose (x,y,z,beta[deg])
28 )

```

General simulation settings are defined in lines 1–3. Afterwards the static map is defined using a *stage model* object. The size property is important in order to define the transformation between pixels of *maze.png* and their actual sizes in the world. The bitmap is shifted by an offset defined in *pose* in order to adjust the bitmap position relative to the map frame. Simulated robots in this tutorial are equipped with a laser range finder set up in lines 12–20. Finally, the robot itself is setup in lines 22–28. The code specifies a differential drive robot (`drive "diff"`) with the previously defined laser scanner attached and the desired box size as well as the initial pose. The `base_link` frame is automatically located in the geometric center of the box, specified by the `size` parameter, which in this case coincides with the center of rotation. A case in which the origin must be corrected occurs for the car-like model (see Sect. 5.2).

In order to test the created robot model invoke the following commands in a terminal:

```

$ roscore
2 $ rosrun stage_ros stageros `rospack find teb_tutorial`/stage/
    maze_diff_drive.world

```

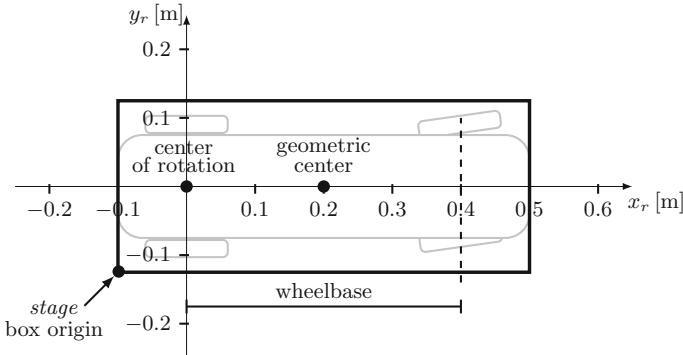


Fig. 7 Dimensions of the car-like robot for simulation

5.2 Car-Like Robot

In this section you generate a second world file that spawns a car-like robot. The 2D-contours of the car-like robot are depicted in Fig. 7 (gray-colored). For the purpose of this tutorial, only the boundary box of the robot is considered in the simple *stage* model. The length in y_r is increased slightly in order to account for the steerable front wheels. The top left and bottom right corners are located at $\mathbf{v}_{tl} = [-0.1, 0.125]^T$ m and $\mathbf{v}_{br} = [0.5, -0.125]^T$ m respectively w.r.t. the robot's base frame `base_link` (defined by the x_r - and y_r -axis). For car-like robots with front steering wheels the center of rotation coincides with the center of the rear axle. Since the TEB approach assumes a unicycle model for planning but with additional constraints for car-like robots such as minimum turning radius, **the robot's base frame must be placed at the center of rotation** in order to fulfill this relation.

The next step consist of duplicating the previous world file from Sect. 5.1 and modifying the robot model according to Fig. 7:

```
$ roscd teb_tutorial/stage
2 $ cp maze_diff_drive.world maze_carlike.world # duplicate
     diffdrive.world
$ gedit maze_carlike.world # or use the editor of your choice
```

Replace the robot model (line 21–28) by the following one and save the file:

```
## Spawn robot
2 position(
    name "robot"
4     size [ 0.6 0.25 0.40 ] # (x,y,z) - bounding box of the robot
      origin [ 0.2 0.0 0.0 0.0] # correct center of rotation (x,y,z
          ,beta)
6     drive "car" # kinematic model of a car-like robot
      wheelbase 0.4 # distance between rear and front axles
8     mylaser(pose [ -0.1 0.0 -0.11 0.0 ]) # spawn laser sensor
      pose [ 2.0 2.0 0.0 0.0 ] # initial pose (x,y,z,beta[deg])
10 )
```

Notice, the kinematic model is changed to the car-like one (`drive "car"`). Parameter `wheelbase` denotes the distance between the rear and front axle (see Fig. 7). The size of the robot's bounding box is set in line 4 w.r.t. the box origin as depicted in Fig. 7. `Stage` automatically defines the center of rotation in the geometric center, which is located at $[0.3, 0.125]^T$ m w.r.t. the box origin. In order to move the center of rotation towards the correct location $[0.1, 0.125]^T$ m w.r.t. the box origin, the frame is shifted as specified by parameter `origin`. Load and inspect your robot model in stage for testing purposes:

```
$ roscore
2 $ rosrun stage_ros stageros 'rospack find teb_tutorial' /stage/
  maze_carlike.world
```

The robot is controlled via a `geometry_msgs/Twist` message even though the actual kinematics refer to a car-like robot. But in contrast to the differential-drive robot, the angular velocity (yaw-speed, around z -axis) is interpreted as steering angle rather than the true velocity component.

6 Planning for a Differential-Drive Robot

This section covers the complete navigation setup with the `teb_local_planner` for the differential-drive robot defined in Sect. 5.1. Start by creating configuration files for the global and local costmap (refer to Fig. 3). In the following, configuration files are stored in a separate `cfg` folder inside your `teb_tutorial` package which was created during the steps in Sect. 5.

Create a `costmap_common_params.yaml` file which contains parameters for both the global and local costmap:

```
$ roscd teb_tutorial
2 $ mkdir cfg && cd cfg
$ gedit costmap_common_params.yaml
```

Now insert the following lines and save the file afterwards:

```
# file: costmap_common_params.yaml
2 # Make sure to preserve indentation if copied (for all yaml
  files)
footprint: [ [-0.125,0.125], [0.125,0.125], [0.125,-0.125],
  [-0.125,-0.125] ]
4
transform_tolerance: 0.5
6 map_type: costmap
  global_frame: /map
8 robot_base_frame: base_link
10 obstacle_layer:
  enabled: true
12  obstacle_range: 3.0
    raytrace_range: 4.0
```

```

14   track_unknown_space: true
15   combination_method: 1
16   observation_sources: laser_scan_sensor
17     laser_scan_sensor: {data_type: LaserScan, topic: scan,
18       marking: true, clearing: true}
19
20   inflation_layer:
21     enabled:           true
22     inflation_radius: 0.5
23
24   static_layer:
25     enabled:           true

```

The robot footprint is specified according to the projection of the dimensions of the robot ($0.25\text{ m} \times 0.25\text{ m} \times 0.4\text{ m}$) introduced in Sect. 6 onto the x - y -plane. The footprint must be defined in the `base_link` frame, which center coincides with the center of rotation. In this tutorial the selected `map_type` is `costmap` which creates an internal 2d grid. If the robot is equipped with 3D range sensors it is often desired to include the height of obstacles. This allows for ignoring obstacles beyond a specific height or tiny obstacles above the ground floor. For this purpose, the `costmap_2d`¹⁶ package also supports voxel grids. Refer to the `costmap_2d` wiki page for further information.

The obstacle layer is defined in lines 9–16 which includes external sensors such like our laser range finder by implementing ray-tracing. In this tutorial the laser range finder is expected to publish its range data on topic `scan`.

An inflation layer adds exponentially decreasing cost to cells w.r.t. their distance from actual (lethal) obstacles. This allows the user to set a preference for maintaining larger separation from obstacles whenever possible. Although the `tेब_local_planner` only extracts lethal obstacles from the costmap as described in Sect. 3.5 and ignores inflation, an activated inflation layer still influences the global planner and thus the location of virtual goals for local planning (refer to Sect. 3.2 for the description of virtual goals). Consequently, a non-zero `inflation_radius` moves virtual goals further away from (static) obstacles. Finally, the static layer includes obstacles from the static map which are retrieved from the `map` topic by default. The map `maze.png` is published later by the `map_server` node.

After saving and closing the file, specific configurations for the global and local costmap are created.

```

$ roscd teb_tutorial/cfg
2 $ gedit global_costmap_params.yaml # insert content, save and
$   close
$ gedit local_costmap_params.yaml # insert content, save and
$   close

```

¹⁶`costmap_2d`, URL: http://wiki.ros.org/costmap_2d.

The default content for `global_costmap_params.yaml` is listed below:

```
# file: global_costmap_params.yaml
2 global_costmap:
    update_frequency: 1.0
4   publish_frequency: 0.5
    static_map: true
6   plugins:
      - {name: static_layer,     type: "costmap_2d::StaticLayer"}
8      - {name: inflation_layer, type: "costmap_2d::InflationLayer"
      }
```

The global costmap is intended to be a static one which means its size is inherited from the map provided by the `map_server` node (notice the loaded static layer plugin which was defined in `costmap_common_params.yaml`). The previously defined inflation layer is added as well.

The content for `local_costmap_params.yaml` is as follows:

```
# file: local_costmap_params.yaml
2 local_costmap:
    update_frequency: 5.0
4   publish_frequency: 2.0
    static_map: false
6   rolling_window: true
    width: 5.5          # -> computation time: teb_local_planner
8   height: 5.5         # -> computation time: teb_local_planner
    resolution: 0.1    # -> computation time: teb_local_planner
10  plugins:
      - {name: obstacle_layer,  type: "costmap_2d::ObstacleLayer
      "}
```

It is highly recommended to define the local costmap as a rolling window in medium or large environments, since otherwise the implied huge number of obstacles might lead to intractable computational loads. The rolling window is specified by its width, height and resolution. These parameters have a significant impact on the computation time of the planner. The size should not exceed the local sensor range and it is often sufficient to set the width and height to values of approx. 5–6 m. The resolution determines the discretization granularity respectively how many grid cells are allocated in order to represent the rolling window. Since each occupied cell is treated as a single obstacle by default (see Sect. 3.5), a small value (resp. high resolution) indicates a huge number of obstacles and therefore long computation times. On the other hand, the resolution must be fine enough to cope with small obstacles, narrow hallways and passing doors. Finally, the previously defined obstacle layer is activated in order to incorporate dynamic obstacles obtained from the laser range finder.

Prior to generating the overall launch file, a configuration file for the local planner is created:

```
$ rosdep teb_tutorial/cfg
2 $ gedit teb_local_planner_params.yaml
```

The content of the `teb_local_planner_params.yaml` is listed below:

```

# file: teb_local_planner_params.yaml
2 TebLocalPlannerROS:

4 # Trajectory
5 dt_ref: 0.3
6 dt_hysteresis: 0.1
7 global_plan_overwrite_orientation: True
8 allow_init_with_backwards_motion: False
9 max_global_plan_lookahead_dist: 3.0
10 feasibility_check_no_poses: 3

12 # Robot
13 max_vel_x: 0.4
14 max_vel_x_backwards: 0.2
15 max_vel_theta: 0.3
16 acc_lim_x: 0.5
17 acc_lim_theta: 0.5
18 min_turning_radius: 0.0 # diff-drive robot (can turn in place
19 !
20     !
21 footprint_model:
22     type: "point" # include robot radius in min_obstacle_dist

22 # Goal Tolerance
23 xy_goal_tolerance: 0.2
24 yaw_goal_tolerance: 0.1

26 # Obstacles
27 min_obstacle_dist: 0.25
28 costmap_obstacles_behind_robot_dist: 1.0
29 obstacle_poses_affected: 10

30 # Optimization
31 no_inner_iterations: 5
32 no_outer_iterations: 4

```

For the sake of readability, only a small subset of available parameters is defined here. Feel free to add other parameters, e.g. after determining suitable parameter sets with `rqt_reconfigure`. In this example configuration, the point footprint model is chosen for optimization (parameter `footprint_model`). The circumscribed radius R of the robot defined in Sect. 5.1 is derived by applying geometry calculus: $R = 0.5\sqrt{0.25^2 + 0.25^2}$ m \approx 0.18 m. In order to compensate possible small distance violations due to penalty terms the parameter `min_obstacle_dist` is set to 0.25 m.

Parameter `costmap_obstacles_behind_robot_dist` specifies how many meters of the local costmap portion beyond the robot are taken into account in order to extract obstacles from cells.

After all related configuration files are created, the next step consist of defining the launch file that starts all nodes required for navigation and includes configuration files. Launch files are usually added to a subfolder called `launch`:

```
$ cd ~/catkin_ws/src/teb_tutorial/
2 $ mkdir launch && cd launch
$ gedit robot_in_stage.launch # create a new launch file
```

Add the following content to your newly created launch file:

```
<!!-- file: robot_in_stage.launch -->
2 <launch>
<!-- ***** Global Parameters ***** -->
4 <param name="/use_sim_time" value="true"/>

6 <!-- ***** Stage Simulator ***** -->
<node pkg="stage_ros" type="stageros" name="stageros" args="$(
    find teb_tutorial)/stage/maze_diff_drive.world">
8     <remap from="base_scan" to="scan"/>
</node>

10 <!-- ***** Navigation ***** -->
12 <node pkg="move_base" type="move_base" respawn="false" name="
    move_base" output="screen">
    <rosparam file="$(find teb_tutorial)/cfg/
    costmap_common_params.yaml" command="load" ns="
    global_costmap"/>
14     <rosparam file="$(find teb_tutorial)/cfg/
    costmap_common_params.yaml" command="load" ns="
    local_costmap"/>
    <rosparam file="$(find teb_tutorial)/cfg/
    local_costmap_params.yaml" command="load" />
16     <rosparam file="$(find teb_tutorial)/cfg/
    global_costmap_params.yaml" command="load" />
    <rosparam file="$(find teb_tutorial)/cfg/
    teb_local_planner_params.yaml" command="load" />

18     <param name="base_global_planner" value="global_planner/
    GlobalPlanner"/>
20     <param name="planner_frequency"   value="1.0" />
     <param name="planner_patience"    value="5.0" />

22     <param name="base_local_planner" value="teb_local_planner/
    TебLocalPlannerROS"/>
24     <param name="controller_frequency"   value="5.0" />
     <param name="controller_patience"    value="15.0" />
26 </node>

28 <!-- ***** Maps ***** -->
30 <node name="map_server" pkg="map_server" type="map_server" args
    ="$(find teb_tutorial)/maps/maze.yaml" output="screen">
    <param name="frame_id" value="/map"/>
32 </node>

34 <!-- ***** AMCL ***** -->
<node pkg="amcl" type="amcl" name="amcl" output="screen">
36     <param name="initial_pose_x"           value="2"/>
```

```

38      <param name="initial_pose_y"           value="2" />
39      <param name="initial_pose_a"           value="0" />
40      <param name="odom_model_type"          value="diff" />
41      <param name="use_map_topic"            value="true" />
42      <param name="transform_tolerance"       value="0.5" />
43    </node>
44  </launch>

```

After activating simulation time, the *stage_ros* node is loaded. The path to the world file previously created in Sect. 5.1 is forwarded as an additional argument. The command `$(find teb_tutorial)` automatically searches for the *teb_tutorial* package path in your workspace. Since *stage_ros* publishes the simulated laser range data on topic `base_scan`, but the costmap is configured for listening on `scan`, remapping is performed here.

Afterwards, the core navigation node *move_base* is loaded. All costmap and planner parameters are included relative to the *move_base* namespace. Some additional parameters are defined, such as which local and global planner plugins to be loaded. The selected global planner is commonly used in ROS. Parameters `planner_frequency` and `planner_patience` define the rate (in Hz) at which the global planner is invoked and how long the planner waits (seconds) without receiving any valid control before backup operations are performed, respectively. Similar settings are applied to the local planner with parameters `controller_frequency` and `controller_patience`. We also specify the `teb_local_planner/TebLocalPlannerROS` as the local planner plugin.

The two final nodes are the *map_server* node which provides the maze map and the *amcl* node for adaptive monte carlo localization. The latter corrects odometry errors of the robot by providing and adjusting the transformation between map and `odom`. *amcl* requires an initial pose which is set to the actual robot pose as defined in the stage world file. All other parameters are kept at their default settings in this tutorial.

Congratulations, the initial navigation setup is completed now. Further parameter adjustments are easily integrated into the configuration and launch files. Start your launch file in order to test the overall scheme:

```
$ rosrun teb_tutorial robot_in_stage.launch
```

Ideally, a stage window occurs, no error messages appear and *move_base* prints “*odom received!*”. Afterwards start a new terminal and open *rviz* in order to start the visualization and send navigation goals:

```
$ rosrun rviz rviz
```

Make sure to set the fixed frame to `map`. Add relevant displays by clicking on the *Add* button. You can easily show all available displays by selecting the tab *by topic*. Select the following displays:

- from /map/: Map
- from /move_base/TebLocalPlannerROS/: global_plan/Path, local_plan/Path, teb_markers/Marker and teb_poses/PoseArray
- from global_costmap/: costmap/Map
- from local_costmap/: costmap/Map and footprint/Polygon (we do not have a robot model to display, so use the footprint).

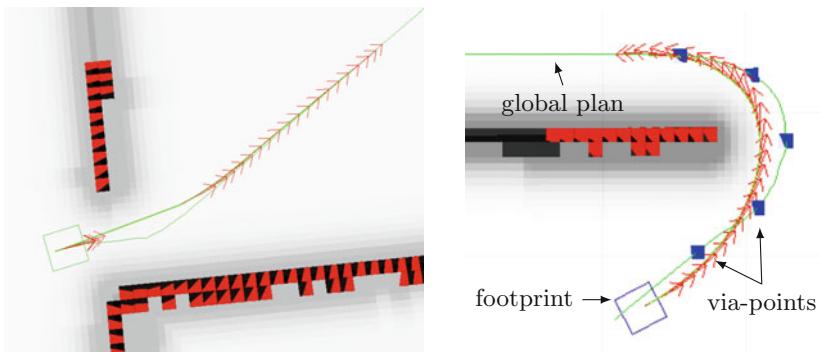
Now specify a desired goal pose using the *2D Nav Goal* button in order to start navigation.

In the following, some parameters are modified during runtime: Keep everything running and open in a new terminal:

```
$ rosrun rqt_reconfigure rqt_reconfigure
```

Select `move_base/TebLocalPlanner` in the menu in order to list all parameters which can be changed online. Now increase `min_obstacle_dist` to 0.5 m which is larger than the door widths in the map (the robot assumes a free space of 2×0.5 m then). Move the robot through a door and observe what is happening. The behavior should be similar to the one shown in Fig. 8a. The planner still tries to plan through the door according to the global plan and since the solution constitutes a local minimum. From the optimization point of view, the distance to each pose is minimized such that poses must be moved along the trajectory in both directions in order to avoid penalties (introducing the gap). However, `min_obstacle_dist` is chosen such that a door passing cannot be intended. As a consequence, the robot collides. After testing, reset the parameter back to 0.25 m.

The following task involves configuration of the trade-off between time optimality and global path following. Activate the via-points objective function by increasing the parameter `global_plan_via_point_sep` to 0.5. Command a



(a) Improper parameter value for `min_obstacle_dist`

(b) Via-points added along the global plan

Fig. 8 Testing navigation with the differential-drive robot

new navigation goal and observe the new blue quadratic markers along the global plan (see Fig. 8b). Via-points are generated each 0.5 m (according to parameter value `global_plan_via_point_sep`). Each via-point constitutes an attractor for the trajectory during optimization. Obviously, the trajectory still keeps a certain distance to some via-points as shown in Fig. 8b. The optimizer minimizes the weighted sum of both objectives: time optimality and reaching via-points. By increasing the weight `global_plan_via_point_sep` (via `rqt_reconfigure`) and commanding new navigation goals you might recognize that the robot increasingly tends to prefer the original global plan over the fastest trajectory. Note, an excessive optimization weight for via-points might cause the obstacle cost to become negligible in comparison to the via-point cost. In that case avoiding dynamic obstacles does not work properly anymore. However, a suitable parameter setting for a particular application is determined in simulation.

7 Planning for a Car-Like Robot

This section describes how to configure the car-like robot defined in Sect. 5.2 for simulation with `stage`. The steps for setting up the differential drive robot as described in the previous section must be completed before in order to avoid redundant explanations.

Create a copy of the `tेब_local_planner_params.yaml` file for the new car-like robot:

```
$ roscd tेब_tutorial/cfg
2 $ cp tेब_local_planner_params.yaml
     tेब_local_planner_params_carlike.yaml
$ gedit tेब_local_planner_params_carlike.yaml
```

Change the robot section according to the following snippet:

```
# file: tेब_local_planner_params_carlike.yaml
2 # Robot
max_vel_x: 0.4
4 max_vel_x_backwards: 0.2
max_vel_theta: 0.3
6 acc_lim_x: 0.5
acc_lim_theta: 0.5
8 min_turning_radius: 0.5          # we have a car-like robot!
wheelbase: 0.4                   # wheelbase of our robot
10 cmd_angle_instead_of_rotvel: True # angle instead of the rotvel
      for stage
weight_kinematics_turning_radius: 1 # increase, if the penalty
      for min_turning_radius is not sufficient
12 footprint_model:
      type:"line"
14 line_start: [0.0, 0.0] # include robot expanse in
      min_obstacle_dist
```

```
line_end: [0.4, 0.0] # include robot expanse in
min_obstacle_dist
```

Parameter `min_turning_radius` is non-zero in comparison to the differential-drive robot configuration. The steering angle ϕ of the front wheels of the robot is limited to $\pm 40 \text{ deg} (\approx \pm 0.7 \text{ rad})$. From trigonometry the relation between the turning radius r and the steering angle ϕ is defined by $r = L / \tan \phi$ [9]. Hereby, L denotes the wheelbase. Evaluating the expression with $\phi = 0.7 \text{ rad}$ and $L = 0.4 \text{ m}$ reveals a minimum turning radius of 0.47 m . Due to the penalty terms, it is rounded up to 0.5 m for the parameter `min_turning_radius`. Since `move_base` provides a `geometry_msgs/Twist` message containing linear and angular velocity commands v and ω respectively, the signals are transformed to a robot base driver that only accepts the linear velocity v and a steering angle ϕ . Since the turning radius is expressed by $r = v/\omega$, the relation to the steering angle ϕ follows immediately: $\phi = \text{atan}(Lv/\omega)$. The case $v = 0$ is treated separately, e.g. by keeping the previous angle or by setting the steering wheels to their default position. For robots accepting an `ackermann_msgs/AckermannDriveStamped` message type, a simple converter node/script is added to communicate and map between `move_base` and the base driver. As described in Sect. 5.2 *stage* requires the default `geometry_msgs/Twist` type but with changed semantics: the angular velocity component is interpreted as steering angle. The `tet_local_planner` already provides the automatic conversion for this type of interface by activating parameter `cmd_angle_instead_rotvel`. The steering angle ϕ is set to zero in case of zero linear velocities ($v = 0 \frac{\text{m}}{\text{s}}$).

The footprint model is redefined for the rectangular robot (according to Sect. 5.2). The line model is recommended for rectangular-shaped robots. Instead of defining the line over the complete width ($-0.1 \text{ m} \leq x_r \leq 0.5 \text{ m}$), 0.1 m are subtracted in order to account for the robot's expansion along the y_r -axis, since this value is added to parameter `min_obstacle_dist` similar to the differential-drive robot in Sect. 6. With some additional margin, `min_obstacle_dist` = 0.25 m should perform well, such that the parameter remains unchanged w.r.t. the previous configuration.

Create a new launch file in order to test the modified configuration:

```
$ roscd teb_tutorial/launch
2 $ cp robot_in_stage.launch carlike_robot_in_stage.launch
$ gedit carlike_robot_in_stage.launch
```

The `stage_ros` node must now load the `maze_carlike.world` file. Additionally, the local planner parameter configuration file must be replaced by the car-like version. An additional parameter `clearing_rotation_allows` is set to `false` in order to deactivate recovery behaviors which require the robot to turn in place. Relevant snippets are listed below:

```

<!-- file: carlike_robot_in_stage.launch -->
2 <!-- ... -->
<!-- ***** Stage Simulator ***** -->
4 <node pkg="stage_ros" type="stageros" name="stageros" args="$(
    find teb_tutorial)/stage/maze_carlike.world">
    <remap from="base_scan" to="scan"/>
6 </node>

8 <!-- ***** Navigation ***** -->
<node pkg="move_base" type="move_base" respawn="false" name="
    move_base" output="screen">
10   <!-- ... -->
    <rosparam file="$(find teb_tutorial)/cfg/
        teb_local_planner_params_carlike.yaml" command="load" />
12   <!-- ... -->
    <param name="clearing_rotation_allowed" value="false" /> <!--
        -- Our carlike robot is not able to rotate in place -->
14 </node>
<!-- ... -->

```

Close any previous ROS nodes and terminals and start the car-like robot simulation:

```
$ roslaunch teb_tutorial robot_in_stage.launch
```

If no errors occur, navigate your robot through the environment. Again run *rviz* for visualization with all displays configured in Sect. 5.1 and *rqt_reconfigure* for playing with different parameter settings. An example scenario is depicted in Fig. 9. The displayed markers in *rviz* indicate occupied cells of the local costmap which are taken into account as point obstacle during trajectory optimization.

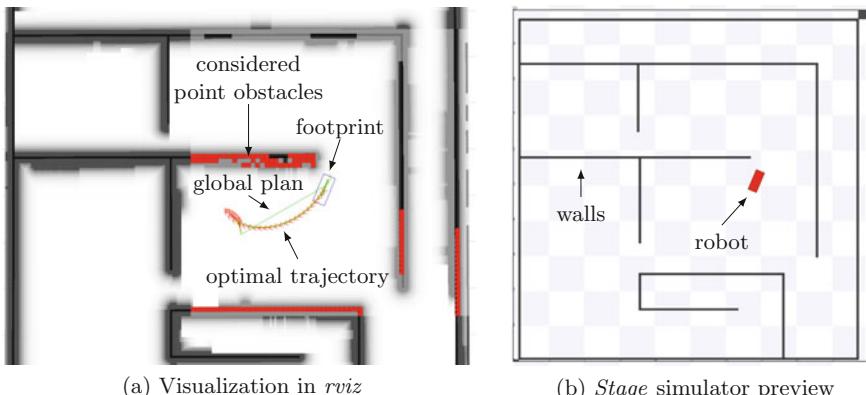


Fig. 9 Navigating a car-like robot in simulation

8 Conclusion

This tutorial chapter presented a step-by-step guide on how to setup the package *teb_local_planner* in ROS for navigation with a differential-drive and a car-like robot. The package implements an online trajectory optimization scheme termed Timed-Elastic-Band approach and it seamlessly integrates with the navigation stack as local planner plugin. The fundamental theory and concepts of the underlying approach along with related ROS parameters was introduced. The package provides an effective alternative to the currently available local planners as it supports trajectories planning with cusps (backward motion) and car-like robots. To our knowledge, the latter is currently not provided by any other local planner. The package allows the user to quantify a spatial-temporal trade-off between a time optimal trajectory and compliance with the original global plan. Further work intends to address the automatic tuning of cost function weights for common cluttered environments and maneuvers. Furthermore, a benchmark suite for the performance evaluation of the different planners available in ROS could be of large interests for the community. Benchmark results facilitate the appropriate selection of planners for different kinds of applications. Additionally, future work aims to include dynamic obstacles, support of additional kinematic models as well as further improving algorithmic efficiency.

References

1. Rösmann, C., Feiten, W., Wösch, T., Hoffmann, F., and T. Bertram. 2012. Trajectory modification considering dynamic constraints of autonomous robots. In *7th German Conference on Robotics (ROBOTIK)*, 74–79.
2. Rösmann, C., Feiten, W., Wösch, T., Hoffmann, F., and T. Bertram. 2013. Efficient trajectory optimization using a sparse model. In *6th European Conference on Mobile Robots (ECMR)*, 138–143.
3. Rösmann, C., Hoffmann, F., and T. Bertram. 2015. Planning of multiple robot trajectories in distinctive topologies. In *IEEE European Conference on Mobile Robots*, 1–6.
4. Kümmerle, R., Grisetti, G., Strasdat, H., Konolige, K., and W. Burgard. 2011. G2o: A general framework for graph optimization. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 3607–3613.
5. Nocedal, J., and S.J. Wright. 1999. *Numerical Optimization*., Springer series in operations research New York: Springer.
6. Morari, M., and J.H. Lee. 1999. Model predictive control: past, present and future. *Computers and Chemical Engineering* 23 (4–5): 667–682.
7. Bhattacharya, S., Kumar, V., and M. Likhachev. 2010. Search-based path planning with homotopy class constraints. In *Proceedings of National Conference on Artificial Intelligence*.
8. Guimaraes, R.L., de Oliveira, A.S., Fabro, J.A., Becker, T., and V.A. Brenner. 2016. ROS Navigation: Concepts and Tutorial. In *Robot Operating System (ROS) - The Complete Reference* (A. Koubaa, ed.), vol. 625 of *Studies in Computational Intelligence*, pp. 121–160, Springer International Publishing.
9. LaValle, S.M. 2006. *Planning Algorithms*. New York, USA: Cambridge University Press.

Christoph Rösmann was born in Münster, Germany, on December 8, 1988. He received the B.Sc. and M.Sc. degree in electrical engineering and information technology from the Technische Universität Dortmund, Germany, in 2011 and 2013 respectively. He is currently working towards the Dr.-Ing. degree at the Institute of Control Theory and Systems Engineering, Technische Universität Dortmund, Germany. His research interests include nonlinear model predictive control, mobile robot navigation and fast optimization techniques.

Frank Hoffmann received the Diploma and Dr. rer. nat. degrees in physics from Christian-Albrechts University of Kiel, Germany. He was a Postdoctoral Researcher at the University of California, Berkeley from 1996–1999. From 2000 to 2003, he was a lecturer in computer science at the Royal Institute of Technology, Stockholm, Sweden. He is currently a Professor at TU Dortmund and affiliated with the Institute of Control Theory and Systems Engineering. His research interests are in the areas of robotics, computer vision, computational intelligence, and control system design.

Torsten Bertram received the Dipl.-Ing. and Dr.-Ing. degrees in mechanical engineering from the Gerhard Mercator Universität Duisburg, Duisburg, Germany, in 1990 and 1995, respectively. In 1990, he joined the Gerhard Mercator Universität Duisburg, Duisburg, Germany, in the Department of Mechanical Engineering, as a Research Associate. During 1995–1998, he was a Subject Specialist with the Corporate Research Division, Bosch Group, Stuttgart, Germany. In 1998, he returned to Gerhard Mercator Universität Duisburg as an Assistant Professor. In 2002, he became a Professor with the Department of Mechanical Engineering, Technische Universität Ilmenau, Ilmenau, Germany, and, since 2005, he has been a member of the Department of Electrical Engineering and Information Technology, Technische Universität Dortmund, Dortmund, Germany, as a Professor of systems and control engineering and he is head of the Institute of Control Theory and Systems Engineering. His research fields are control theory and computational intelligence and their application to mechatronics, service robotics, and automotive systems.

Part III

**Integration of ROS with Internet and
Distributed Systems**

ROSLink: Bridging ROS with the Internet-of-Things for Cloud Robotics

Anis Koubaa, Maram Alajlan and Basit Qureshi

Abstract The integration of robots with the Internet is nowadays an emerging trend, as new form of the Internet-of-Things (IoT). This integration is crucially important to promote new types of cloud robotics applications where robots are virtualized, controlled and monitored through the Internet. This paper proposes ROSLink, a new protocol to integrate Robot Operating System (ROS) enabled-robots with the IoT. The motivation behind ROSLink is the lack of ROS functionality in monitoring and controlling robots through the Internet. Although, ROS allows control of a robot from a workstation using the same ROS master, however this solution is not scalable and rather limited to a local area network. Solutions proposed in recent works rely on centralized ROS Master or robot-side Web servers sharing similar limitations. Inspired from the MAVLink protocol, the proposed ROSLink protocol defines a lightweight asynchronous communication protocol between the robots and the end-users through the cloud. ROSLink leverages the use of a proxy cloud server that links ROS-enabled robots with users and allows the interconnection between them. ROSLink performance was tested on the cloud and was shown to be efficient and reliable.

A. Koubaa (✉) · M. Alajlan

Center of Excellence Robotics and Internet of Things (RIOT) Research Unit,
Prince Sultan University, Riyadh, Saudi Arabia
e-mail: akoubaa@coins-lab.org

M. Alajlan

e-mail: maram.ajlan@coins-lab.org

A. Koubaa

Gaitech Robotics, Hong Kong, China

A. Koubaa

CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

M. Alajlan

King Saud University, Riyadh, Saudi Arabia

B. Qureshi

Prince Sultan University, Riyadh, Saudi Arabia

e-mail: qureshi@psu.edu.sa

Keywords Robot Operating System (ROS) · Cloud robotics · Internet-of-Things · ROSLink · Mobile robots · Protocol Design

1 Introduction

Cloud Robotics [1–3] is a recent and emerging trend in robotics that aims at leveraging the use of Internet-of-Things (IoT) and cloud computing technologies to promote robotics applications from two perspectives: (*i*) Virtualization: providing seamless access to robots through Web and Web services technologies, (*ii*) Remote Brain: offloading intensive computations from robots to the cloud resources to overcome the computation, storage and energy limitations of robots.

Nowadays, Robot Operating System (ROS) [4] represents a defacto standard for the development of robotics applications. ROS as a middleware, provides several levels of software abstraction to hardware and robotics resources (i.e. sensor and actuators) in addition to the reuse of open source project libraries. It has been designed to overcome difficulties when developing large-scale service robots reducing the complexity of robotics software construction. Although widely used in developing applications for service robots, ROS lacks the native support for control and monitoring of robots through the Internet. It is possible to write ROS nodes (i.e. programs) in a remote workstation on the same local area network (LAN), where both the robot machine and the workstation use the ROS Master Uniform Resource Identifier (URI),, however controlling the ROS nodes from a remote location is challenging. To address this limitation many research works have been proposed focusing on client-server based architecture [5–10].

A milestone work that addressed this issues is the ROSBridge protocol [11]. It is based on Websockets server installed on the robot side that allows to send the internal status of the robot based on ROS topics and services and receives commands to Websockets clients to process them. This approach enabled the effective integration of ROS with the Internet; however, the fact that the Websockets server is running on the robot machine requires the robot to have a public IP address to be accessible by Websockets clients, which is not possible for every robot, or being on the same local area networks. Network address translation (NAT) could also be used when the robot is behind a NAT domain, but still this option may be cumbersome in deployment. In [12], the author proposed a ROS as a Web service which allow to define a Web service server in the robot to access through the Internet. However, this solution share the same limitation as ROSBridge as the server is located at the robot side.

This paper fills the gap and proposes ROSLink, a communication protocol that overcomes the aforementioned limitations by (*i*) implementing specifications of client in the robot side, (*ii*) manifestation of a proxy server located at a public IP server machine, such a cloud server. The idea is inspired by the MAVLink protocol [13], where the robot sends its data in serialized messages through a network client to a ground station that acts as a server which in turn, receives these messages, processes them and sends control commands to the robot. As such, it is no longer needed for a

robot to have a public IP address, whereas it will still be accessible behind the proxy server.

The contribution of this paper are two folded. First, we propose ROSLink a new communication protocol that defines a 3-tier architecture. The ROSLink Bridge client executes in the robot side; the ROSLink Proxy acts as a server in the ground station, and a client application at the user side that interacts with the robot through the ROSLink protocol. Second, we validate the proposed ROSLink protocol through an experimental study on the ground Turtlebot robot as well as the aerial AR.Parrot drone. We demonstrate the effectiveness and feasibility of ROSLink.

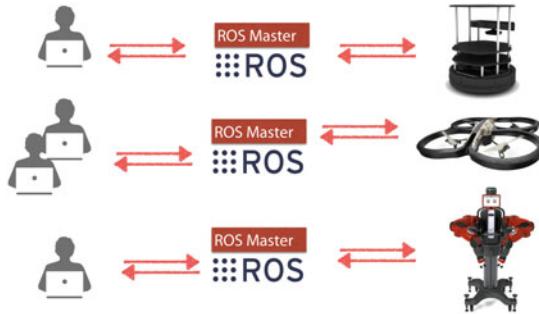
The remainder of this paper is organized as follows. Section 2 presents motivating examples and objectives behind the design of ROSLink. Section 3 presents the ROSLink communication protocol. Section 4 presents the experimental validation of ROSLink and the evaluation of its performance. Finally, Sect. 5 concludes the paper.

2 Motivating Problems and Objectives

2.1 Problem Statement

The motivation behind this work is to integrate ROS with the Internet of Things. ROS does not natively support monitoring and control of robots through the Internet. In fact, as illustrated in Fig. 1a, ROS allows to control a robot from a workstation using the same ROS master, but this solution is not scalable and rather limited to a local area network usage. The typical scenario is that every robot starts its own ROS Master node, and users can control the robot from their workstations if they configure their ROS network settings to use the same ROS Master running in the robot. This standard approach does not natively allow to control the robot through the Internet as robots typically do not have a public IP address. The use of port forwarding behind NAT can be considered in certain cases, but might not be possible in other cases, like connection through 3G/4G connection.

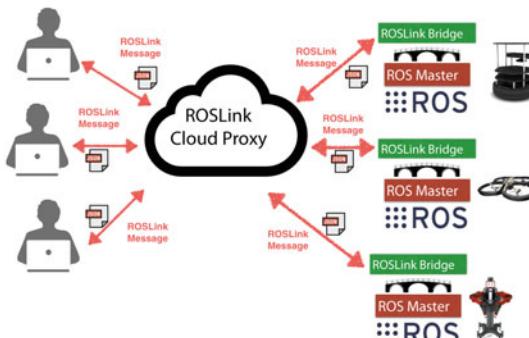
One possible solution, as illustrated in Fig. 1b, is to use one ROS Master node for all robots, where the Master node runs on a central server with a public IP address on the Internet. All users will connect to the same ROS Master and can access any robot by publishing and subscribing to their topics and services. However, this solution has several limitations. First, some ROS topics, services and nodes may be in conflict with having the same name. This issue requires a careful design of namespace for ROS nodes, services and topics to avoid any conflict. With the large number of robots, this solution becomes very complex. The second issue is the lack of scalability, i.e. the ROS Master might become overloaded when several robots are bound to it at a given time. Apart from the known networking issues while considering that several ROS topics are bandwidth greedy, there is no viable solution to mapping individual users to their robots since all topics will be visible to all users.



(a) **Standard Approach:** Typical connection between ROS robots and ROS users. A user is connected to the ROS Master of the robot to control and monitor its status, typically, in a local area network.



(b) **Centralized Approach:** One central ROS Master to which all robots and users connected. This solution is not scalable, does not provide effective management of robots and users.



(c) **ROSLink Approach:** A cloud based approach, where ROS robots and users interact through the cloud. The ROSLink cloud provides users and robots management, service oriented interfaces, and real-time streaming services.

Fig. 1 ROS operation approaches

Our approach consists in the design of ROSLink, a lightweight communication protocol, inspired from MAVLink [13], to allow for a cloud-based interaction between ROS robots and their users, as depicted in Fig. 1c. The idea is to add a ROSLink Bridge on top of ROS for every robot such that this bridge sends all the status of the robot using JSON serialized messages. ROSLink Bridge is a ROS node that accesses all topics and services of interest in ROS, and sends selected information in ROSLink messages, serialized in JSON format. These messages are sent to the ROSLink Cloud Proxy, which processes the messages and forwards them to the individual user and/or users of the robot. In addition, users send commands to the robot through the ROSLink cloud proxy utilizing ROSLink JSON messages, which are later processed by the ROSLink Bridge resulting in execution of the corresponding ROS action. The ROSLink cloud-based approach presents three major advantages (1) to be independent from the ROS master nodes of the robots, (2) ensure seamless communication between users and robots through the cloud, (3) provide effective management of robots, users and underlying services.

2.2 Overview

The main objective of ROSLink is to control and monitor a ROS-enabled robot through the Internet. In the literature, most of the related works focused on using two-tiered client/server approach while the server is implemented in the robot and the client is implemented in the user application. In fact, most of these researches are based on the instantiation of ROSBridge and ROSJS frameworks [11, 14] to build remotely controlled robots. ROSBridge represents a milestone that enabled this kind of remote control of ROS-enabled tele-operated robots. However, the drawbacks of this approach are (1) robot-centric approach, which restricts the scalability of the system as the server is centralized in the robot itself, (2) the deployment on Internet is rather difficult as the robot needs to have a public IP address or accessible through a NAT forwarding port, when it is inside a local area network.

To overcome these limitations, we propose ROSLink a three-tiered client/server model, where the client is implemented in the robot and the user, whereas the server is located at a public domain and acts as a proxy to link the robots with their users. ROSLink overcomes the two aforementioned problems. First, there is no longer a server implemented inside the robot, so it does no longer follow the robot-centric approach. In contrast, the robot implements the client side through the ROSLink Bridge components, which is a ROS node that interfaces with ROS on the one hand, and on the other hand sends ROS data to the outside world through a network interface (UDP, TCP, or Websockets). Besides, the ROSLink server side of the model is implemented in a publicly available server called ROSLink proxy, which acts as a mediator between the robots and the users. Robots and users send messages to the proxy server, which will dispatch them accordingly to the other side.

ROSLink makes a complete abstraction of ROS by defining a communication protocol (inspired from the MAVLink protocol) that provides all information about

the robot through ROS topics/services without exposing ROS ecosystem to the users. The user does not need to be familiar with ROS to be able to use ROSLink to send commands for robot. ROSLink defines a set of interfaces for the users to interact with the robot, and a set of messages to exchange between them.

ROSLink messages are constructed based on ROS topics/services parameters to either get data from or submit data to execute an action. The messages are represented as JSON strings. JSON format is opted to use in data interchange because it is platform-independent and language-independent data representation format [15]. In addition, it is a lighter-weight solution as compared to XML, which is more appropriate for resource-constrained and bandwidth-constrained platforms. This will allow the client application developer to choose any programming language (C++, Java, JavaScript, Python, etc.) to develop a client application that interacts with ROSLink to command and monitor ROS-enabled robots.

In summary, ROSLink differs from previous works, and in particular from ROSBridge in these aspects:

- ROSLink implements a client in the robot as well as in the user applications, and implements a server in an intermediate proxy, whereas ROSBridge implements a Websocket server in the robot and Websocket clients in user applications.
- ROSBridge is based on the Websocket protocol, whereas ROSLink can be implemented with any transport layer protocol (TCP, UDP and Websockets). In this paper, we used UDP and Websockets interfaces to implement ROSLink.
- ROSLink does not rely on ROSBridge as in previous works but defines its own communication protocol between ROS and non-ROS users.

3 The ROSLink Protocol

In what follows, we present an overview of system and software architecture.

3.1 *ROSLink System Architecture*

The general architecture of ROSLink is presented in Fig. 2.

The system is composed of three main parts:

- **The ROSLink Bridge:** this is the main component of the system. It is the interface between ROS and the ROSLink protocol. This bridge has two main functionalities: (1) it reads data from messages of ROS topics and services, serializes the data in JSON format according to ROSLink protocol specification, and sends to a ground station, a proxy server or a client application, (2) receives JSON serialized data through a network interface from a ground station or a client application, deserializes it from the JSON string, parse the command, and executes it through ROS.

- **The ROSLink Proxy and Cloud:** it acts as a proxy server between the ROSLink Bridge (embedded in the robot) and the user client application. Its role is to link a user client application to a ROS-enabled robot through its ROSLink Bridge. The ROSLink proxy is mainly a forwarder of ROSLink messages between the robot and user. It allows to keep the user updated with the robot status, and also forward control commands from the user to the robot. In addition ROSProxy interact with ROSLink Cloud component, to maintain and manages the list of robots and users, create a mapping between them, and perform all management functionality, including security, quality-of-service monitoring, etc.
- **The ROSLink Client Application:** it basically represents a control and monitoring application of the robot. This application is intended to monitor the status of the robot that it receives through ROSLink messages via the ROSLink Proxy from the robot. In addition, it sends commands through ROSLink messages to control the robot activities.

3.2 ROSLink Communication Protocol

We designed the ROSLink communication protocol to allow interaction between the different parts of the ROSLink system, namely the ROSLink Bridge, ROSLink Proxy and the ROSLink Client application.

The ROSLink communication protocol is based on two main things: (1) The transport protocol to use to communicate between the users, clouds and robots. (2) the message specification and its serialization in JSON format.

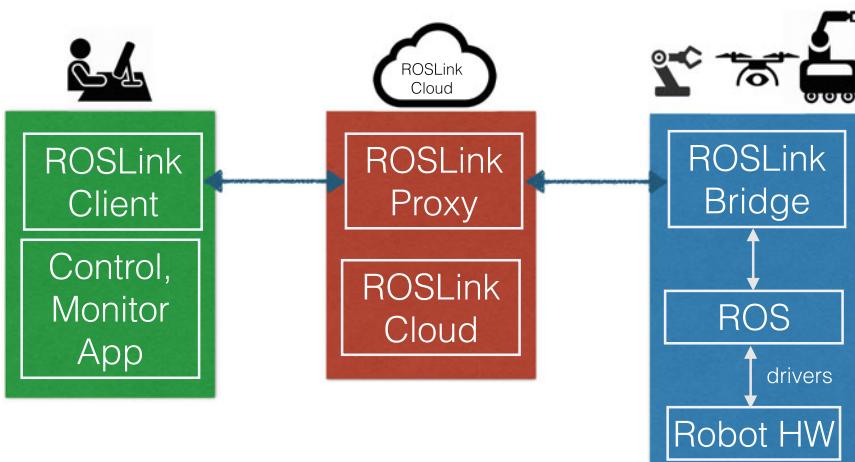


Fig. 2 ROSLink architecture

Transport Protocol ROSLink Bridge, ROSLink Proxy and ROSLink Client all of them use a network interface to communicate. There are different options for the transport protocol, which include UDP, TCP and Websockets. Communication through Serial port and telemetry devices is not considered as we only aim at a communication pattern through the Internet.

In our ROSLink implementation, we considered both UDP and Websockets transport protocols. The ROSLink Proxy implements both UDP and Websockets servers providing different interfaces for robots and users to interact with it. On the other hand, ROSLink Bridge and ROSLink Client can implement either protocol UDP, or Websockets clients or both together that interact with the ROSLink Proxy server. This gives enough flexibility to the developer to choose the most appropriate transport protocol for his application. On the one hand, we opted for the UDP connection because it is better and lighter weight choice for real-time and loss-tolerant streaming applications as compared to TCP, as it is the case with ROSLink data exchange model. In fact, the robot will be streaming its internal status (e.g. position, odometry, velocities, etc.) in real-time to the proxy server, which will deliver it to the ROSLink client application. On the other hand, the Websockets interface also provide an idea protocol for more reliable transport of data streams as compared to UDP, while meeting real-time requirements. In fact, Websockets is a connection-based protocol that opens the connections between the two communicating ends before data exchanges, and ensure connection to remain open all along the message exchange sessions. The connection is closed when any of the two ends terminate the sessions, which make it more reliable. It is also possible to think of using a TCP connection for better reliability of transfer, but in our context, the lost of data occasionally is not that critical. It might be critical in case of closed-control loop applications, which is out of our scope at this stage.

ROSLink Message Types The ROSLink communication protocol is based on the exchange of ROSLink messages. ROSLink messages are JSON formatted strings that contain information about the command and its parameters. To standardize the type of messages exchanged, we specified a set of ROSLink messages that are supported by the ROSLink Proxy. These message can be easily extended based on the requirements of the user and the application. There are two main categories of ROSLink messages: (i.) *State messages*: these are message sent by the robot and carry out information about the internal state of the robot, including its position, orientation, battery level, etc. (ii.) *Command messages*: these are messages sent by the client application to the robot and carry out commands to make the robot execute some actions, like for example moving, executing a mission, going to a goal location, etc. In what follows, we identify an example of messages and command types:

- **Presence message:** the robot should declare its presence regularly to declare itself and to be considered as active. Typically, Heartbeat messages sent at a certain frequency (typically one message per second) are used for this purpose.
- **Motion messages:** In robot mission, it is important to know the location and odometry motion parameters (i.e. linear and angular velocities) of the robot at a

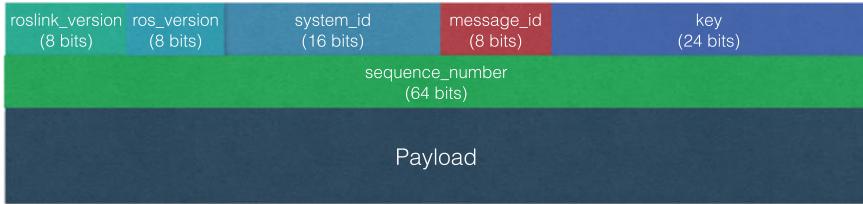


Fig. 3 ROSLink message header structure

certain time. Thus, a `motion` message containing position information of the robot should be periodically broadcast.

- **Sensor messages:** The robot needs to broadcast its internal sensor data such as IMU, laser scanners, camera images, GPS coordinates, actuators states, etc. ROSLink also defines several sensor messages to exchange these data between the robot and the user.
- **Motion commands:** For the robot to navigate in ROS, certain commands are sent to it like `Twist` messages in ROS, goal/waypoint locations, and takeoff/landing command for drones. ROSLink also specifies different types of commands to make the robot moves as desired.

The aforementioned list is not exhaustive as other types of messages can be designed based on the requirements of the users and available information from the robot. In what follows, we present the ROSLink message structures for the main state messages and commands.

ROSLink Message Structure A ROSLink message is composed of a header and a payload. The structure of the ROSLink message header is presented in Fig. 3.

ROSLink Message Header: The total header size is 128 bits. The `roslink_version` is encoded as a short int on 8 bits and specifies the version of ROSLink protocol. This is because in the future, new ROSLink versions would be released and it is important to specify which version a message belongs to for correct parsing. The `ros_version` specifies the ROS version (e.g. Indigo). The `system_id` is an int encoded into 16 bits and specifies the ID of the robot. It helps in differentiating robots from each other at the server side. It is possible to encode the `system_id` in 8 bits to reduce the header size, but the problem this restricts the scalability of the system to only 256 robots ID. The `message_id` specifies the type of message received. It helps in correctly parsing the incoming message and extract underlying information. The `sequence_number` denotes the sequence of the packet, identifies a single packet, and avoid processing duplicate packets. Finally, the `key` field is encoded on 24 bits and is used to identify a robot, and to map it to a user. A user that would like to have access to a robot, must use the same key that the robot is using in its Heartbeat message.

ROSLink Message Payload: The payload carries out data relevant for each ROSLink message type. ROSLink defines several state message and command types.

In what follows, we give an overview of the most common state and command messages. For a complete set of messages, the reader may refer to [16].

The most basic ROSLink message is the **Heartbeat** message, which is sent periodically from the robot to the ROSLink proxy server, and vice-versa. Every ROSLink Bridge should implement the periodic transmission of the Heartbeat message. The objective of the Heartbeat message is for the proxy server to ensure that the robot is active and connected, upon reception of that message. In the same way, a robot that receives a Heartbeat message from the ROSLink Proxy server ensures that the server is alive. This message increases the reliability of the system when it uses a UDP connectionless protocol, such that both ends make sure of the activity of the other end. Failsafe operations can be designed when the robot loses reception of Heartbeat messages from the user such as stopping motion or returning to start location until connectivity is resumed.

The Heartbeat message structure is defined in JSON representation in Listing 1.1. In the ROSlink protocol, the `message_id` of the Heatbeat message is set to zero.

Listing 1.1 Heartbeat Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": 0,
  "sequence_number": int64,
  "payload": {
    "type": int8
    "name": String,
    "system_status": int8,
    "owner_id": String,
    "mode": int8
  }
}
```

The **Robot Status** message contains the general system state, like which onboard controllers and sensors are present and enabled in addition to information related to the battery state. Listing 1.2 presents the Robot Status message structure, which has a `message_id` equal to 1.

Listing 1.2 Robot Status Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": 1,
  "sequence_number": int64,
  "payload": {
    "onboard_control_sensors_present": uint32,
    "onboard_control_sensors_enabled": uint32,
    "voltage_battery": uint16,
    "current_battery": int16,
    "battery_remaining": int8,
  }
}
```

The **Global motion** message represents the position of the robot and its linear and angular velocities. This information is sent to the ROSLink client at high frequency to keep track of robot motion state in real-time. An instance of the Global motion message structure is expressed in Listing 1.3:

Listing 1.3 Global Motion Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": int8,
  "sequence_number": int64,
  "payload": {
    "time_boot_ms": uint32
    "x": float64,
    "y": float64,
    "z": float64,
    "vx": float64,
    "vy": float64,
    "vz": float64,
    "wx": float64,
    "wy": float64,
    "wz": float64,
    "pitch": float64,
    "roll": float64,
    "yaw": float64,
  }
}
```

Listing 1.4 presents the **Range Finder Data** message, which carries out information and data about laser scanners attached to the robot. The Range Finder Data sensor information enables to develop control application on the client through the cloud, such as obstacle avoidance reactive navigation, SLAM, etc.

Listing 1.4 Range Finder Data Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": int8,
  "sequence_number": int64,
  "payload": {
    "time_usecs": int64
    "angle_min": float32,
    "angle_max": float32,
    "angle_increment": float32,
    "time_increment": float32,
    "scan_time": float32,
    "range_min": float32,
    "range_max": float32,
    "ranges": float32[],
    "intensities": float32[],
  }
}
```

The following Listings present a few examples on command messages that can be sent from the ROSLink client application to the robot through the cloud.

The most basic command message is the **Twist command** message, which controls the linear and angular velocities of the robot, and is illustrated in Listing 1.5. This ROSLink Twist message maps directly with the Twist message defined in ROS. Once the ROSLink Twist command reaches the ROSLink Bridge, it is first deserialized from the JSON wrapper, then extract velocity commands and publishes them as a ROS Twist message to make the robot move.

Listing 1.5 Twist Command Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": int8,
  "sequence_number": int64,
  "payload": {
    "lx": float,
    "ly": float,
    "lz": float,
    "ax": float,
    "ay": float,
    "az": float
  }
}
```

To stop the robot, it simply requires to send a `Twist` command message with all velocities set to zeros. In the same way, the **Go to Waypoint** command message defines a command to send the robot to a specific goal location. The parameters `x`, `y` and `z` represent the 3D coordinates of the goal location. The `frame_type` field represents the world frame if it is set to `true`, and the robot frame if it is set to `false`.

Listing 1.6 Go-To-Waypoint Command Message Structure

```
{
  "roslink_version": int8,
  "ros_version": int8,
  "system_id": int16,
  "message_id": int8,
  "sequence_number": int64,
  "payload": {
    "frame_type": boolean,
    "x": float,
    "y": float,
    "z": float
  }
}
```

Several other commands and state messages were also defined like the takeoff/land command messages in the context of drones, and the GPS state message that provides information of the GPS sensor, etc.

3.3 Integration of ROSLink Proxy in Dronemap Planner

As mentioned in Sect. 3.1, ROSLinkProxy is responsible for map users to robots, manages them, and control the access of users to robots. We integrated ROSLink Proxy into the Dronemap Planner cloud application [17]. Dronemap Planner is modular service-oriented cloud based system that was originally developed for the management of MAVLink drones and to provide seamless access to them through the Internet.

The software architecture of Dronemap Planner defines a proxy layer interface that allow to mediate between robots and users. In [17], this proxy interface was implemented as a MAXProxy component that mediates between MAVLink drones and users. As such Dronemap Planner is a comprehensive system that allow to control both MAVLink and ROSLink robots simultaneously. Video demonstrations of Dronemap Planner related to control of drones over the Internet are available in [18].

We extended Dronemap Planner to also support the ROSLink protocol by implementing the proxy layer interface as ROSLinkProxy that allows the exchanges of ROSLink messages between ROS-enabled robots and their users. Dronemap Planner provides a comprehensive system for drones and users management, and session handling. In addition, Dronemap Planner provides both Websockets and UDP sockets network interfaces for ROSLink robots and users.

We have used Dronemap Planner to run experiments related to the performance evaluation of ROSLink in controlling and monitoring ROS-enabled robots through the Internet, in Sect. 4.

4 Experimental Validation

In this section, we present an experimental study to demonstrate the effectiveness of ROSLink and to assess its impact on real-time open-loop control applications.

We investigate the impact of network and cloud delays on the performance of open-loop control applications of ROS-enabled robots. With open-loop control, the commands are sent to the robot without the need for any feedback from the robot. The problem can be formulated as follows: “*If the control application is offloaded from the robot to the ROSLink client, what is the impact of network and cloud processing delays on the performance of the control?*”

Trajectory Control Application To address this question, we consider a prototype open-loop control application of the motion of Turtlesim robot (default simulator in ROS) to follow a spiral trajectory. We have chosen a spiral trajectory motion control application because the resulting trajectory is sensitive to delays and jitters. A spiral trajectory is defined by a combination of an increasing linear velocity over time and a constant angular velocity. The general algorithm for drawing a spiral trajectory is presented in the following listing:

Listing 1.7 Spiral Trajectory Motion Control Algorithm

```

double constant_angular_speed=ANGULAR_SPEED_CONSTANT;
double linear_velocity_step = LINEAR_STEP_CONSTANT;
double time = SIMULATION_TIME_CONSTANT;
double rate = FREQUENCY_OF_UPDATE_CONSTANT;
int number_of_iterations = (int)(time*rate);
double linear_velocity = _INIT_LINEAR_VELOCITY;

for(int i=0;i<number_of_iterations;i++){
    linear_velocity=linear_velocity+linear_velocity_step;
    angular_speed =constant_speed;
    send_velocity_command(linear_velocity, angular_speed);
    sleep (1/rate);
}

```

We implemented this algorithm in both the ROSLink Control Application in the client side, as well as on-board of the robot using ROS. The objective is to qualitatively and quantitatively compare the resulting trajectories executed on-board using ROS against the ones produced by ROSLink Control application through the Dronemap Planner cloud.

Experimental Set-up We consider a spiral trajectory generation through the transmission of Twist messages where linear and angular velocities follow the algorithms in Listing 1.7.

We analyzed the impact of networks and cloud processing delays on the generated spiral trajectories with varying the linear velocity steps, and also the discrete time granularity captured by the update rate variable in Listing 1.7.

Using ROS, Twist velocity commands are directly published from a ROS script running on the same machine of the robot ROS Master. When using ROSLink, Twist velocity commands are sent in ROSLink Twist messages to the Dronemap Planner cloud, which forwards them to the ROSLink Bridge which is running on the robot. Finally the Twist ROSLink command is converted into a ROS Twist messages published on the velocity topic. An example of spiral trajectory generated by ROS and ROSLink are presented in Fig. 4.

Qualitative Analysis Figure 5 presents the generated spiral trajectories using open-loop motion control for both ROS and ROSLink. The spiral corresponds to a rate frequency equal to 2 Hz, an initial velocity of 1.0, a constant angular velocity equal to 4.0 and a linear step equal to 0.05 and 0.13. A first observation is that these spiral trajectories are visually correlated and follow a similar pattern based on the linear speed incremental steps. However, it can be observed that the spiral generated by ROSLink client control commands has more curves than the trajectory generated by Twist control command published directly from ROS. This becomes more contrasted when the velocity step increases. The main reasons behind this behavior are both delays and jitters. The delay variation observed with ROSLink are induced by the network and cloud processing results in updating the linear velocity at different instances than with ROS.

Quantitative/Statistical Analysis Figure 5 depicts the box plot of the jitters for four scenarios, namely, (i) using ROS onboard, (ii) using ROSLink when the cloud

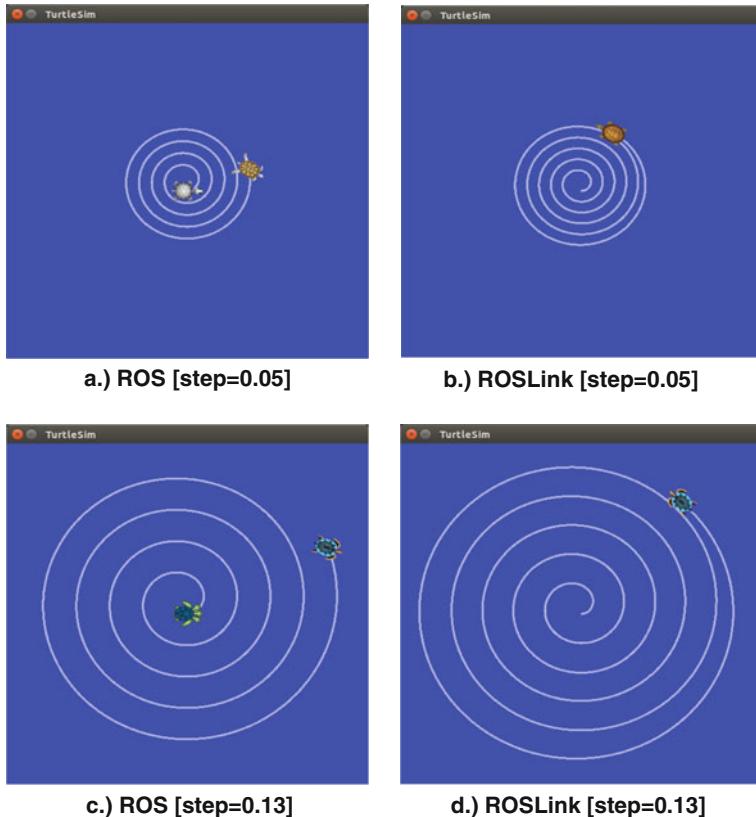


Fig. 4 ROS and ROSLink spiral generated trajectories

server is located in the same localhost as ROS (ROSLinkLH), (iii) two scenarios with ROSLink (ROSLinkCL1 and ROSLinkCL2) where the cloud server is located on the DreamCompute cloud instance machine, while being accessed from two different networks. Table 1 presents the average values, standard deviations and coefficient of variations for the jitter.

The ROSLinkLH scenario allows to investigate the impact of cloud processing without networking delays as ROS and ROSLink are in the same machine. On the other hand, ROSLinkCL1 and ROSLinkCL2 capture the impacts of both cloud processing and networking over the Internet. As anticipated, in the presence of the jitter, ROS is much stable compared with ROSLink scenarios. In fact, the average jitters with ROSLink scenarios are around 1000 times larger compared to ROS, as can be observed in Table 1. This provides an explanation for the discrepancies between the two spiral trajectories. However, the jitters remain in order of tens of microseconds, which is acceptable and provides no threat of compromising the real-time control of the robot through the cloud. Furthermore, we observe that the network has an

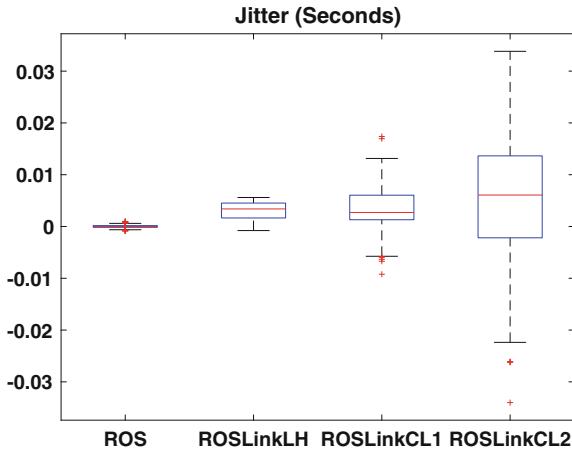


Fig. 5 ROS and ROSLink spiral generated trajectories

Table 1 Effect of Jitter on various ROSLink deployments

	Average	StdDev	Coeff Var
ROS	3.5274e-06	3.1892e-04	90.4110
ROSLinkLH	0.0031	0.0016	0.5206
ROSLinkCL1	0.0035	0.0044	1.2684
ROSLinkCL2	0.0044	0.0122	2.7605

additional impact to the cloud processing on the jitter, when comparing the jitter of ROSLink cloud server when deployed on Localhost, against ROSLink cloud server which is deployed over the Internet. The unpredictable network delays on the communication channel contribute naturally to a larger jitter. Results shown in Table 1 confirm this observation, where the standard deviation of coefficient of variation reach upwards of up to 5 to 6 six times increase in case of the Internet deployment of the ROSLink cloud.

It must be noted that although these jitters exist, they do not compromise the quality of control and similarity of trajectories. To verify the similarity and correlation between the generated trajectories, we applied chi-square test statistical method for assessing the goodness of fit between the time series of the trajectories at a significance level of 5%. It has been found that all trajectories are correlated based on the chi-square goodness of fit.

The results presented here illustrate the effectiveness of robot control through the cloud with reasonable real-time constraints. The quality of control could be improved using a feedback-control, which we will investigate further in future work. In addition, ROSLink could also be effectively used for request-response communication pattern between the client and the robot by for example using it to send goal points to be executed by the ROS navigation services of the robot.

5 Related Work

Over the past few years the robotics community has made quite a few forays into using robots over the Internet. Earlier contributions in remote access and control of the robots over the Internet were made by on-line robotics labs. The remote robotics labs allows users to address research and engineering problems in physical laboratories which are made available over the Internet. Examples of such labs include the UWA Telerobot [19], UJI on-line robot [20], Lego robots lab [5], PR2 Remote Lab [6] among many others. Previous endeavors focused on executing simple experiments remotely supporting on-line learning but did not address shared control of remote robots through middle-ware. The PR2 Lab [6] introduced a middle-ware that allows developing web based robots applications using the open-source Robot Web Tools (RWT) [14] interfacing robots running the ROS. Further works in developing and using middle-ware involving web accessibility of ROS based applications includes Robotic Programming Network (RPN) presented in [7], which allows users to execute programs through a client in a browser window using python. While RPN is a viable solution for remote learning, users of the system need to be familiar with ROS packages and programming interfaces which could be complex for new learners.

Esteller-Curto in [8] provide a REST-based architecture server to control a ROS enabled robot using the client server approach where the robot executes the client application connecting to the server. The communication between the client and the server is done using HTTP headers and in XML. Although an implementation is provided, the work lacks a high level architecture to store information from various clients so as to avoid collision at the server. The DAvinCi project presented in [9] introduced a cloud robotics framework that allows multiple client applications executing on ROS enabled robots to communicate to the DAvinCi server over the Internet. The server maintains the state of collaborative work completed by the clients. For large number of clients, the intermediate server can be exposed to network congestion thus needing further work in reliability and scalability of the service. This project provided a foray extending the robot-to-robot communications over the Internet to the cloud based robotics middle-ware. Narita et. al. in [10] present a reliable cloud based robot services platform extending their earlier work on Robot Service Network Protocol. The suggested protocol adopts push communication and a reliable messaging mechanism in order to perform reliable communication on robot services based on web service technologies. Authors in [14] present rosjs, a Javascript library for ROS, that allows developers to expose robot functionality as web services. The rosjs allows developers to create robot applications that can be used in the web browser and extends ROS to provide security and data logging mechanisms.

The ROSBridge protocol in [11] uses a Websockets server installed on the robot side that allows to send the internal status of the robot based on ROS topics and services and receives commands to Websockets clients to process them. This approach enabled the effective integration of ROS with the Internet; however, the fact that the Websockets server is running on the robot machine requires the robot to have a public IP address to be accessible by Websockets clients, which may not be possible for all

robots. In [12], the author proposed ROS-as-a-web-service which allows to define a Web service server in the robot to access through the Internet. However, this solution share the same limitation as ROSBridge as the server is located at the robot side.

6 Conclusion

In this chapter, we proposed, ROSLink, a novel communication protocol to integrate ROS-enabled robots with the Internet-of-Things for cloud robotics. We designed and developed ROSLink protocol that allows to access any ROS-enabled robot through the Internet. We have specified the message set of the protocol and used JSON serialization between the different ends of the system. ROSLink performance was tested on the cloud on Internet and was shown to be efficient and reliable for controlling robots through the cloud.

It can be noticed from the chapter outcomes that ROSLink is a lightweight and effective solution to integrate ROS-enabled robots with the Internet of Things. Having the server implemented into a publicly accessible cloud makes it possible to map any robot to any user in a seamless fashion through the Internet. There is no need for NAT translation as in solutions based on a robot-side server, like ROSBridge and others. Furthermore, the high-speed connections available today's Internet and sufficient bandwidth is an enabling factor for providing high quality of services for applications deployed with ROSLink. In fact, ROSLink has a lightweight overhead (including both header and payload size) and thus it does not consume much bandwidth over the network, which makes it scalable. The evaluation study discussed the impact of network on the performance for different networks' configurations.

We are currently working on extending the performance evaluation study for considering the closed loop control, namely with a PID controller for altitude and position control of a drone over ROSLink. We also aim at improving the security measures of ROSLink by using authentication and encryption of message to prevent possible attacks like impersonation and authorized access to robots.

Acknowledgements This work is supported by the Dronemap project entitled “DroneMap: A Cloud Robotics System for Unmanned Aerial Vehicles in Surveillance Applications” under the grant number 35-157 from King Abdul Aziz City for Science and Technology (KACST), and supported by Prince Sultan University. In addition, the authors would like to thank the Robotics and Internet of Things (RIoT) Unit at Prince Sultan University’s Innovation Center for their support to this work. The authors would like also to thank Gaitech Robotics in China for their support to this work.

References

1. Kuffner, J. 2010. Cloud-enabled robots. In *IEEE-RAS International Conference on Humanoid Robots*. IEEE.

2. Chaari, R., F. Ellouze, A. Koubaa, B. Qureshi, N. Pereira, H. Youssef, and E. Tovar. 2016. Cyber-physical systems clouds: A survey. *Computer Networks* 108: 260–278. <http://www.sciencedirect.com/science/article/pii/S1389128616302699>.
3. Qureshi, B., and A. Koubaa. 2014. Five traits of performance enhancement using cloud robotics: A survey. *Procedia Computer Science* 37: 220–227. <http://www.sciencedirect.com/science/article/pii/S1877050914009983>.
4. Quigley, M., K. Conley, B.P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. 2009. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.
5. Casini, M., A. Garulli, A. Giannitrapani, and A. Vicino. 2014. A remote lab for experiments with a team of mobile robots. *Sensors* 14: 16486–16507.
6. Pitzer, B., S. Osentoski, G. Jay, C. Crick, and O.C. Jenkins. 2012. Pr2 remote lab: An environment for remote development and experimentation. In *IEEE International Conference on Robotics and Automation (ICRA)*.
7. Casañ, G.A., E. Cervera, A.A. Moughlbay, J. Alemany, and P. Martinet. 2015. Ros-based online robot programming for remote education and training. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, 6101–6106.
8. Esteller-Curto, R., E. Cervera, A.P. del Pobil, and R. Marin. 2012. Proposal of a rest-based architecture server to control a robot. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, 708–710.
9. Arumugam, R., V.R. Enti, L. Bingbing, W. Xiaojun, K. Baskaran, F.F. Kong, A.S. Kumar, K.D. Meng, and G.W. Kit. Davinci: A cloud computing framework for service robots. In *2010 IEEE International Conference on Robotics and Automation*, 3084–3089.
10. Narita, M., S. Okabe, Y. Kato, Y. Murakwa, K. Okabayashi, and S. Kanda. 2013. Reliable cloud-based robot services. In *39th Annual Conference of the IEEE Industrial Electronics Society*.
11. Crick, C., G.T. Jay, S. Osentoski, B. Pitzer, and O.C. Jenkins. 2011. rosbridge: ROS for Non-ROS Users. In *International Symposium on Robotics Research (ISRR 2011)*. AZ, USA: Flagstaff.
12. Koubaa, A. 2015. ROS As a Service: Web Services for Robot Operating System. *Journal of Software Engineering for Robotics* 6 (1).
13. MAVLink Micro Air Vehicle Communication Protocol. 2016. <http://qgroundcontrol.org/mavlink/start>. Accessed 7 Jun 2016.
14. Osentoski, S., G. Jay, C. Crick, B. Pitzer, C. DuHadway, and O.C. Jenkins. 2011. Robots as web services: Reproducible experimentation and application development using rosjs. In *2011 IEEE International Conference on Robotics and Automation (ICRA)*, 6078–6083.
15. The JSON Data Interchange Format, First edition ed. ECMA International. 2013. <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
16. ROSLink Message Specification Set. 2016. <http://wiki.coins-lab.org/roslink/ROSLINKCommonMessageSet.pdf>. Accessed 22 Oct 2016.
17. Koubaa, A., B. Qureshi, M.-F. Sriti, Y. Javed, and E. Tovar. 2016. Dronemap planner: A service-oriented cloud-based management system for the internet-of-drones. In *The 32nd ACM Symposium on Applied Computing*.
18. Dronemap Planner Demos. 2016. <http://wiki.coins-lab.org/index.php?title=Dronemap>. Accessed 6 Nov 2016.
19. Telerobot. 2016. <http://telerobot.mech.uwa.edu.au/Telerobot/index.html> Accessed 10 May 2016.
20. Marín, R., P. Sanz, and A. del Pobil. 2003. The uji online robot: An education and training experience. *Autonomous Robots* 15 (3): 283–297. <http://dx.doi.org/10.1023/A:1026220621431>.

ROS and Docker

Ruffin White and Henrik Christensen

Abstract In this tutorial chapter we'll cover the growing intersection between ROS and Docker, showcasing new development tools and strategies to advance robotic software design and deployment within a ROS/Gazebo context by utilizing advances in Linux containers. Tutorial examples here will focus on robotics software development for education, research, and industry, specifically: constructing repeatable and reproducible environments, leveraging software defined networking, as well as running and shipping portable ROS applications.

Keywords ROS · Docker · Repeatability · Reproducibility · Node networking · Portable deployment · Distributed computing

1 Introduction

The ROS ecosystem builds from a fast growing, open source, continuously evolving community of newly released distros, updated dependencies, and deprecated packages. This can prove troublesome for teaching, developing or even publishing while using any ROS code-base. Additionally, practitioners in the multidisciplinary field of robotics are certainly not solely composed of trained software engineers. Building, running and shipping complex ROS apps and services can be a daunting endeavor for non-experts, presenting a formidable learning curve encountered by those proceeding beyond beginner tutorials.

Robotics still lacks a suitable work flow with respect to continuous integration and test verification [3, 4]. Issues with repeatable and reproducible environmental setups can make developing robotics software with collaborators non trivial, discouraging code-reuse thus prompting much unnecessary reinvention. Many however are finding the use of Docker to be a helpful tool to tackle these challenges [2, 8].

R. White (✉) · H. Christensen
Contextual Robotics Institute, University of California,
9500 Gilman Drive, La Jolla, San Diego, CA 92093, USA
e-mail: rwhitema@eng.ucsd.edu
URL: <http://jacobsschool.ucsd.edu/contextualrobotics>

One of the largest robotic planning projects, MoveIt! [7], now uses ROS with Docker as a tool for continuous integration and collaboration between maintainers. Containers have enabled the MoveIt! community to perform faster and more frequent tests with the same CI resources, as well as enabling maintainers to build patches and review pull requests for various releases and branches without cross contaminating their own development environments.

Additionally, new projects such as Secure ROS (SROS) have also found uses for containers. SROS is an addition to the ROS API and ecosystem to support modern cryptography and security measures to improve the state of security for future robotics subsystems [9]. SROS currently uses Docker to distribute portable run-time and development environments, inviting the community to quickly interact and contribute with the latest progress of the project.¹

In work with maritime autonomy [6], authors utilize containers as a means to deploy experimental algorithms to autonomous robotic systems. The methods presented enable the Naval Surface Warfare Center to minimize the level of effort and lead time required to repeatedly re-baseline unmanned underwater vehicles, high cost equipment that is also time-shared among many other research groups. Authors show results of a quickly-deployable, easily reconfigurable, and vehicle-agnostic autonomy solution that helps maximize the usage of limited resources.

The topics and examples covered to address the listed challenges are overviewed, ordered in steadily advancing complexity, and are intended to guide the reader from novice to well informed. However to best apprehend the material presented, readers should have prior developer level experience with ROS, e.g. building packages from source. Having a basic user level acquaintance with Docker, e.g. being familiar with at least the most common Docker CLI commands, e.g. `docker pull`, `build`, `run`, may also helpful but not required.

1.1 Overview

- **Background:** What is Docker? What are Linux Containers vs. Virtual Machines? What is the dependency matrix from hell? What are the official ROS and Gazebo DockerHub repos? What role can Docker play in robotics and how does its development mirror what is accruing in the web development community?
- **Setup:** What minimum requirements are needed? Is my OS and hardware supported? How can I download various releases of ROS using Docker?
- **Examples:**
 - **Education | Container and Image Basics:** How can Docker soften the learning curve for ROS and Gazebo by simplifying the setup process? How can the community share working examples or reproduce broken ones for collaborative

¹<http://wiki.ros.org/SROS/Installation/Docker>.

- debugging? How can containers provide fail-fast learn-fast disposable work-spaces, encouraging experimentation without hesitation?
- **Industry | Networking and Deployment:** How can nodes be distributed across machines without local access or VPNs? How can compose files encapsulate the start-up of complex launch procedures?
 - **Research | Using Devices and Peripherals:** How can GPUs and hardware peripherals be mounted for deep learning and perception tasks? How can images serve to archive code alongside publications? How can we quickly collaborate with others by sharing complex compilation setups?
- **Notes:** What are some caveats, best practices, and suggested third party tools to watch out for?

2 Background

Over the last few years an insurgence of Linux containers has taken root in the world of software development. Linux containers themselves have existed for some time, but until recently creating and managing them has not always been simple or straightforward.

However, thanks in part to improved tooling and a simplified user experience offered by growing open source projects such as Docker, this method of building and distributing software is beginning to change how we work and collaborate. And with the establishment of the Open Container Initiative, a open governance structure formed under the auspices of the Linux Foundation and backed by much of the web industry, open industry standards around container formats and runtimes will inevitably continue to mature.

A Linux container is basically an operating-system-level virtualization method for running multiple isolated Linux systems on a control host using a single Linux kernel, offering an environment similar to a Virtual Machine, but without the overhead that comes with running a separate kernel and simulating all the hardware and networking. Simply speaking, containers sit between the spectrum of chroots and VMs, being slightly closer to the former.

Docker implements a client-server system where the Docker daemon (or engine) runs on a host and it is accessed via a client. The client, which may or may not be on the same host, can control an engine (or even a swarm of engines on multiple hosts) to spawn, manage, and network multiple containers. Containers run from a thin writable layer on top of a specified image, where an image is a list of read-only layers that represent filesystem differences. A container's writable layer can be committed to construct a new read-only image, while common read-only layers can be shared across images.

During the ROS release of Jade Turtle in 2015, Open Source Robotic Foundation (OSRF) and authors collaborated to publish an Official Docker Hub repository for ROS² and Gazebo³ [8]. These Dockerized images are intended to provide a simplified and consistent foundation to build and deploy robotic applications. Built from the official Ubuntu image and OSRF’s official Debian packages, the images serve as a quick and secure vector for releases.

Developing such complex robotic systems with cutting edge implementations of newly published algorithms remains challenging, as repeatability and reproducibility of robotic software can fall to the wayside in the race to innovate and satisfy the ever growing dependency matrix from hell—the various permutations of architectures, peripherals, and libraries that exist in robotics. With the added difficulty in coding, tuning and deploying multiple software components that span many engineering disciplines, a more collaborative approach becomes attractive. However, the technical difficulties in sharing and maintaining a collection of software over multiple robots and platforms has for some time exceeded the time and effort that many smaller labs and businesses could afford.

With the advance and standardization of software containers, roboticists are primed to acquire a host of improved developer tooling for building and shipping software. To help alleviate the growing pains and technical challenges of adopting new practices, we have focused on providing an official resource for using ROS with these new technologies.

3 Setup

3.1 Requirements

For this tutorial we’ll be leveraging many of the modern features found in recent releases of the Docker engine, as well as additional tools surrounding the larger Docker ecosystem. The exact versions used while authoring these examples are shown below, however later versions should also function since many of these features are now quite stable and have matured.

```

1 $ docker -v
2 Docker version 1.11.1, build 5604cbe
3 $ docker-compose -v
4 docker-compose version 1.6.2, build 4d72027

```

Optional requirements include a local installation of ROS on the same machine you plan to have host your Docker engine/install, since all our encountered

²https://hub.docker.com/_/ros/.

³https://hub.docker.com/_/gazebo/.

examples will be “containerized”. Since a demonstration interconnecting ROS from the host will be provided, a matching ROS installation may be useful for using GUI’s and visualizations. Note however that any mention of a required host operating system has been omitted, because, unlike ROS, Docker can be easily installed on many distributions that support any modern Linux kernel (current minimum 3.10). Mac-OSX and MS-Windows are also Docker supported, but require a VM to service a running Linux kernel, however this and the general 64-bit requirement may change. A host installation of a recent LTS such as Ubuntu 14.04 or 16.04 is advised, especially if you wish to install the only other optional requirement—Nvidia driver and `nvidia-docker`⁴ plugin for improved performance of the `ros_caffe` example using CUDA enabled hardware.

3.2 Installation

You can follow the proscribed and up-to-date installation instructions⁵ for your OS from Docker’s Document website, or if your distribution supports deb/rpm, you may use the script provided by Docker to install the engine (compose installed separately⁶):

```
curl -fsSL https://get.docker.com/ | sh  
sudo service docker restart
```

Additionally, to avoid sudo while commanding Docker, you may choose to add your user to the `docker` Unix group. Take care however, as the Docker group is equivalent to `root`. After installing Docker, you’ll want to install Docker-compose from the same documentation website, permitting you to succinctly describe and launch our later examples using short yaml compose files. To test Docker, and download some useful images for later, you needn’t run more than this command from line 1:

⁴<https://github.com/NVIDIA/nvidia-docker>.

⁵https://docs.docker.com/linux/step_one/.

⁶<https://docs.docker.com/compose/>.

```

1 $ docker run -it --rm ros roscore
2 Unable to find image 'ros:latest' locally
3 latest: Pulling from library/ros
4 943c334059c7: Pull complete
5 ...
6 f9b3f610dc9c: Pull complete
7 Digest: sha256:e1c7...c2b1
8 Status: Downloaded newer image for ros:latest
9 ... logging to /root/.ros/log/ab44...0002/roslaunch-c6619a48f368-1.log
10 Checking log directory for disk usage. This may take awhile.
11 Press Ctrl-C to interrupt
12 Done checking log file disk usage. Usage is <1GB.
13 started roslaunch server http://c6619a48f368:35403/
14 ros_comm version 1.11.19
15 SUMMARY
16 =====
17 PARAMETERS
18 * /rosdistro: indigo
19 ...

```

You should see an output very similar to the above, where Docker simply runs roscore from the ros:latest image. If the Docker engine can not find this image locally, it will automatically pull what it needs from DockerHub. Once roscore is running, and because we've specified this session to be interactive and clean up afterward, you can kill roscore, and thereby stop and remove the originating container using Ctrl-C.

3.3 Building

Before we proceed with any examples, let's take a moment to walk through a Dockerfile used to build the tagged images you just downloaded. The hierarchy of available official tags is keyed to the most common ROS meta-packages, designed to have small disk footprints and simple configurations:

- `ros-core`: minimal bare-bones ROS install
- `ros-base`: basic libraries (tagged with distro name, newest LTS as `latest`)
- `robot`: basic install for robot platforms
- `perception`: basic install for perception tasks

The rest of the common meta-packages such as `desktop` and `desktop-full` are hosted on automatic build repos under OSRF's DockerHub profile. These last meta-packages include graphical libraries and hook to a host of other large dependencies such as X11, X server, etc. In the interest of keeping the official library images lean and secure, the desktop images are just hosted by OSRF.

```

1 # This is an auto generated Dockerfile for ros:indigo-ros-core
2 # generated from
3   → templates/docker_images/create_ros_core_image.Dockerfile.em
4 # generated on 2016-04-26 21:55:54 +0000
5 FROM ubuntu:trusty
6 MAINTAINER Tully Foote tfoote+buildfarm@osrfoundation.org

```

First thing you'll notice is the auto generated comments that specify the date the Dockerfile was generated as well as the template used to derive it. There are many ROS Dockerfiles, one for each tagged image under the official DockerHub repo, so a template engine is used to generate and maintain all the Dockerfiles within the osrf/docker_images⁷ GitHub repo. The template engine itself is made available at the osrf/docker_templates,⁸ providing a means to programmatically generate custom ROS related Dockerfiles, to be simultaneously leveraged within the second generation ROS build farm using Docker.

The final two lines, 4 and 5, define the originating parent image and maintainer contact information. Currently all ROS images are built from LTS Ubuntu images also provided by the Official DockerHub Library. Note here that the OS minor version number is not necessarily specified. This permits the Official ROS images to quickly rebuild with the latest minor release updates to the Ubuntu image. It is wise to consider this compromise of abstraction vs. specificity, e.g. with ROS version dependent code-base; defining only `FROM ros` may break the application upon the next LTS release, bumping up the implicit `latest` tag.

```

7 # setup environment
8 RUN locale-gen en_US.UTF-8
9 ENV LANG en_US.UTF-8
10
11 # setup keys
12 RUN apt-key adv --keyserver ha.pool.sks-keyservers.net --recv-keys
   → 421C365BD9FF1F717815A3895523BAEEB01FA116
13
14 # setup sources.list
15 RUN echo "deb http://packages.ros.org/ros/ubuntu trusty main" >
   → /etc/apt/sources.list.d/ros-latest.list

```

Above we see the basic ROS installation setup for Ubuntu. The base image used derives from a modified version of Ubuntu's Cloud Images, rather than the full desktop install, so we need to configure some locales and expected variables for ROS environments. Here we also use a key-server with a high-availability server pool to add the ROS repository credentials. Note that a Dockerfile should be written to help mitigate any man-in-the-middle attacks during the build process: using https;

⁷https://github.com/osrf/docker_images.

⁸https://github.com/osrf/docker_templates.

importing PGP keys full fingerprints to check package signing; embedding check sums directly in the Dockerfile if PGP signing is not provided.

```

17 # install bootstrap tools
18 RUN apt-get update && apt-get install --no-install-recommends -y \
19     python-rosdep \
20     python-rosinstall \
21     python-vcstools \
22     && rm -rf /var/lib/apt/lists/*
23
24 # bootstrap rosdep
25 RUN rosdep init \
26     && rosdep update

```

Next, some Python dependencies are bootstrapped for the rosdep tool. Note the style that every `apt-get update/install` is written on the same line and superseded with `rm -rf /var/lib/apt/lists/*`. This is roughly the opposite of `apt-get update` since it ensures that the resulting layer doesn't include the extra ~8 MB of APT package list data. This also enforces appropriate `apt-get update` usage, preventing images from containing stale package listing data.

```

28 # install ros packages
29 ENV ROS_DISTRO indigo
30 RUN apt-get update && apt-get install -y \
31     ros-indigo-ros-core=1.1.4-0* \
32     && rm -rf /var/lib/apt/lists/*

```

Now we finally install the particular ROS meta package that the deriving image is tagged for. Here we intend that rebuilding the same Dockerfile should result in the same version of the image being packaged. And an official repo Dockerfile will serve as a base image for all those preceding, so being version explicit is valuable to the repeatability and transparency of the builds.

If an installation version can not be satisfied, the build should fail outright, preventing an inadvertent rebuild of a Dockerfile containing something other than what is given by its tag. For dependent packages installed by `apt` there's usually no need to pin them to a version, but this is something you may want to consider. An additional benefit to pinning the version is this provides the maintainer a chance to preserve and break the build cache where needed, when for instance updating a package with a version bump. Updating of environment variables within Dockerfiles is also sometimes used for the same purpose.

```
34 # setup entrypoint
35 COPY ./ros_entrypoint.sh /
36
37 ENTRYPOINT ["/ros_entrypoint.sh"]
38 CMD ["bash"]
```

Lastly, the default entrypoint is configured and then the default run command defined. Here, the entrypoint simply sources ROS's own setup script, as shown below. The entrypoint can be amended to source your own ROS workspace, enabling brief Docker run commands to launch your own ROS package.

```
1 #!/bin/bash
2 set -e
3
4 # setup ros environment
5 source "/opt/ros/$ROS_DISTRO/setup.bash"
6 exec "$@"
```

4 Examples

Now let's cover some example use cases for using Docker with ROS. All example code and detailed tutorials will be made freely available in the corresponding public GitHub repo.⁹

4.1 Education

Let's take the scenario that you are the instructor for a robotics course utilizing ROS. It's just the beginning of the course, but you would like to give the students a working ROS tutorial to keep them engaged. However you'd also like to prevent the first half of the lecture and following office hours from diverging into an arduous ROS install-fest. We'll make an optimistic assumption your students already have a working Linux or VM install with Docker, but not necessarily a homogeneous set of releases or distributions. However, we'd also like to avoid breaking anything in any way, due in part to their other coursework dependencies and setups.

⁹https://github.com/ruffsl/ros_docker_demos.

Let's begin by giving the students a small Dockerfile to build our example:

```

1  FROM ros:indigo
2  RUN apt-get update && apt-get install -y \
3      build-essential \
4      && rm -rf /var/lib/apt/lists/
5  ENV CATKIN_WS=/root/catkin_ws
6  RUN rm /bin/sh \
7      && ln -s /bin/bash /bin/sh

```

Here we'll start from an official ROS image and install the dependencies we know the students will need. The official images cater to runtime deployments, but can be easily extended for our build requirements. We'll also define our catkin workspace directory, as well as switch to bash for sourcing the environment needed with catkin. One could instead COPY and RUN an executable bash script in the Docker build context, that being same directory as the Dockerfile, but we'll swap the shell to keep everything self-contained in the Dockerfile.

```

8   RUN source /ros_entrypoint.sh \
9     && mkdir -p $CATKIN_WS/src \
10    && cd $CATKIN_WS/src \
11    && catkin_init_workspace \
12    && git clone https://github.com/ros/ros_tutorials.git \
13    && touch ros_tutorials/turtlesim/CATKIN_IGNORE

```

Next we'll show the students how to create a catkin workspace and clone the source for the example. Note that we're ignoring one package, as we've omitted to download and install any large GUI desktop dependencies such as QT.

```

14  RUN source /ros_entrypoint.sh \
15    && cd $CATKIN_WS \
16    && catkin_make
17  RUN sed -i \
18    '/source "/opt/ros/$ROS_DISTRO/setup.bash"/a source
19    "\$CATKIN_WS/devel/setup.bash" \
/rosmake_entrypoint.sh

```

Finally we'll build the tutorial package and include the setup of our catkin workspace into the original entrypoint. Students can then be instructed to start with the following two commands in the same path they've saved the Dockerfile:

```
1 $ docker build --tag=ros:tutorials .
2 Sending build context to Docker daemon 2.56 kB
3 Step 1 : FROM ros:indigo
4    --> e7ccb7b11eeb
5 ...
6 Successfully built f2cc5810fb94
7 $ docker run -it ros:tutorials bash -c "roscore & rosrund
8   ↪ roscpp_tutorials listener & rosrund roscpp_tutorials talker"
9 ...
10 [ INFO] [1462299420.261297314]: hello world 5
11 [ INFO] [1462299420.261495662]: I heard: [hello world 5]
12 [ INFO] [1462299420.361333784]: hello world 6
13 ^C[ INFO] [1462299420.361548617]: I heard: [hello world 6]
14 [rosout-1] killing on exit
```

From here, students can swap out the URLs for their own repositories and append additional dependencies. Should students encounter any build or runtime errors, Dockerfiles and/or images could be shared (from Git Hub and/or Docker Hub) with the instructor or other peers on say answers.ros.org to serve as a minimal example, capable of quickly replicating the errors encountered for further collaborative debugging.

What we've shown so far has been a rather structured work-flow from build to runtime, however containers also offer a more interactive and dynamic work-flow as well. As shown from this tutorial video,¹⁰ we can interact with containers directly. A container can persist beyond the life cycle of its starting process, and is not removed until the docker daemon is directed to do so. Naming or keeping track of your containers affords you the use of isolated ephemeral work-spaces in which to experiment or test, stopping and restarting them as needed.

Note that you should avoid using containers to store a system state or files you wish to preserve. Instead, a developer may work within a container iteratively, progressively building the larger application in increments and taking periodic respites to commit the state of their container/progress to a newly tagged image layer. This could be seen as a form of state wide revision control, with save points allowing the developer to reverse changes by simply spawning a new container from a previous layer. All the while the developer could also consolidate his progress by noting the setup procedure within a new Dockerfile, testing and comparing it against the lineage of working scratchwork images.

4.2 Industry

In our previous education example, it was evident how we simply spawned all the tutorial nodes for a single bash process. When this process (PID 1) is killed, the con-

¹⁰<https://youtu.be/9xqekKwzmV8>.

tainer is also killed. This explains the popular convention of keeping to one process per container, as it is indicative to modern paradigm of microservices architecture, etc. This is handy should we desire the life-cycles of certain deployed ROS nodes to be longer than others. Let's revisit the previous example utilizing software defined networking to interlink the same ROS nodes and services, only now, running from separate containers.

Within a new directory, `foo`, we'll create a file named `docker-compose.yml`:

```

1 version: '2'
2
3 services:
4   master:
5     image: ros:indigo
6     environment:
7       - "ROS_HOSTNAME=master.foo_default"
8     command: roscore
9
10  talker:
11    build: talker/.
12    environment:
13      - "ROS_HOSTNAME=talker.foo_default"
14      - "ROS_MASTER_URI=http://master.foo_default:11311"
15    command: rosrun roscpp_tutorials talker
16
17  listener:
18    build: listener/.
19    environment:
20      - "ROS_HOSTNAME=listener.foo_default"
21      - "ROS_MASTER_URI=http://master.foo_default:11311"
22    command: rosrun roscpp_tutorials listener

```

With this compose file, we have encapsulated the entire setup and structure of our simple set of ROS 'microservices'. Here, each service, (master, talker, listener), will spawn a new container named appropriately, originating from the `image` designated or Dockerfiles specified in the `build` field. Notice that the `environment` fields configure the ROS network variables to match each service's domain name under the `foo_default` network named by our project's directory. The `foo_default` name-space can be omitted, as the default DNS resolution within the `foo_default` will resolve using the local service or container names. Still, remaining explicit helps avoid collisions while adding host enabled DNS resolution (later on) over multiple Docker networks.

Before starting up the project, we'll also copy the same Dockerfile from the previous example into the project's `talker` and `listener` sub-directories. With this, we can start up the project detached, and then monitor the logs as below:

```

1 ~/foo$ docker-compose up -d
2 Creating foo_master_1
3 Creating foo_listener_1
4 Creating foo_talker_1
5
6 ~/foo$ docker-compose logs
7 Attaching to foo_talker_1, foo_master_1, foo_listener_1
8 ...
9 talker_1 | [ INFO] [1462307688.323794165]: hello world 42
10 listener_1 | [ INFO] [1462307688.324099631]: I heard: [hello world 42]

```

Now let's consider the example where we'd like to upgrade the ROS distro release used for just our talker service, leaving the rest of our ROS nodes running and uninterrupted. We'll use Docker-compose to recreate our new talker service:

```

12 ~/foo$ docker exec -it foo_talker_1 printenv ROS_DISTRO
13 indigo
14
15 ~/foo$ sed -i -- 's/indigo/jade/g' talker/Dockerfile
16
17 ~/foo$ docker-compose up -d --build --force-recreate talker
18 Building talker
19 Step 1 : FROM ros:jade
20 ...
21 Successfully built 3608a3e9e788
22 Recreating foo_talker_1
23
24 ~/foo$ docker exec -it foo_talker_1 printenv ROS_DISTRO
25 jade

```

Here we first check the ROS release used in the container, and change the version used in the originating Dockerfile for the talker service. Next we use some shorthand flags to inform Docker-compose to re-bring-up the talker service by recreating a new talker container by rebuilding the talker image. We then check the ROS distro again and see the reflected update. You may also go back to docker compose logs and find that the counter in the published message has been reset.

From here on we can abstract our interaction with the docker engine, and instead point our client towards a Docker Swarm,¹¹ a method for one client to spin up containers from a cluster of Docker engines. Normally a tool such as Docker Machine¹² can be used to bootstrap a swarm and define a swarm master. This entails provisioning and networking engines from multiple hosts together, such that requested containers can be load balanced across the swarm, and containers running from different hosts can securely communicate.

¹¹<https://docs.docker.com/swarm>.

¹²<https://docs.docker.com/machine>.

4.3 Research

Up to this point, we've considered relatively benign Docker enabled ROS projects where our build dependencies were fairly shallow, simply those accrued through default apt-get, and run time dependencies without any external hardware. However, this is not always the case when an original project builds from fairly new and evolving research. Let's assume for the moment you're a computer vision researcher, and a component of your experiment utilises ROS for image acquisition and transport culminating into live published classification probabilities from a trained deep convolutional neural network (CNN). Your bleeding edge CNN relies on a specific release of parallel programming libraries, not to mention the supporting GPU and imaging capture peripheral hardware.

Here we'll demonstrate the reuse of existing public Dockerfiles to quickly obtain a running setup, stringing together the latest published images with preconfigured installations of CUDA/CUDNN, and meticulous source build configurations for Caffe [5]. Specifically we'll use a Caffe image from a Docker Hub provided by the community. This image then-in-turn builds from a CUDA/CUDNN image from NVIDIA, that then-in-turn uses the official Ubuntu image on Docker Hub. All necessary Dockerfiles are made available through the respective Docker Hub repos, so that you may build the stack locally if you choose (Fig. 1).

However, in the interest of build time and demonstration, we literally build FROM those before us. This involves a small modification and addition to the Dockerfile for `ros-core`. By simply redirecting the parent image structure of the Dockerfile to point to the end-chain image, with each image in the prior chain encompassing a component of our requirements, we can easily customize and concatenate the lot to describe and construct an environment that contains just what we need.

For brevity, detailed and updated documentation/Dockerfiles are kept to same repository as the ros-caffe project.¹³ A link to a video demonstration¹⁴ can also be found at the project repo. Shown here will be the notable key-points in pulling/running the image classification node from your own Docker machine.

First we'll modify the `ros-core` Dockerfile to build from an image with Caffe built using CUDA/CUDNN, in this case we'll use a popular set of maintained automated build repos from Kai Arulkumaran [1]:

```
1 FROM kaixhin/cuda-caffe
```

Next we'll amend the RUN command that installs ROS packages to include the additional ROS dependencies for our `ros_caffee` example package:

¹³https://github.com/ruffsl/ros_caffe.

¹⁴<https://youtu.be/T8ZnnTpriC0>.



Fig. 1 A visual of the base image inheritance for the `ros_caffe:gpu` image

```

24 # install ros packages
25 RUN apt-get update && apt-get install -y \
26     ros-${ROS_DISTRO}-ros-core \
27     ros-${ROS_DISTRO}-usb-cam \
28     ros-${ROS_DISTRO}-rosbridge-server \
29     ros-${ROS_DISTRO}-roswww \
30     ros-${ROS_DISTRO}-mjpeg-server \
31     ros-${ROS_DISTRO}-dynamic-reconfigure \
32     python-twisted \
33     python-catkin-tools && \
34     rm -rf /var/lib/apt/lists/*

```

Note the reuse of the `ROS_DISTRO` variable within the Dockerfile. When building from the official ROS image, this helps makes your Dockerfile more adaptable, allowing for easy reuse and migration to the next ROS release, just by updating the base image reference.

```

40 # setup catkin workspace
41 ENV CATKIN_WS=/root/catkin_ws
42 RUN mkdir -p $CATKIN_WS/src
43 WORKDIR $CATKIN_WS/src
44
45 # clone ros-caffe project
46 RUN git clone https://github.com/ruffsl/ros_caffe.git
47
48 # Replacing shell with bash for later source, catkin build commands
49 RUN mv /bin/sh /bin/sh-old && \
50     ln -s /bin/bash /bin/sh
51
52 # build ros-caffe ros wrapper
53 WORKDIR $CATKIN_WS
54 ENV TERM xterm
55 ENV PYTHONIOENCODING UTF-8
56 RUN source "/opt/ros/${ROS_DISTRO}/setup.bash" && \
57     catkin build --no-status && \
58     ldconfig

```

Finally, we can simply clone and build the catkin package. Note the use of `WORKDIR` to execute `RUN` commands from the proper directories, avoiding the need

to hard-code the paths in the command. The optional variables and arguments around the catkin build command are used to clear a few warnings and printing behaviors the catkin tool has while running from a basic terminal session.

Now that we know how this is all built, let's skip ahead to running the example. You'll first need to clone the project's git repo and then download the caffe model to acquire the necessary files to run the example network, as explained in the project README on the github repo. We can then launch the node by using the `run` command to pull the necessary images from the project's automated build repo on Docker Hub. The run script within the docker folder shows an example of using the GPU version:

```

1 nvidia-docker run \
2   -it \
3   --publish 8080:8080 \
4   --publish 8085:8085 \
5   --publish 9090:9090 \
6   --volume="${PWD}/../ros_caffe/data:
7     /root/catkin_ws/src/ros_caffe/ros_caffe/data" \
8   --device /dev/video0:/dev/video0 \
  ruffsl/ros_caffe:gpu roslaunch ros_caffe_web ros_caffe_web.launch

```

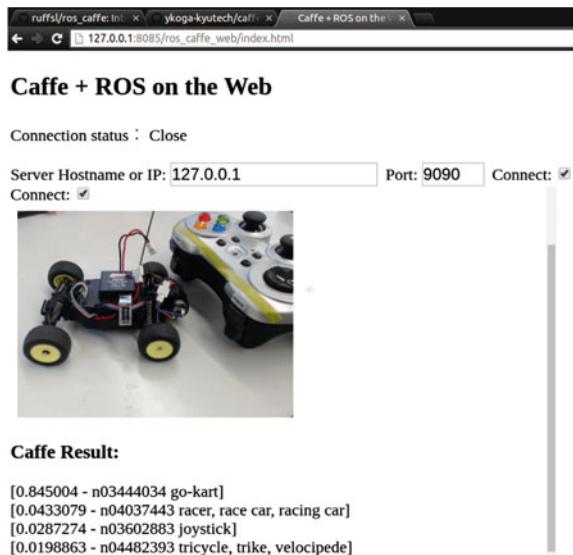
The Nvidia Docker plug-in is a simple wrapper function around Docker's own `run` call, injecting additional arguments that include mounting the device hardware and driver directories. This permits our CUDA code to function easily within the container without necessarily baking the version specific Nvidia driver within the image itself. You can easily see all implicit properties affected by using the `docker inspect` command with the name of the container generated and notice devices such as `/dev/nvidia0` and mounted volume driver named after your graphics driver version. Be sure you have enough available VRAM, about 500 MB to load this network. You can check your memory usage using `nvidia-smi`. If you don't have a GPU, then you may simply alter the above command by changing `nvidia-docker` to just `docker`, as well as swapping the `:gpu` image tag with `:cpu`.

The rest of the command is relatively clear; specifying the port mapping for the container to expose web ros interface through localhost as well as mounting the volume including our downloaded caffe model. The device argument here is used to provide the container with a video capture device; one can just as easily mount `/dev/bus/usb` or `/dev/joy0` for other such peripherals. Lastly we specify the image name and `roslaunch` command. Note that we can use this command as is since we've modified the image's entrypoint to source our workspace as well.

Once the ros-caffe node is running, we can redirect our browser to the local URL¹⁵ to see a live video feed of the current published image and label prediction from the neural network as shown in Fig. 2.

¹⁵http://127.0.0.1:8085/ros_caffe_web/index.html.

Fig. 2 A simple ros-caffe web interface with live video stream and current predicted labels, published from containerized nodes with GPU and webcam device access



4.4 Graphical Interfaces

One particular aspect routinely utilized by the ROS community includes all the tools used to introspect and debug robotic software through the use of graphical interfaces, such as *rqt*, *Rviz*, or *gzviewer*. Although using graphical interfaces is perhaps outside of the original use case of Docker, it is perfectly possible and in-fact relatively viable for many applications. Thanks to Linux's pervasive use of the files system for everything, including video and audio devices, we can expose what we need from the host system to the container.

Although the easiest means of permitting the use of a GUI may be to simply use the host's installation of ROS or Gazebo, as demonstrated in this video,¹⁶ and thus set the master URI or server address to connect to the containers via virtual networking and DNS containers described earlier, it may be necessary to run a GUI from within the container, be it custom dependencies or accelerated graphics. There are of course a plethora of solutions for various requirements and containers, ranging from display tunneling over SSH, VNC client server sessions, or directly mounting X-server unix sockets and forwarded alsound or pulseaudio connections. Each method of course comes with its own pros and cons, and in light of this evolving frontier, the reader is encouraged to read on ROS Wiki's Docker page¹⁷ in order to follow the latest tutorials and resources.

Below is a brief example of the turtlebot demo using Gazebo and RVIZ GUIs via X Server sockets and graphical acceleration from within a container. First we'll

¹⁶https://youtu.be/P_phnA57LM.

¹⁷<http://wiki.ros.org/docker>.

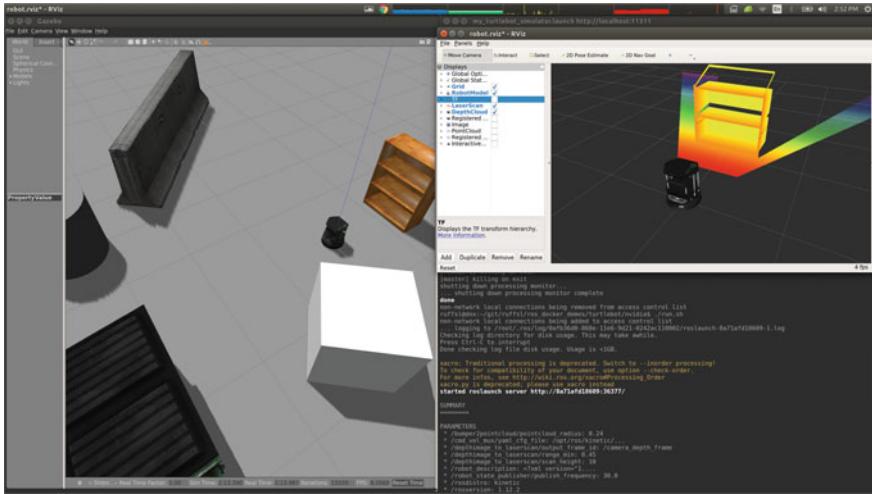


Fig. 3 An example of containerized GUI windows rendered from within the host's desktop environment

build from OSRF's ROS image using the `desktop-full` tag, as this will have the Gazebo and RVIZ pre-installed. Then we'll add the turtlebot packages, the necessary world models, and custom launch file (Fig. 3).

```

1  FROM osrf/ros:kinetic-desktop-full
2
3  # install turtlebot simulator
4  RUN apt-get update && apt-get install -y \
5      ros-${ROS_DISTRO}-turtlebot* \
6      && rm -rf /var/lib/apt/lists/*
7
8  # Getting models from [http://gazebosim.org/models/]. This may take a few
9  # seconds.
10 RUN gzserver --verbose --iters 1
11   -- /opt/ros/${ROS_DISTRO}/share/turtlebot_gazebo/
12   worlds/playground.world
13
14 # install custom launchfile
15 ADD my_turtlebot_simulator.launch /

```

Note the single iteration of `gzserver` with the default turtlebot world used to prefetch the model from the web and into the image. This helps cuts Gazebo's start-up time, saving each deriving container from downloading and initializing the needed model database at runtime. The launchfile here is relatively basic, launching the simulation, the visualisation, and a user control interface:

```

1 <launch>
2   <include file="$(find
3     → turtlebot_gazebo)/launch/turtlebot_world.launch" />
4   <include file="$(find
5     → turtlebot_teleop)/launch/keyboard_teleop.launch" />
6   <include file="$(find
7     → turtlebot_rviz_launchers)/launch/view_robot.launch" />
8 </launch>

```

For hardware acceleration using discreet graphic for Intel, we'll need to also add some common Mesa libraries:

```

14 # Add Intel display support by installing Mesa libraries
15 RUN apt-get update && apt-get install -y \
16   libgl1-mesa-glx \
17   libgl1-mesa-dri \
18 && rm -rf /var/lib/apt/lists/*

```

For hardware acceleration using dedicated graphics for Nvidia, we'll need to add some hooks and variables instead for the nvidia-docker plugin:

```

14 # Add Nvidia display support by including nvidia-docker hooks
15 LABEL com.nvidia.volumes.needed="nvidia_driver"
16 ENV PATH /usr/local/nvidia/bin:${PATH}
17 ENV LD_LIBRARY_PATH
  → /usr/local/nvidia/lib:/usr/local/nvidia/lib64:${LD_LIBRARY_PATH}

```

Note how any deviations between the two setups was left to the last few lines of the Dockerfile, specifically any layers of the image that will no longer be hardware agnostic. This enables you to share as much of the common previous layers between the two tags as possible, saving disk space, and shortening build times by reusing the cache. Finally we can launch GUI containers by permitting access to the X Server, then mounting the Direct Rendering Infrastructure and unix socket:

```

1 xhost +local:root
2
3 # Run container with necessary Xorg and DRI mounts
4 docker run -it \
5   --env="DISPLAY" \
6   --env="QT_X11_NO_MITSHM=1" \
7   --device=/dev/dri:/dev/dri \
8   --volume=/tmp/.X11-unix:/tmp/.X11-unix \
9   ros:turtlebot-intel \
10  rosrun my_turtlebot_simulator.launch
11
12 xhost -local:root

```

The environment variables are used to inform GUIs of the display to use, as well as fix a subtle QT rendering issue. For Nvidia, things look much the same, except for use of the nvidia-docker plugin to add the needed device and volume arguments:

```

1 xhost +local:root
2
3 # Run container with necessary Xorg and GPU mounts
4 nvidia-docker run -it \
5   --env="DISPLAY" \
6   --env="QT_X11_NO_MITSHM=1" \
7   --volume=/tmp/.X11-unix:/tmp/.X11-unix \
8   ros:ubuntu-nvidia \
9   rosrun my_turtlebot_simulator.launch
10
11 xhost -local:root

```

You can view an example using this method from the previous linked demo video for ros-caffe, or a older GUI demo video¹⁸ now made simpler via the nvidia-plugin for qualitative evaluation.

5 Notes

As you take further advantage of the growing Docker ecosystem for your robotics applications, you may find certain methods and third-party tools useful in continuing simplifying or becoming more efficient in common development tasks while using Docker. Here we'll cover just a few helpful practices and tools most relevant for ROS users.

5.1 Best Practices and Caveats

There are many best practices to consider while using Docker, and as with any new technology or paradigm, we need to know the gotchas. While much is revealed within Docker's own tutorial documentation and helpful posts within the community,¹⁹ there are a few subjects that are more pertinent to ROS users than others.

ROS is a relatively large ‘stack’ as compared to other commonly used codebases with Docker, such as smaller lightweight web stacks. If the objective is to distribute and share Robotics based images using ROS, it’s worthwhile to be mindful of the size of the images you generate to be bandwidth considerate. There are many ways to mitigate bloat from an image through careful thought while structuring the

¹⁸<https://youtu.be/djLKMdMsdxM>.

¹⁹https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/.

Dockerfile. Some of this was described while going over the official ROS Dockerfile, such as always removing temporary files before completion of a layer generated from each Docker command.

However there are a few other caveats to consider concerning how a layer is constructed. One being to never change the permissions of a file inside a Dockerfile unless unavoidable; consider using the entrypoint script to make the changes if necessary for runtime. Although a git/Docker comparison could be made, Docker only notes what files have changed, not necessarily how the files have been modified inside the layer. This causes Docker to replicate/replace the files while creating a new layer, potentially doubling the size if you're modifying large files, or potentially worse, every file.

Another way keep disk size down can be to flatten the image, or certain spans of layers. This however prevents the sharing of intermediate layers among commonly derived images, a method Docker uses to minimize the overall disk usage. Flattening images also only helps in squashing large modifications to image files, but does nothing if the squashed file system is just inherently large.

When building Dockerfiles, you'll want to be considerate of the build context, i.e. the parent folder of the Dockerfile itself. For example, it's best to build a Dockerfile from a folder that includes just the files and folders you'd like to ADD or COPY into the image. This is because the docker client will tar/compressing the directory (and all subdirectories) where you executed the build and send it to the docker daemon. Although files that are not referenced to will not be included in the image, building a Dockerfile from say your /root/, /home/ or /tmp/ directory for example would be unwise, as the amount of unnecessary data sent to the daemon would slow/kill the build. A .dockerignore could also be used to avoid this side effect.

Finally, a docker container should not necessarily be thought of as a *complete* virtual environment. As opposed to VM's with their own hypervized kernel and start-up processes, the only process that runs within the container is that which you command. This means that there is no system init, up-start or system starting syslog, cron jobs and daemons, or even reaping orphaned zombie processes. This is usually ok, as a container's life cycle is quite short and we normally only want to execute what we specify. However, if you intend to use containers as a more full fledged system requiring say proper signals handling, consider using minimal init system for Linux containers such as *dumb-init*.²⁰ For most cases with ROS users, roslaunch does a rather good job signalling child processes and thus serves as a fine anchor for a container's PID 1, and so simply running multiple ROS nodes per container is reasonable. For those more concerned using custom launch alternatives, a relevant post here²¹ expands on this subject further.

²⁰<https://github.com/Yelp/dumb-init>.

²¹<https://blog.phusion.nl/2015/01/20/docker-and-the-pid-1-zombie-reaping-problem>.

5.2 *Transparent Proxy*

One task you may find yourself performing frequently while building and tweaking images, especially if debugging say minimum dependency sets, is downloading and installing packages. This is sometimes a painful endeavor, made even more so if your network bandwidth is all but extraordinary, or your corporation works behind custom proxy and time is short. One way around this is to leverage Docker's shared networking and utilize a proxy container. *Squid-in-a-can*²² is one such example of a simple transparent squid proxy within a Docker container.

This services every other Docker container, including containers used during the build process while generating image layers, a local cache of any frequent http traffic. By easily changing the configuration file, you can leverage any of the more advanced squid proxy features, while avoiding the tedious install and setup of a proper squid server on various hosts' distribution.

5.3 *Docker DNS Resolver*

We've shown before how ROS nodes running from separate containers within a common software defined network can communicate utilising domain names given to containers and resolved by Docker's internal DNS. Communicating to the same containers from the host through the default bridge network is also possible, although not as straightforward without the host having similar access to the software defined network's local DNS. We can quickly circumvent this issue as we did with the proxy, by running the required service from another container within the same network. In this case we can use a simple DNS server such as *Resolvable*²³ to help the local host resolve container domain names within the virtual network.

One word of caution: one should avoid using domain names that could collide, as in the case of running two instances of the industry networking example on the same Docker engine, e.g. two sets of roscores and nodes on different project networks, say `foo` and `bar`. If we were to then include a Resolvable container into each project, the use of local domain names such as `master` or `talker` could then collide for the host, whereas explicit domain naming including the project's network post-fix such as `foo_default` would still properly resolve.

²²<https://github.com/jpetazzo/squid-in-a-can>.

²³<https://github.com/gliderlabs/resolvable>.

References

1. Arulkumaran, K. 2015. Kaixin/dockerfiles. <https://github.com/Kaixin/dockerfiles>.
2. Boettiger, C. 2015. An introduction to docker for reproducible research. *SIGOPS Operating Systems Review* 49 (1): 71–79. doi:[10.1145/2723872.2723882](https://doi.org/10.1145/2723872.2723882).
3. Bonsignorio, F., and A.P. del Pobil. 2015. Toward Replicable and Measurable Robotics Research. *IEEE Robotics & Automation Magazine* 22 (3): 32–35. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7254310>.
4. Guglielmelli, E. 2015. Research Reproducibility and Performance Evaluation for Dependable Robots. *IEEE Robotics & Automation Magazine* 22 (3): 4–4. <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7254300>.
5. Jia, Y., E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. [arXiv:1408.5093](https://arxiv.org/abs/1408.5093).
6. Mabry, R., J. Ardonne, J. Weaver, D. Lucas, and M. Bays. 2016. Maritime autonomy in a box: Building a quickly-deployable autonomy solution using the docker container environment. In *IEEE Oceans*.
7. Sucan, I.A., and S. Chitta. Moveit! <http://moveit.ros.org>.
8. White, R. 2015. ROS + Docker: Enabling repeatable, reproducible and deployable robotic software via containers. ROSCon, Hamburg Germany. <https://vimeo.com/142150815>.
9. White, R., M. Quigley, and H. Christensen. 2016. SROS: Securing ROS over the wire, in the graph, and through the kernel. In *Humanoids Workshop: Towards Humanoid Robots OS. Cancun, Mexico*.

Author Biographies

Ruffin White is a Ph.D. student in the Contextual Robotics Institute at UC San Diego, under the direction of Dr. Henrik Christensen. Having earned his Masters of Computer Science at the Institute for Robotics and Intelligent Machines, Georgia Tech, he remains an active contributor to ROS and collaborator with the Open Source Robotics Foundation. His research interests include mobile robotic mapping, with a focus on semantic understanding for SLAM and navigation, as well as advancing repeatable and reproducible research in the field of robotics by improving development tools for robotic software.

Dr. Henrik I. Christensen is a Professor of Computer Science at Dept. of Computer Science and Engineering UC San Diego. He is also the director of the Institute for Contextual Robotics. Prior to UC San Diego he was the founding director of the Institute for Robotics and Intelligent machines (IRIM) at Georgia Institute of Technology (2006–2016). Dr. Christensen does research on systems integration, human-robot interaction, mapping and robot vision. He has published more than 300 contributions across AI, robotics and vision. His research has a strong emphasis on “real problems with real solutions.” A problem needs a theoretical model, implementation, evaluation, and translation to the real world.

A ROS Package for Dynamic Bandwidth Management in Multi-robot Systems

Ricardo Emerson Julio and Guilherme Sousa Bastos

Abstract Communication is an important component in robotic systems. The application goals such as, finding a victim or teleoperate a robot in an obstacle avoiding application, may get affected if there are problems in communication between system agents. The developed package, *dynamic_bandwidth_manager* (DBM), was designed to maximize bandwidth usage in multi-robot systems. DBM controls the rate that a node publishes a topic, managing different channels where commands, sensory data and video frames are exchanged. In this tutorial chapter, we present several important concepts that are crucial to work with robot communication using ROS: (1) how the increasing number of robots makes an impact on communication, (2) the ROS communication layer (topics and services using TCP and UDP), (3) how to analyze the bandwidth consumption in a system developed in ROS, and (4) how use DBM to manage bandwidth usage. A detailed tutorial on developed package is presented. It shows how DBM is designed in order to prioritize communication channels according to environment events and how the most important topics gets more bandwidth from the system. This tutorial was developed under Ubuntu 15.04 and for ROS Jade version. All presented components are published on our ROS package repository: http://wiki.ros.org/dynamic_bandwidth_manager.

Keywords Multi-robot · Dynamic bandwidth management · Communication

1 Introduction

Multi-robot systems can be used for a set of tasks such as rescue operations [1, 2], large-scale explorations [3], and other tasks that can be subdivided between multiple robots [4]. Communication is an important component that merits careful consid-

R.E. Julio (✉) · G.S. Bastos

System Engineering and Information Technology Institute—IESTI, Federal University of Itajubá—UNIFEI, Av. BPS, 1303, Pinheirinho, Itajubá, MG CEP: 37500-903, Brazil
e-mail: ricardoej@gmail.com
URL: <http://www.unifei.edu.br>

G.S. Bastos
e-mail: sousa@unifei.edu.br

eration in a multi-robot system. The number of packets transmitted between agents of a system can increase as the number of sensors, actuators, and additional robots, increases as well [5].

A teleoperation system is a good example that illustrates data transmission between agents. In that sort of system, an user or an automated control device can control a swarm of mobile robots [6, 7] directly driving the robotic motor or sending targets for the robots. The important issue in this example is that a video streaming is transmitted to a control device while a system operator remotely controls the robots. Therefore, the number of transmitted packets is increased when the number of robots increases or when there is a video quality improvement, which may affect the system performance in a bandwidth constrained environment. For this reason, bandwidth is an important component that must be considered. A loss of packets may occur when the number of packets in transmission is greater than available bandwidth. Thus, the frequency of all sensors should be adjusted to not exceed the available bandwidth. The task of adjusting the communication rates can be challenging; in a static solution, the frequencies cannot be adjusted when there is a change in the environment or in the available bandwidth.

In such systems it may not be necessary to transmit data from sensors all the time and in same frequency. Considering the teleoperation example, if the robot is stopped or away from an obstacle, the operator does not need constant updating of the robot camera image. Thus, the frequency of video streaming can be decreased whenever the robot speed decreases or when no obstacles are close to the robot. In other words, the frequency of sensor can be decreased if there is nothing relevant to the task occurring at that time in the environment [8].

The *dynamic_bandwidth_manager* (DBM) [9] package was designed to provide a way of controlling the frequency that a topic publishes data. DBM can be applied to any topic in the system and the frequencies are calculated based on topic priorities. It helps developers to create topics with dynamic frequencies that will depend on changes in the environment, such as available bandwidth and interesting events of a task (speed, distance to obstacles and so forth).

However, how are related speed and distance to obstacles to the bandwidth? In a system developed using ROS, sensory data are sent using topics. Usually, these topics publish data with a static frequency calculated using a design parameter and it does not change if changes occur in robots environment. If a robot is stopped in a teleoperation application, may not be necessary publish its camera image for a central computer. In a restricted bandwidth system it may be prohibitive to send data unnecessarily, being other robots get affected.

A dynamic solution to the presented problem is presented in this chapter; topics that send sensor data to other elements may have their frequencies dynamically adjusted by the system. Environment events such as speed and distance to obstacles can be used to set topic priorities and define which of them may have more frequency at a given time.

As an example, we may consider a scenario with three robots in an application of identifying victims in disaster areas. Each robot moves through the area reading information from environment such as camera images. All information is sent to a

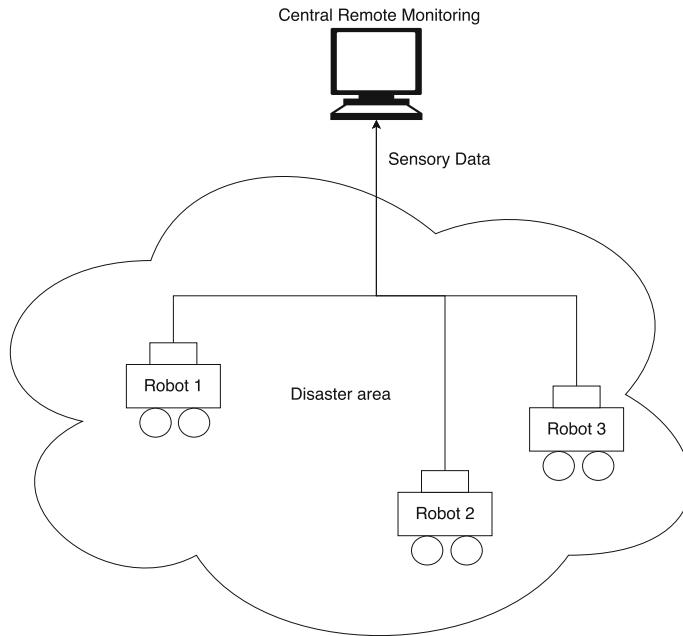


Fig. 1 Scenario in an application of identifying victims in disaster areas

remote central for monitoring where human operators assist in victim identification task using the information sent by the robots. The presented scenario is shown in Fig. 1. In this example, a desired communication rate of camera images for maximum application efficiency is 16 Hz for each robot [10]. In other words, each robot must send data read by the camera 16 times every second. This communication rate ensures that the human operator can teleoperate the robot through the disaster area avoiding obstacles and the monitoring system can predict with greater certainty the presence and location of victims in the area monitored by the camera of the robot.

In this example, the system is used in an environment with bandwidth restrictions. The network supports sending just 30 messages per second in total (considering messages of all robots). Sharing bandwidth equally, each robot sends data in a frequency of 10 Hz. This 10 Hz baud rate allows a user teleoperate a robot to identify a victim, but with a lower level of accuracy (the higher the frequency, greater the accuracy and lower the error in robot teleoperation). Thus, the user can teleoperate a robot, but he is subject to restrictions in the video sent by the robot. This degradation in the video quality may lead collision with obstacles or failure to identify a victim (Fig. 2).

As described above, bandwidth restrictions can impact the effectiveness of a solution. Set a static frequency of 10 Hz for all robots prevents communication exceeds the maximum bandwidth available, but it does not allow a robot find a victim with maximum accuracy even if other robots are far from that goal. In this case, the system could reduce the sending frequency of robots that have not yet detected any victim

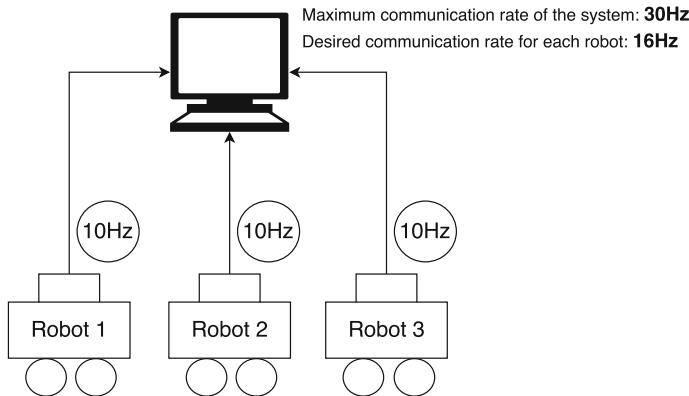


Fig. 2 Static communication rates in an application of identifying victims in disaster areas

to the minimum acceptable frequency. This allows a higher frequency for the robot that found a victim and now need to find its exact location.

A key contribution of this chapter is the development of the DBM package where some concepts about robot communication using ROS will be presented. The problem of using several topics in an environment with bandwidth constraint will be addressed and a feasible solution for managing topics in order to minimize this problem is discussed.

After a brief discussion about the motivation of this chapter, we will introduce the following topics:

- A summary of ROS publish-subscribe mechanism is provided as essential background information for the understanding of the problem;
- A review about how the increasing number of robots (or topics) impacts on communication in an environment with bandwidth constraint;
- A simple example on monitoring bandwidth consumption in a ROS-based system;
- A discussion on topics frequency control to maximize bandwidth usage or avoid loss of communication;
- All components of DBM architecture with their interactions description;
- A case study on how to use the developed package in an teleoperation application using a simulated environment;
- And, a discussion about the results.

2 ROS Topics

As discussed in [11], topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data *subscribe* to

the relevant topic; nodes that generate data *publish* to the relevant topic. There can be multiple publishers and subscribers to a topic.

Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching type. The Master does not enforce type consistency among the publishers, but subscribers will not establish message transport unless the types match. Furthermore, all ROS clients check to make sure that an MD5¹ computed from the .msg files match. This check ensures that the ROS Nodes were compiled from consistent code bases.

ROS currently supports TCP/IP-based and UDP-based message transport. The TCP/IP-based transport is known as TCPROS and streams message data over persistent TCP/IP connections. TCPROS is the default transport used in ROS, which is the only transport that client libraries are required to support. The UDP-based transport, which is known as UDPROS, is currently supported only in roscpp and separates messages into UDP packets.

ROS nodes negotiate the desired transport at runtime. For example, if a node prefers UDPROS transport but the other node does not support it, the system fallbacks on TCPROS transport. This negotiation model enables new transports to be added over time as compelling use cases arise.

Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls (i.e. receive a response to a request) should use services instead. There is also the Parameter Server [12] for maintaining small amounts of state.

The ROS Master acts as a nameservice in the ROS. It stores topics and services registration information for ROS nodes. Nodes communicate with the Master to report their registration information. As these nodes communicate with the Master, they can receive information about other registered nodes and make connections as appropriate. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are run.

It is important to make clear that nodes connect to other nodes directly; the Master only provides lookup information, much like a DNS server. Nodes that subscribe to a topic will request connections from nodes that publish that topic, and will establish that connection over an agreed upon connection protocol. In other words, when a node receives data from a topic, this communication does not pass through the ROS Master.

3 Bandwidth Consumption in Topics Publishing

Publishing topics with large messages such as camera images can cause problems in a ROS-based system. The system performance can be impaired if the amount of information transmitted over the network is larger than the available bandwidth. In

¹MD5 (Message-Digest Algorithm 5) is a cryptographic hash function producing a 128-bit (16-byte) hash value commonly used to verify data integrity.

this case, loss or delay in delivery of messages can occur, causing loss of information that can be crucial for the proper functioning of the system. But can we see how much bandwidth the topic is using? And how large messages overload the network in a ROS-based system? In this section we will show how to use the *rostopic bw* and *rostopic hz* to display the bandwidth and the publishing rate of a topic, how large messages can overload a system with bandwidth restrictions and how DBM can help avoid this problem.

3.1 Publishing Camera Images in ROS

Every time a message is published on a ROS topic and a subscriber is running on a remote machine, data are transmitted over the network. These data consume part of available bandwidth for the application and depending on the size of messages and transmission frequency, communication can exceed the available bandwidth causing delay or loss of information.

This behavior can be shown publishing camera images to other nodes in the system. Camera images are used as an example because it is simple to simulate using a laptop with a webcam and has a significant message size, but other types of message may have the same problem such as PointCloud, LaserScan, etc.

The *usb_cam_node* interfaces with standard USB cameras (e.g. the Logitech Quickcam) using *libusb_cam* and publishes images as a ROS message of type *sensor_msgs:Image* (http://docs.ros.org/api/sensor_msgs/html/msg/Image.html) using the *image_transport* (http://wiki.ros.org/image_transport) package. In this example, we will use this node to publish camera images.

The *usb_cam* can easily be installed on a Ubuntu 15.04 distribution using ROS Jade. The most updated information about *usb_cam* package can be found on the *usb_cam* wiki page (http://wiki.ros.org/usb_cam). There are some steps to installing and running *usb_cam*:

1. Install ROS (follow the latest instructions on the ROS installation page) (<http://wiki.ros.org/ROS/Installation>).
2. Download *usb_cam* package to catkin src folder (i.e. `/catkin_ws/src`):

```
$ git clone https://github.com/bosch-ros-pkg/usb_cam  
~/catkin_ws/src/usb_cam
```

3. Build the downloaded package:

```
$ cd ~/catkin_ws/  
$ catkin_make
```

4. Setup the environment:

```
$ source ~/catkin_ws/devel/setup.bash
```

5. Run *usb_cam_node*:

```
$ rosrun usb_cam usb_cam-test.launch
```

Using *image_view* node we can see camera video published by *usb_cam_node*. This may be done running the following command (note that we are subscribing in compressed² image transport mode). Access the *image_view* wiki page (http://wiki.ros.org/image_view) for more information about the package.

```
$ rosrun image_view image_view
  image:=/usb_cam/image_raw
  _image_transport:=compressed
```

3.2 Monitoring Bandwidth Usage in ROS

Monitoring the bandwidth consumed by topics is an important task in robot systems that rely on communication. The *rostopic bw* tool displays the bandwidth used by a topic and *rostopic hz* displays its publishing rate. It is important to note that, as shown in *rostopic* documentation page (<http://wiki.ros.org/rostopic>), the bandwidth reported by *rostopic bw* is the received bandwidth. If there are network connectivity issues, or *rostopic* cannot keep up with the publisher, the reported number may be lower than the actual bandwidth.

The bandwidth consumption of the compressed camera topic published by *usb_cam_node* is given by the following commands:

1. Displays the bandwidth used by camera topic (Fig. 3):

```
$ rostopic bw /usb_cam/image_raw/compressed
```

2. Displays the publishing rate of camera topic (Fig. 4):

```
$ rostopic hz /usb_cam/image_raw/compressed
```

As shown in Fig. 3 and using a camera resolution of 640×480 , the mean size of camera image message is approximately 18 KB (see field “mean” on *rostopic bw* result). The default framerate of the *usb_cam_node* is 30 FPS, i.e. the data is published using a frequency of 30 Hz (as can be seen in Fig. 4). It means that the average bandwidth consumption of the topic */usb_cam/image_raw/compressed* is approximately 570 KB/s, as also described in field “average” on *rostopic bw* result.

²The *image_transport* package provides transparent support for transporting images in low-bandwidth compressed formats such as PNG and JPEG. Thus, the image with any compression is published, for example, in a topic */usb_cam/image_raw* and the compressed image using PNG or JPEG in a topic */usb_cam/image_raw/compressed*. Follow this link http://wiki.ros.org/image_transport for more information about raw and compressed images in ROS.

```
Terminal File Edit View Search Terminal Help
ricardoej@ricardo-laptop:~$ rostopic bw /usb_cam/image_raw/compressed
subscribed to [/usb_cam/image_raw/compressed]
average: 581.97KB/s
  mean: 18.98KB min: 18.89KB max: 19.11KB window: 30
average: 570.31KB/s
  mean: 18.82KB min: 18.42KB max: 19.11KB window: 60
average: 569.36KB/s
  mean: 18.86KB min: 18.42KB max: 19.20KB window: 90
average: 564.90KB/s
  mean: 18.69KB min: 17.89KB max: 19.20KB window: 100
average: 565.86KB/s
  mean: 18.71KB min: 17.89KB max: 19.20KB window: 100
average: 565.17KB/s
  mean: 18.69KB min: 17.89KB max: 19.13KB window: 100
average: 568.94KB/s
  mean: 18.80KB min: 18.47KB max: 18.95KB window: 100
average: 567.86KB/s
  mean: 18.81KB min: 18.69KB max: 18.95KB window: 100
average: 568.30KB/s
  mean: 18.81KB min: 18.69KB max: 18.89KB window: 100
average: 561.07KB/s
  mean: 18.80KB min: 18.71KB max: 18.88KB window: 100
average: 566.69KB/s
  mean: 18.76KB min: 18.64KB max: 18.88KB window: 100
```

Fig. 3 Result of *rostopic bw* command

```
Terminal File Edit View Search Terminal Help
ricardoej@ricardo-laptop:~$ rostopic hz /usb_cam/image_raw/compressed
subscribed to [/usb_cam/image_raw/compressed]
average rate: 29.914
  min: 0.030s max: 0.040s std dev: 0.00316s window: 30
average rate: 29.912
  min: 0.010s max: 0.041s std dev: 0.00615s window: 60
average rate: 29.946
  min: 0.010s max: 0.043s std dev: 0.00564s window: 90
average rate: 29.952
  min: 0.010s max: 0.043s std dev: 0.00517s window: 120
average rate: 29.984
  min: 0.010s max: 0.043s std dev: 0.00481s window: 150
average rate: 29.980
  min: 0.000s max: 0.043s std dev: 0.00532s window: 180
average rate: 29.987
  min: 0.000s max: 0.043s std dev: 0.00508s window: 210
average rate: 29.988
  min: 0.000s max: 0.043s std dev: 0.00492s window: 240
average rate: 29.991
  min: 0.000s max: 0.043s std dev: 0.00476s window: 270
average rate: 29.977
  min: 0.000s max: 0.043s std dev: 0.00461s window: 300
average rate: 29.988
  min: 0.000s max: 0.043s std dev: 0.00451s window: 330
```

Fig. 4 Result of *rostopic hz* command

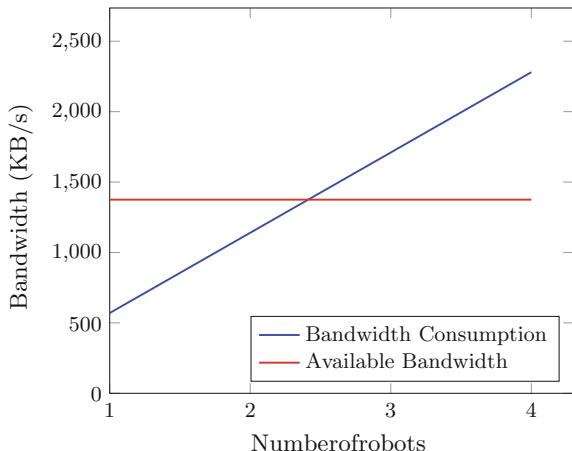
As we can see, bandwidth consumption of only one topic publishing compressed camera images with a resolution of 640×480 to a frequency of 30 Hz is 570 KB/s. If the number of topics publishing camera images in the system increases, the available bandwidth can be exceeded. Table 1 shows the bandwidth consumption in a system with four camera image topics.

If the robots communicate via a WiFi network using 802.11b WiFi standards, their corresponding maximum speeds is 11 Mbps, i.e. 1375 KB/s. If the number of

Table 1 Bandwidth consumption in a network with more camera image topics

Topics number	Bandwidth consumption (KB/s)
1	570
2	1140
3	1710
4	2280

Fig. 5 Bandwidth consumption and available bandwidth in a network with more camera image topics



robots publishing camera images increases there may be a network overhead in the system. Figure 5 shows the bandwidth consumption on a camera image topic when the number of robots or topics in the system increases. As we can see, if the number of robots is greater than 2, the use of bandwidth exceeds the available bandwidth and this can lead to communication problems.

This example shows, in a simple way, how sending information using ROS topics can overload the network. Thus, it is necessary to develop strategies to manage the publication rate of topics in order to avoid this problem.

3.3 Install DBM Package

The step-by-step instructions for installing DBM are shown below. Before you start, PuLP package must be installed before DBM package. PuLP is a library for the Python scripting language that enables users to describe mathematical programs [13]. It is used by *default_optimizer_node* to solve the linear optimisation problem described in Sect. 4.2. To install PuLP follow the instructions on the PuLP installation page (https://pythonhosted.org/PuLP/main/installing_pulp_at_home.html).

1. Download DBM package to catkin src folder (i.e. /catkin_ws/src):

```
$ git clone
  https://github.com/ricardoej/dynamic_bandwidth_
  manager ~/catkin_ws/src/dynamic_bandwidth_manager
```

2. Build the downloaded package:

```
$ cd ~/catkin_ws/
$ catkin_make
```

3. Setup the environment:

```
$ source ~/catkin_ws/devel/setup.bash
```

3.4 Using DBM to Manage Bandwidth Consumption

The *dynamic_bandwidth_manager* (DBM) [9] package was designed to provide a way of controlling the frequency that a topic publishes data. It helps developers to create topics with dynamic frequencies that will depend on the topic priority at a given time. To show how this package works, we will use the example with camera images to manage the bandwidth consumption using DBM. The detailed architecture of the package is defined in the following sections.

After install DBM, we need a set of topics that should be controlled by DBM. Figure 5 shows that 3 topics publishing compressed camera images can exceeds available bandwidth in a system using a network with a maximum speed of 11 Mpps.

Thus, in order to test DBM we will use a system with 3 topics publishing compressed camera images in different machines with a webcam (machines A, B and C) and a network with available bandwidth of 11 Mbps. The available bandwidth can be configured using parameters in DBM. So, in this example, we need not worry about the network specifications. A fourth machine (Master) must run the ROS Master and *image_view* node so we can see the published images. Figure 6 shows how the system should be designed.

There are some steps to configure the environment:

1. Run the ROS Master in machine Master:

```
$ roscore
```

2. Setup network following this link <http://wiki.ros.org/ROS/NetworkSetup>.
3. Download *usb_cam* package to machines A, B and C as described in Sect. 3.1.
4. Edit file *usb_cam-test.launch* to remap *image_raw* topic name using the machine name (A, B and C). A good explanation about names remapping can be found in <http://wiki.ros.org/roslaunch/XML/remap>. Use the name */[machine_name]/usb_cam/image_raw*.
5. Run *usb_cam_node* in machines A, B and C:

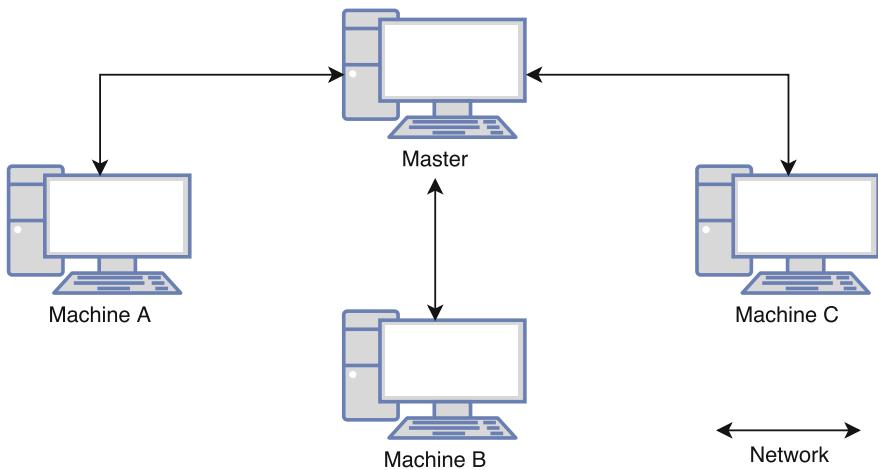


Fig. 6 System design of the DBM example

```
$ rosrun usb_cam usb_cam-test.launch
```

6. Run `image_view` in machine Master for all three topics (run each command in a different terminal):

```
$ rosrun image_view image_view
  image:=~/machineA/usb_cam/image_raw
  _image_transport:=compressed
$ rosrun image_view image_view
  image:=~/machineB/usb_cam/image_raw
  _image_transport:=compressed
$ rosrun image_view image_view
  image:=~/machineC/usb_cam/image_raw
  _image_transport:=compressed
```

You should now be able to view the images of the three cameras in each of the `image_view` running on Master machine. Follow the steps in Sect. 3.1 to check the frequency of topics and consumed bandwidth. The values should be approximately as described in Table 1.

NOTE: If you can not build an environment with 3 different machines, DBM provides a node that publishes messages with a predetermined size. It is important to note that as this message published only simulates a message, you can not view the images using `image_view`. Thus, the 3 machines with a webcam can be replaced running the following command in different terminals:

```
$ rosrun dynamic_bandwidth_manager
  fake_message_publisher_node.py
  topic_name:=~/[ machine_name ]/usb_cam/image_raw
```

```
_message_size_in_kb := 18
_max_rate := 30
```

DBM *dbm_bridge_node* subscribes in a topic that has to be managed and controls its frequency based on a topic priority defined in Parameter Server. Run one *dbm_bridge_node* for each published topic (in machines A, B and C) with the command bellow:

```
$ rosrun dynamic_bandwidth_manager dbm_bridge_node . py
    _topic_name := / [ machine_name ] / usb_cam / image_raw
    _min_frequency := 1
    _max_frequency := 30
```

Where *_topic_name* is the topic name, *_min_frequency* is the minimum frequency and *_max_frequency* is the maximum frequency at the topic will be published.

We need configure the available bandwidth in 11 Mbps as defined in our previous example. This can be done using a parameter in Parameter Server. The following sections will further explain all the parameters used in the DBM. At this point we need only run the following command:

```
$ rosparam set /dbm/max_bandwidth_in_mbit 11
```

NOTE: DBM takes into account only topics that have subscribers running on remote machines. If you are running *dbm_bridge_node* using only one machine the following command should be executed:

```
$ rosparam set /dbm/manage_local_subscribers true
```

Finally, *default_optimizer_node* solves the linear optimization problem described in Sect. 4.2 calculating a topic frequency based on topic priority. Run *default_optimizer_node* with the command:

```
$ rosrun dynamic_bandwidth_manager default_optimizer_node . py
```

Running the command *rostopic list* we can see that three other topics were created with name */[machine_name]/usb_cam/image_raw/optimized*. This optimized topic publishes the same data but with a frequency managed by DBM. Using the commands *rostopic bw* and *rostopic hz* we can see the bandwidth consumption and the frequency of each optimized topic. Table 2 shows this values (with approximate values).

As can be seen, the frequency of each topic is set to 24 Hz in order to not exceed the available bandwidth. The bandwidth consumed by all optimized topics is about

Table 2 Bandwidth consumption and frequencies using DBM

Topic	Bandwidth consumption (KB/s)	Frequency (Hz)
machineA	442	24
machineB	442	24
machineC	442	24

Table 3 Bandwidth consumption and frequencies using DBM with changes in priority

Topic	Priority	Bandwidth consumption (KB/s)	Frequency (Hz)
machineA	1.0	18	30
machineB	1.0	552	30
machineC	0.0	552	1

1330 KB/s. That is, the consumed bandwidth did not exceed the available bandwidth of 1375 KB/s.

3.5 *Changing Topic Priorities*

DBM sets the frequencies based on topic priorities. The most priority topic gets a higher frequency. The priority of a topic can be changed by setting a parameter in the Parameter Server. Run the following commands to change the priority of the topics on machine A and machine B to 1 and 0 to machine C:

```
$ rosparam set /machineA/usb_cam/image_raw/dbm/priority 1
$ rosparam set /machineB/usb_cam/image_raw/dbm/priority 1
$ rosparam set /machineC/usb_cam/image_raw/dbm/priority 0
```

Table 3 shows the priority, bandwidth consumption and the frequency of each optimized topic after changing the topic priorities.

As we can see, the frequencies of topics in machine A and machine B are set to the maximum frequency configured for the topics, 30 Hz. The priority of topic on machine C is 0, so the frequency is set to the minimum value, 1 Hz. This allows that the most priority topics gets a higher frequency while the topics with lower priorities have their frequencies adjusted to low values. A video demonstration of this example can be found in [14].

This section shows an example of how DBM controls the frequency that a topic publishes data in order to avoid the system exceeds the available bandwidth. The following sections present more detailed architecture of the package and how priorities may be based on environment events.

4 Event-Based Bandwidth Optimization

In this section we explore a strategy to optimize bandwidth consumption of ROS topics. We will begin with a definition of topic priority based on environment events. Therefore, this priority is applied to a linear optimization problem in order to define the best frequency for each topic managed by the developed package.

4.1 Event-Based Topic Priority

The topic frequencies may be dynamically controlled by the environment state and available bandwidth. This approach is built upon the assumption that the communication rate of a topic depends on how important the topic is at a given time.

In the application of identifying victims described above, the system may provide a frequency for each robot. A robot that identifies a nearby victim must send best camera images to enable the user identify the exact location of the victim and operate it while avoiding obstacles. Thus, the system decreases the frequency of other robots and increases the frequency of robots that need most at this moment. Thereby, the victim position is found more accurately and the rescue team does not waste time.

In this case, bandwidth optimization is made according to the requested requirements, considering the more important environment events to the task execution. Figure 7 shows the frequencies of each robot when the *Robot 2* finds a victim. At this time, the *Robot 1* and the *Robot 3* do not have any evidence of victims in theirs monitoring area. Therefore, they can have their frequency adjusted to lower values.

In DBM package, this behavior was implemented by assigning a priority for each topic based on environment events. Thus, the priority can be modeled as a function of environment events and represents how important a topic is for the application. These events are modeled depending on the application where the package is being used.

Using teleoperation as an example, when an operator remotely controls a set of robots based on images sent by a camera, we can define the robot speed and the distance of obstacles as environment events. Thus, if the robot speed increases and the distance of the obstacles decreases, the priority of the topic that represents the camera sensor increases.

The priority p_i of the communication channel c_i is calculated as a function of the environment events e_i that affect this communication channel, such as speed, distance to obstacles, time to collision, and so for (Eq. 1).

$$p_i = f(e_1, e_2, \dots, e_n). \quad (1)$$

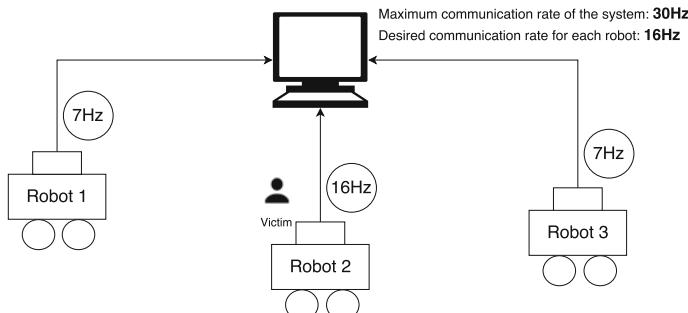


Fig. 7 Dynamic communication rates in an application of identifying victims in disaster areas

The result of that function is normalized to values in the range [0 : 1], as shown in Eq.(2), where 1 is the highest priority, which represents that the communication channel must use the higher frequency within the bounds established by the application and the available bandwidth. Thus, p'_i at a given time can be defined by [9]:

$$p'_i = \frac{p_i \cdot w_i}{\sum_{k=1}^c p_k \cdot w_k}, \quad (2)$$

where c is the communication channels and w_i the message size of the communication channels.

This normalization ensures that the message size is taken into account in the calculation of frequencies. Without this adjustment, the optimizer does not assign frequencies in proportion to the priority, generating inconsistent results with the package goal.

4.2 Bandwidth Management Based on Topic Priority

As described in previous section, each topic has a priority defined by environment events such as speed and distance to obstacles. But, how does the system calculate a frequency for managed topics based on its priority? DBM package implements a default strategy using a linear optimization problem as described in this section and in [9]. There are other works exploring this problem as in [8, 15].

The total bandwidth consumed by all managed channels is constrained by the total bandwidth available to the system, as described in Eq. (3):

$$\sum_{i=1}^n w_i \cdot f_i \leq B_{max}, \quad (3)$$

where

- w_i is the message size sent by channel i ,
- f_i is the sending frequency of the channel i ,
- B_{max} is the total bandwidth available for the system,
- n is the number of managed communication channels.

All frequencies f_i are bounded with a minimum and a maximum value $f_{i_{min}}$ and $f_{i_{max}}$. Communication channels may be maximized in order to increase the available bandwidth to the application. The priority p'_i define which channels are more important at a given time to the application and need to get more resources from the bandwidth. This is achieved by adjusting the bounds $f_{i_{min}}$ and $f_{i_{max}}$ according to the value of p'_i . If a channel has a priority $p'_i = 1$ the bounds of frequency f_i should be calculated close to the maximum ($f_{i_{max}}$). In other words, the new value of minimum frequency $f'_{i_{min}}$ is a function of p'_i . Thus, $f'_{i_{min}}$ can be defined by:

$$f'_{i_{min}} = (f_{i_{max}} - f_{i_{min}})p'_i + f_{i_{min}}. \quad (4)$$

The Eq. (4) defines a minimum value to the frequency f_i at a given time based on p_i . If the priority $p_i = 0$ the frequency is bounded with the minimum and maximum values defined by a channel. While the priority increases, the lower limit for the frequency is close to the maximum, making the system enable a greater bandwidth to the channel.

The frequencies of each managed channel will be formulated as a linear optimization problem. Thus, the problem formulation becomes:

$$\begin{aligned} & \text{maximize} && \sum_{i=1}^n w_i \cdot f_i \\ & \text{subject to} && \sum_{i=1}^n w_i \cdot f_i \leq B_{max} \\ & && f_i \geq f'_{i_{min}} \\ & && f_i \leq f_{i_{max}}. \end{aligned} \quad (5)$$

However, in some cases, there is no solution for the problem and the system informs it to the designer. Typically, in this case, the maximum bandwidth available to the system should be increased.

5 DBM Package Description

This section provides a brief explanation of DBM package. We describe the basic architecture and give an overview to the main classes and nodes. Thus, a flow chart with the basic operation of the package is presented. All DBM source code can be found in [16].

A dynamic frequency strategy for ROS topics is discussed and some examples are shown using the main classes. Finally, some issues about on how to extend the package are discussed.

5.1 Package Architecture

Package architecture is divided into four libraries: (*DBMPublisher*, *DBMSubscriber*, *DBMOptimizer*, and *DBMRate*); and into two nodes: (*default_optimizer_node*, and *dbm_bridge_node*).

- **DBMPublisher** is a class that inherits ros::Publisher, receives a minimum and a maximum frequency, and creates a managed topic;

Fig. 8 Creating a topic using *DBMPublisher*

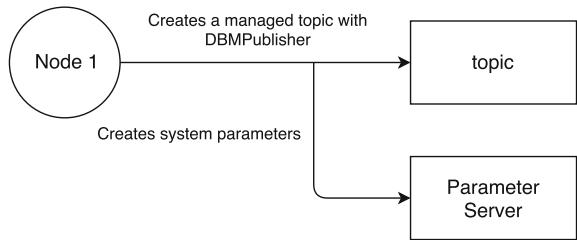
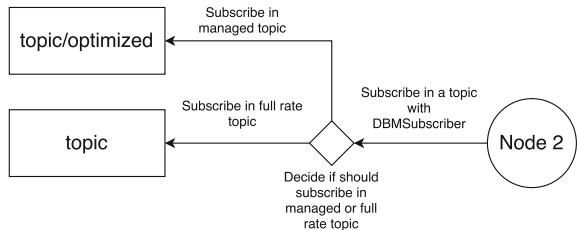


Fig. 9 Subscribing in a topic using *DBMSubscriber*



- **DBMSubscriber** is used to subscribe in a managed topic created by DBMPublisher;
- **DBMOptimizer** enables creation of optimization strategies;
- **DBMRate** helps to run loops with a dynamic rate stored in Parameter Server;
- **default_optimizer_node** solves a linear optimization problem Sect. 4.2 calculating a topic frequency based on topic priority.
- **dbm_bridge_node** allows the use of DBM into existing projects without changing their source code.

A node that publishes messages using a managed frequency creates a topic using DBMPublisher and informs the frequency limits (minimum and maximum values). All system parameters are created in Parameter Server when the package is creating a managed topic. Figure 8 shows the behavior of the library when creating a topic using DBMPublisher.

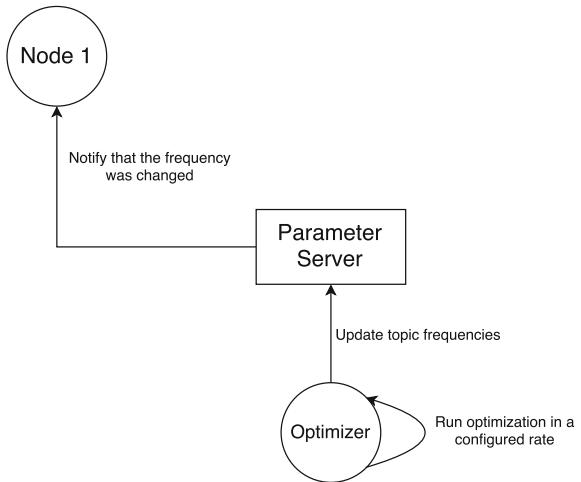
In order to perform a dynamic bandwidth management, the system should take in consideration that the message length can change and do not treat it as a static value. Whenever a message is sent by a managed channel, the *DBMPublisher* checks if the size has changed and changes the parameter for the channel. Thus, the size of messages sent through the communication channels can dynamically change.

Another node subscribes in topic using DBMSubscriber class. If the node is running on the same machine where the topic is published then DBMPublisher subscribes in a *full-rate topic*.³ Otherwise, the topic with a managed sending frequency is used (Fig. 9).

The *default_optimizer_node*, or any other node that is implementing an optimization strategy using DBMOptimizer, runs the optimization algorithm in a rate

³A *full-rate topic* is a topic with no optimization and publishes messages in a maximum frequency configured for the topic.

Fig. 10 Update of frequencies by the optimizer



configured in the parameter `/dbm/optimization_rate_in_seconds` and updates the topics frequencies in Parameter Server. Any node which is publishing a managed topic is notified and updates its communication rate (Fig. 10).

Figure 11 shows a summary of the package operation as described above.

5.2 Dynamic Frequency in a ROS Topic

In ROS, communication channels are represented by topics; through them sensor data are sent to other system elements. The code below shows the creation of a `rescue_node` that publishes a topic called `camera/image` using ROS class `ros::Rate` to control the topic frequency:

Listing 1.1 Using `ros::Rate` in `camera/image` topic

```

#!/usr/bin/env python
# license removed for brevity
import rospy
from sensor_msgs.msg import Image

def get_rescue_info():
    # Returns the camera image message

def run():
    pub = rospy.Publisher('/camera/image',
                         Image, queue_size=10)
    rospy.init_node('rescue_node', anonymous=True)
    rate = rospy.Rate(15) # Static frequency of 15hz
  
```

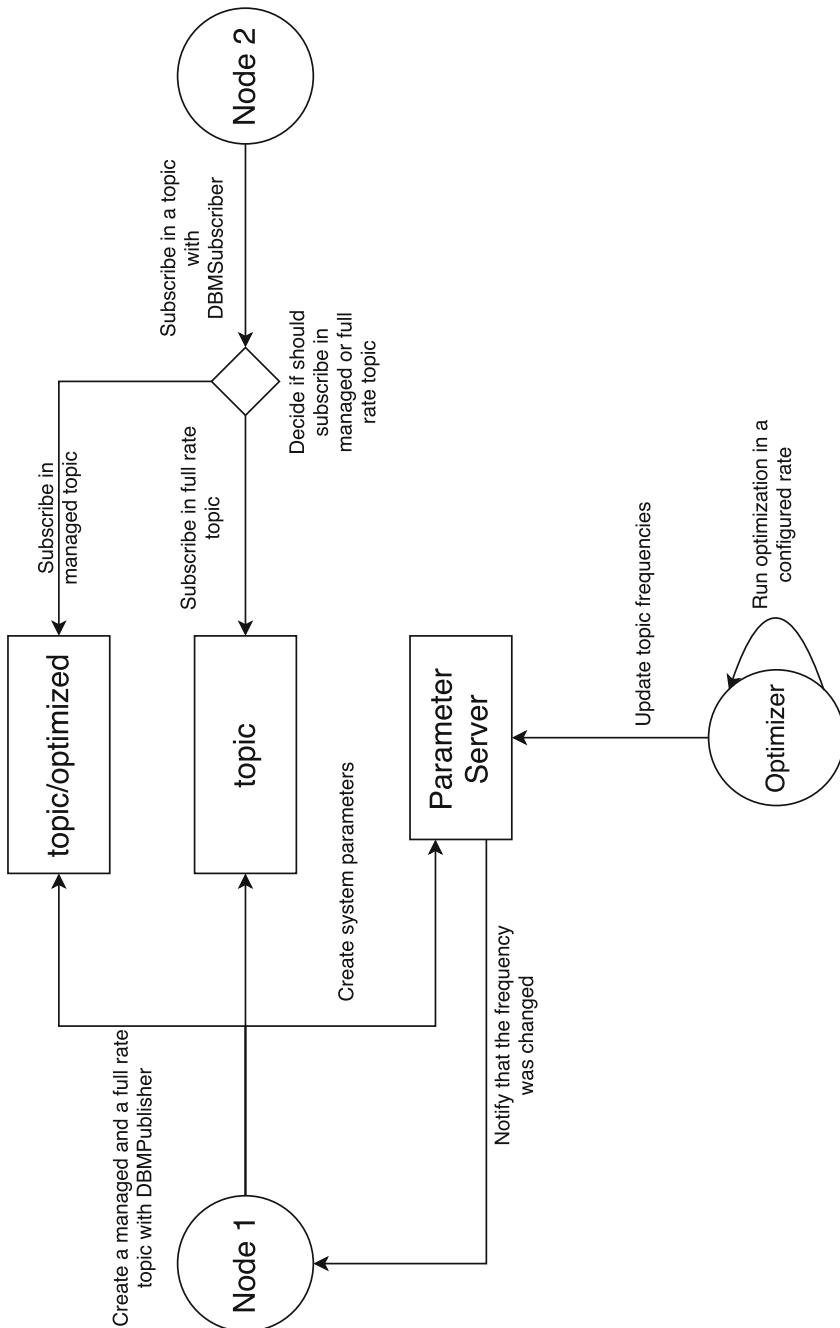


Fig. 11 Summary of the package basic operation

```
while not rospy.is_shutdown():
    message = get_rescue_info()
    pub.publish(message)
    rate.sleep()

if __name__ == '__main__':
    try:
        run()
    except rospy.ROSInterruptException:
        pass
```

In ROS topic frequencies can be controlled by `ros::Rate` class. However, this class makes a static rate control which has to be chosen in development time. Thus, an application developed with the default `ros::Rate` class does not allow a dynamic topic frequency. In other words, when using `ros::Rate`, the frequency configured for the topic is hard-coded and there is no alternative to change it in execution time.

To create a dynamic management system for topic frequencies in ROS, it is necessary to implement other strategies to control topic frequencies. DBM provides a DBMRate class which maintains a dynamic rate (stored in Parameter Server) for a loop. This class was built inheriting all features provided by `ros::Rate`. Thus, any fix or improvement implemented in base class is automatically incorporated.

The parameter name that contains the frequency value is reported during the object construction and a parameter is created in Parameter Server. Every time that `sleep()` method is invoked, the frequency value is updated and the loop delay time is adjusted. Figure 12 shows a schema with a basic operation of DBMRate class.

The main issue with this approach is the amount of calls to Parameter Server. To solve this problem, frequency values are stored using Cached Parameters, providing a local cache of the parameter. Using these versions, Parameter Server is informed that this node would like to be notified when the parameter is changed, and prevents

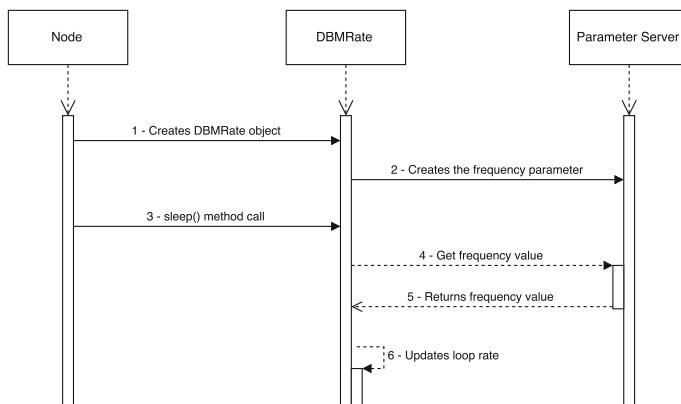


Fig. 12 DBMRate basic schema

the node from having to re-lookup the value with the parameter server on subsequent calls. Using cached parameters are a significant speed increase (after the first call), but should be used sparingly to avoid overloading the master. Cached parameters are also currently less reliable in the case of intermittent connection problems between node and master [11].

Listing 1.2 shows the node *rescue_node* created in code Listing 1.1 using DBM-Rate class. Topic frequency will be adjusted by changing a parameter stored in Parameter Server named */camera/image/dbm/frequency/current_value*.

Listing 1.2 Using DBMRate in *rescue_info* topic

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from sensor_msgs.msg import Image

def get_rescue_info():
    # Returns the camera image message

def run():
    pub = rospy.Publisher('/camera/image',
                         Image, queue_size=10)
    rospy.init_node('rescue_node', anonymous=True)

    # Creates a dynamic rate with key name
    # '/camera/image', minimum frequency of 10hz,
    # maximum frequency of 24hz and default
    # frequency of 24hz
    rate = DBMRate('/camera/image', 10, 24, 24)
    while not rospy.is_shutdown():
        message = get_rescue_info()
        pub.publish(message)
        rate.sleep()

if __name__ == '__main__':
    try:
        run()
    except rospy.ROSInterruptException:
        pass
```

5.3 Creating a New Managed Topic with DBM

In ROS, topics are created using `ros::Publisher` class. This class registers the topic in Master and provides the *publish* method responsible for messages publishing in

this topic. However, as can be seen in code Listing 1.1, the control of the topic rate is done manually in a loop.

DBMPublisher allows to publish topic messages with a dynamic frequency. This class uses DBMRate and receives a minimum and a maximum frequency, and a method returning a message to be sent. Thus, publication of messages is automatically made in accordance with the frequency parameter stored in Parameter Server.

Listing 1.3 creates the same node shown in previous code Listing 1.2, but using DBMPublisher class. The main difference is that there is no longer a need for a loop to send topic messages. When *start()* method is invoked, it receives a function returning a message to be published by the topic (*getRescueInfo* method in this example) and, internally, DBMPublisher class publishes messages in configured frequency.

Listing 1.3 Using DBMPublisher in *rescue_info* topic

```
#!/usr/bin/env python
# license removed for brevity
import rospy
import dynamic_bandwidth_manager
from sensor_msgs.msg import Image

def get_rescue_info():
    # Returns the camera image message

def run():
    # Minimum frequency of 10hz and maximum
    # frequency of 24hz
    pub = dynamic_bandwidth_manager.DBMPublisher(
        '/camera/image', Image, 10, 24)
    rospy.init_node('rescue_node', anonymous=True)

    # Starts message publishing with a frequency
    # stored in Parameter Server
    pub.start(get_rescue_info)

if __name__ == '__main__':
    try:
        run()
    except rospy.ROSInterruptException:
        pass
```

5.4 Using DBM in an Existing Package

Section 5.3 shows how to create new managed topics with DBMPublisher class. But, how do use DBM in existent packages without changes in source code? To address

this issue, DBM provides the node *dbm_bridge_node*. With this node it is possible to control topics frequencies of existent packages without changes on its source code.

DBM *dbm_bridge_node* subscribes in a full-rate topic that has to be managed and publishes the received data in a managed rate topic *[/topic_name]/optimized*. This optimized topic works on the same way that the topic created by DBMPublisher.

An explanation about how use *dbm_bridge_node* is presented in Sect. 3.4. DBM does not make any changes in the full-rate topic. The *dbm_bridge_node* only publishes the data received from the full-rate topic at a managed rate.

5.5 Implementing Other Optimization Strategies

For independence of the optimization algorithm used in the library, a module that deals with bandwidth optimization was created. DBMOptimizer is a ROS library that helps to create more complex strategies for the frequency optimization problem. This module performs the optimization algorithm at each instant as defined by */dbm/optimization_rate_in_seconds* and stores the result of the calculated frequencies in the parameter *[topic_name]/dbm/frequency/current_value*. This last parameter is used by DBMPublisher to recover the topic frequency. Thus, optimization algorithms used by DBM can be replaced without library changes. A researcher can implement new optimization strategies independently and use them to calculate the frequencies of managed topics.

This work implements *default_optimizer_node* using DBMOptimizer, which makes the frequency optimization according to Sect. 4.2. The following code shows an optimization strategy using DBMOptimizer Listing 1.4:

Listing 1.4 Optimization strategy using DBMOptimizer

```
#!/usr/bin/env python
```

```
import rospy
import dynamic_bandwidth_manager
import pulp
import numpy as np

def optimize(managed_topics):
    # Runs optimization and returns a dictionary
    # [topic_name: frequency] (the managed_topics
    # parameter has a list with all managed topics

if __name__ == '__main__':
    try:
        rospy.init_node('default_optimizer',
                        anonymous=True)
        optimizer = dynamic_bandwidth_manager
```

```

    . DBMOptimizer( optimize )
optimizer.start()

except rospy.ROSInterruptException: pass

```

In this example, a new optimization algorithm is created using the DBMOptimizer. The method *optimize(managed_topics)* implements an optimization strategy of the topics frequencies. This method receives as a parameter a list with all topic names managed by DBM and returns a dictionary [topic_name: frequency] with the calculated frequencies. This method is executed automatically by DBM and the frequencies are updated in Parameter Server.

5.6 Local Topics Management

DBM makes a topic frequency adjustment in runtime using environment events. It allows a bandwidth management and sets more bandwidth to most important topics at a moment. However, the question is: how to manage topics that send only messages to other nodes that are running on the same machine? In such cases, the topic does not have any impact on bandwidth utilization and should be ignored by the optimization algorithm.

In order to address this problem, DBMOptimizer decides which topics should be managed by the system at a given time checking that there are no external nodes communicating with the topic. If there is no external node registered in the topic, it is not treated as a managed topic and has its frequency set to maximum value.

Another important issue is when there are nodes running on different machines registered on the same topic and at least one of them is running on a machine where the topic is being published. For example, node 1 and node 2 are running on machines A and B, respectively, and they are subscribed on */camera* topic. This topic is being published by the node 3 which is also running on machine B. Figure 13 illustrates this example.

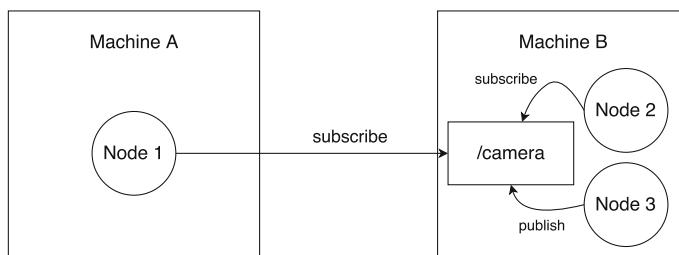


Fig. 13 Problem with local topics

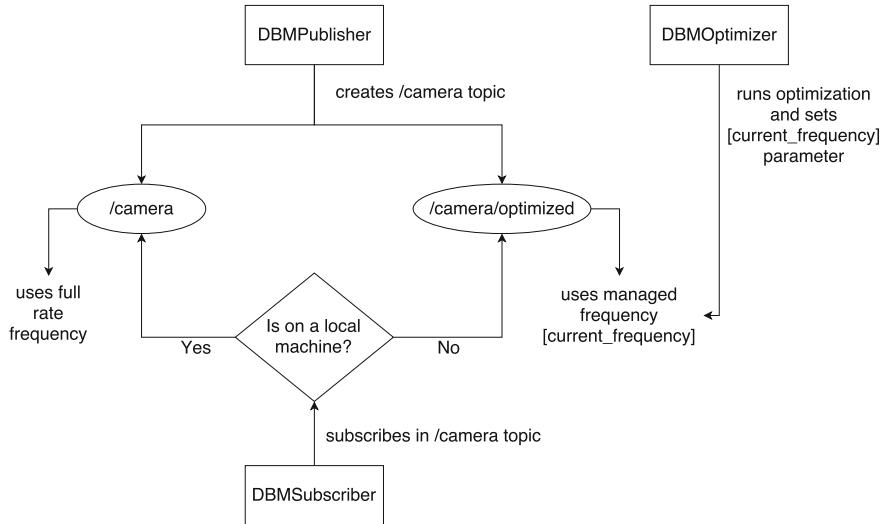


Fig. 14 Managing remote and local topics

Node 2 receives */camera* information, however it is not under bandwidth restrictions (it is running on the same machine where the topic is being published). Thus, full-rate sensor stream should still be available for local processing/logging. To address this issue, DBM creates two topics for each managed communication channel: a full-rate topic (*[topic_name]*) and an optimized rate topic (*[topic_name/optimized]*).

The decision on which topic subscribe is implemented by DBMSubscriber. If a node is running on the same machine where the topic is being published, it subscribes on the full-rate topic. In the other case, where the topic is being published from a remote machine, the node subscribes on the managed topic. Figure 14 shows a scheme illustrating the behavior described above.

5.7 System Parameters

System parameters are a set of parameters used to improve package customization allowing DBM adapt to new applications without the need to change its source code. The parameters are stored in Parameter Server and are shared between nodes. The system parameters are described below:

- */dbm/topics*: List names of all topics that have nodes running on remote machines. It is updated by *DBMOptimizer* every time that optimization algorithm runs;

- $[topic_name]/dbm/frequency/current_value$: Current $[topic_name]$ frequency;
- $[topic_name]/dbm/frequency/min$: Min frequency for $[topic_name]$ topic;
- $[topic_name]/dbm/frequency/max$: Max frequency for $[topic_name]$ topic;
- $[topic_name]/dbm/priority$: Current priority for $[topic_name]$ topic;
- $[topic_name]/dbm/message_size_in_bytes$: Message size of $[topic_name]$ topic;
- $/dbm/max_bandwidth_in_mbit$: Total bandwidth of the system;
- $/dbm/max_bandwidth_utilization$: Percentage of available bandwidth for application (values between $[0 : 100]$);
- $/dbm/optimization_rate_in_seconds$: The rate at which the optimization algorithm is executed.

In an application, there may be messages being transmitted on the network that are not managed by the DBM. Services and other unmanaged topics can be used, as well as other types of communication between system elements. Examples of unmanaged communications may be allocating tasks to the robots, commands or any other type of feature that depends on the use of the network. In such cases, the bandwidth of the system defined by the $/dbm/max_bandwidth_in_mbit$ should not be fully utilized by the managed topics and a portion of this bandwidth should be reserved for unmanaged communications. This can be done by parameter $/dbm/max_bandwidth_utilization$ ensuring that only part of the total bandwidth is used in the calculation of topic frequencies.

6 Experimental Validation

In this section, and as described in [9], we will discuss about a teleoperation application with dynamic bandwidth management using DBM. In a teleoperation application, an user or an automated control device can control a swarm of mobile robots [6, 7] directly driving the robotic motor or sending targets for the robots. In this example, an user will send targets through commands for the robot (turn left, go ahead, and so forth) while viewing the camera image on a remote computer. The main goal is obstacle avoiding. The target's message size is negligible for the application and does not impact in bandwidth utilization. Thus, to simplify the problem, commands sent by the operator to the robots will be disregarded in this example. The important issue in this example is that a video streaming is transmitted to a control device while a system operator remotely controls the robots.

The example was developed in a simulated environment running in machines with webcams. Teleoperation is a reasonable case study because it has well defined elements with a clear instance of how communication may depend on the environment. Applications of teleoperation have been useful in a lot of problems using robots [17]. Setoodeh in [18] describes a conventional teleoperation system with five distinct elements: human operator, master device, controller and communication channel, slave

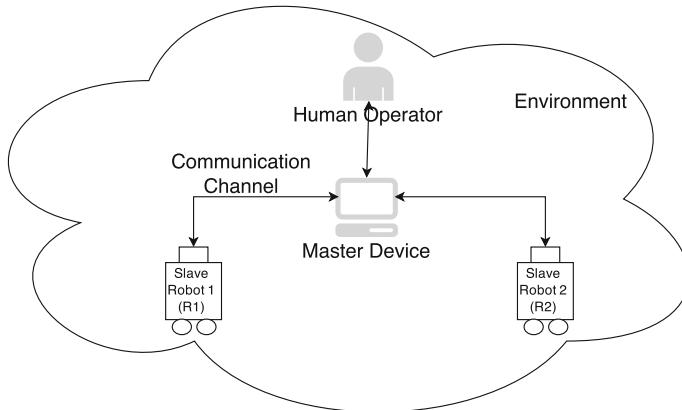


Fig. 15 Teleoperation application

robot, and the environment. The human operator uses the master device to manipulate the environment through the slave robot. Communication and controllers coordinate the operation using communication channels (Fig. 15).

In our application, two robots (R_1 and R_2) are controlled by a human operator using a workstation representing the master device. Communication channels are used to send position and other commands from master to slave and feedback visual information from slave to master. Images of robot camera are sent to the master device where the human operator is controlling the robots. The operator controls the robots manipulating their velocity and direction. In order to do it, the operator must receive visual feedback, which means sufficient information to distinguish the obstacles in the environment. Chen et al. in [10], demonstrated that people had difficulty maintaining spatial orientation in a remote environment with a reduced bandwidth. If the rate of image transmission decreases, the operator may not be able to avoid obstacles. If the rate increases to the bandwidth limit, commands sent to the robot may get lost (or arrive late) with the loss of packets in the network.

6.1 Communication Channels

The communication channel between the human operator and the robot is essential for an effective perception of the remote environment [19]. The quality of video feeds in which a teleoperator relies on for remote perception may degraded and the operators performance in distance and size estimation may get compromised with low bandwidth [20]. Chen et al. in [10], studied common forms of video degradation caused by low bandwidth, which includes reduced frame rate (frames per second or fps).

Table 4 Library system settings

<i>camera/dbm/frequency/min</i>	1 Hz
<i>camera/dbm/frequency/max</i>	16 Hz
<i>camera/dbm/message_size_in_bytes</i>	84000; 21000
<i>/dbm/max_bandwidth_in_mbit</i>	11 Mbps
<i>/dbm/max_bandwidth_utilization</i>	100%
<i>/dbm/optimization_rate_in_seconds</i>	1 s

Chen et al. [10] shows that the minimum video frame rate to avoid degraded video is 10Hz. Higher FRs such as 16Hz are suggested to some applications such as navigation. So, in this work, we will create a communication channel for image camera (topic *camera*) with 1Hz to the minimum frequency (representing cases where the operator does not need to manipulate the robot due to the stability in the environment) and 16Hz to the maximum frequency. The imaging resolution of the robot cameras is assumed to be 640×480 for R_1 and 160×120 for R_2 , which implies that each frame would be of size 84 and 21 KBytes, respectively [15]. The application uses an available bandwidth of 11 Mbps. This represents a data transfer rate of 1375 KBps (Table 4).

6.2 Environment Events

Section 4 describes the event-based priority where a priority is calculated for each channel. This priority is based on environment events that affect the importance of a channel to the system. In our teleoperation application, distance to obstacles and speed are environment events that will be monitored in order to calculate the priority for the camera's image topic. The operator needs more visual feedback when driving at a higher speed or close to obstacles.

Mansour et al. [15] define that the maximum distance detected by the sensors on the robots is 200 cm, and the maximum speed is 50 cm/s. In this simulation, we will define the same parameters in order to get real values in the simulated environment. Thus, the priority based on distance to obstacles and speed will be defined by the functions:

$$t_c = \frac{\text{distance}}{\text{speed}} \quad (6)$$

$$p_{\text{camera}} = \begin{cases} 1, & \text{if } t_c < 3 \\ 0, & \text{if } t_c > 20 \\ \frac{20-t_c}{17}, & \text{otherwise.} \end{cases} \quad (7)$$

Equation (7) defines the priority as a function of the expected time before collision as defined by (6). There are other ways to calculate the priority and the function describing environment events for the same application. They are modelled depending on where the library is being used.

6.3 Bandwidth Management

The *default_optimizer_node* runs the linear optimization problem each one second as described in parameter */dbm/optimization_rate_in_seconds*. We compare the results of the suggested algorithm with one other fixed rate algorithm. The static algorithm divides the available bandwidth among the robots in proportion to the size of the messages. Thus, with an available bandwidth of 1375 KBps (11 Mbps), R_1 gets 1100 KBps and R_2 gets 275 KBps. Respecting bandwidth limits, camera topic will send messages on a frequency of 13 Hz.

In order to evaluate the proposed bandwidth management algorithm, we present some results from the simulation using the dynamic bandwidth management library in Table 5 and Fig. 16. The system prioritizes the communication channels by increasing the frequency and providing greater bandwidth which is close to the maximum available (11 Mbpps) in all simulation times (Table 6).

Step 1 shows how the library sets a greater frequency to the most important communication channels while ensuring maximum use of the bandwidth available to the system. Robot R_1 has an estimated collision time of 20s and robot R_2 has a greater priority because its estimated collision time is 10s. The library has assigned the maximum frequency for the robot R_2 and in order to utilize the maximum of available bandwidth, has assigned 12.37 Hz for the robot R_1 .

Table 5 Frequencies in teleoperation application

Step	Robot 1 (R_1)			Robot 2 (R_2)		
	t_c (s)	p_{camera}	f (Hz)	t_c (s)	p_{camera}	f (Hz)
1	20	0.00	12.37	10	0.59	16
2	30	0.00	12.37	20	0.00	16
3	6	0.82	16	100	0.00	1.48
4	10	0.59	16	∞	0.00	1.48
5	5	0.88	16	∞	0.00	1.48
6	15	0.29	13	15	0.29	13.48
7	2.5	1.00	16	∞	0.00	1.48
8	36	0.00	12.37	10	0.59	16
9	40	0.00	12.37	7	0.76	16
10	50	0.00	12.37	3	1.00	16

Fig. 16 Priorities in teleoperation application

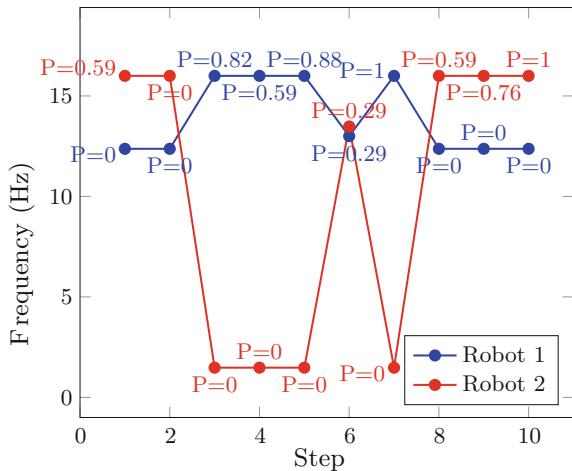


Table 6 Bandwidth used by all communication channels

Step	R_1		R_2	
	t_c (s)	Bandwidth (%)	t_c (s)	Bandwidth (%)
1	20	75.56	10	24.44
2	30	75.56	20	24.44
3	6	97.75	100	2.25
4	10	97.75	∞	2.25
5	5	97.75	∞	2.25
6	15	79.42	15	20.58
7	2.5	97.75	∞	2.25
8	36	75.56	10	24.44
9	40	75.56	7	24.44
10	50	75.56	3	24.44

Step 2 shows a simulation with $P_1 = 0$; $P_2 = 0$; $f_1 = 12.37\text{ Hz}$; $f_2 = 16\text{ Hz}$. This shows a case where the frequencies are not proportional to the priorities but the results are correct since the objective of the proposed algorithm is to maximize the bandwidth utilized by all communication channels (Eq. 5). However, considering the application scenario, a division of the frequency proportional to the priorities might be suitable. Thus, a simple modification of the proposed algorithm can divide the bandwidth among the communication channels when the priorities are equal to zero. Therefore, the frequencies in Step 2 can be recalculated to $f_1 = 13$ and $f_2 = 13\text{ Hz}$.

Steps 3, 4, 5 and 7 show cases where the R_1 is close to an obstacle and the robot R_2 is stopped. Thus, frequency of the robot R_2 can be reduced since it is not being operated and there is no risk of collision and robot R_1 can be operated with maximum visual feedback($f_1 = 16$; $f_2 = 1.48\text{ Hz}$). Step 6 shows a case where priorities are equal and the allocated frequencies are divided between the robots.

Steps 8, 9 and 10 show cases where the priorities of robot R_2 are greater than priorities of robot R_1 . In this cases, system assigned a greater frequency to robot R_2 allowing it to be operated with a higher quality of information.

The total bandwidth used for the communication channels in all simulation time is equal to the total bandwidth available to the system (11 Mbps). Raise the use of bandwidth for the maximum prevents the waste of resources without exceed the bandwidth limits.

In static algorithm, robot frequencies are 13 Hz in all simulation times. This value is greater than the minimum defined by [10] but always lower than the frequency of 16 Hz, suggested for this type of application. Best results are achieved by DBM. Only the robots with priority $P = 0$ obtained a frequency less than 13 Hz and, in most simulation times, the robot with the highest priority obtained a frequency of 16 Hz, providing a better visual feedback and helping the operator to take the decisions and avoid obstacles.

7 Conclusion

This chapter presented a dynamic bandwidth management library for multi-robots systems. The system prioritizes communication channels according to environment events and offers greater bandwidth for the most important channels. A case study on how to use the library was presented and a comparison between a static algorithm and the proposed algorithm was shown. A video demonstration of the application running can be found in our in DBM wiki page (http://wiki.ros.org/dynamic_bandwidth_manager) [21] and online at <https://youtu.be/9nRitwtnBj8>.

Some of the upcoming challenges will be to create a better real-time capability of the system. The proposed library runs the bandwidth management algorithm using a fixed rate. Thereby, rapid changes in the environment or situations in which the robots find themselves may require faster response to ensure the system is not applying bandwidth limits that are out-of-date with respect to the robots situations.

As developed in this work, the library does assume that the total bandwidth is known beforehand. This can degrade the system performance in practical settings, for instance with wireless links, whose bandwidth depends on the physical location of the nodes and the obstacles present in the environment. Experiments in real scenarios (and not simply in a simulated one, as done in this chapter) would be therefore expected to validate the approximation of considering the bandwidth known beforehand.

References

1. Casper, Jennifer, and Robin R. Murphy. 2003. Human-robot interactions during the robot-assisted urban search and rescue response at the World Trade Center. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics* 33 (3): 367–385.
2. Hiroaki, Kitano. 2000. Robocup rescue: A grand challenge for multi-agent systems. In *Proceedings of the fourth international conference on multiagent systems, 2000*, 5–12. New York: IEEE.
3. Rekleitis, Ioannis, Gregory Dudek, and Evangelos Milios. 2001. Multi-robot collaboration for robust exploration. *Annals of Mathematics and Artificial Intelligence* 31 (1–4): 7–40.
4. Lima, Pedro U., and Luis M. Custodio. 2005. Multi-robot systems. In *Innovations in robot mobility and control*, 1–64. Heidelberg: Springer.
5. Balch, Tucker, and Ronald C. Arkin. 1994. Communication in reactive multiagent robotic systems. In *Autonomous Robots 1.1*, 27–52. Dordrecht: Kluwer Academic Publishers.
6. Fong, Terrence, Charles Thorpe, and Charles Baur. 2003. Multi-robot remote driving with collaborative control. *IEEE Transactions on Industrial Electronics* 50 (4): 699–704.
7. Tsuyoshi, Suzuki, et al. 1996. Teleoperation of multiple robots through the Internet. In *5th IEEE international workshop on, robot and human communication, 1996*, 84–89. New York: IEEE.
8. Chadi, Mansour, et al. 2011. Event-based dynamic bandwidth management for teleoperation. In *2011 IEEE international conference on, robotics and biomimetics (ROBIO)*, 229–233. New York: IEEE.
9. Julio, Ricardo E., and Guilherme S. Bastos. 2015. Dynamic bandwidth management library for multi-robot systems. In *2015 IEEE/RSJ international conference on, intelligent robots and systems (IROS)*, 2585–2590. New York: IEEE.
10. Chen, Jessie YC., Ellen C. Haas, and Michael J. Barnes. 2007. Human performance issues and user interface design for teleoperated robots. In *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 37.6, 1231–1245.
11. Wiki ROS. <http://wiki.ros.org/>.
12. ROS Parameter Server. <http://wiki.ros.org/Parameter%20Server>.
13. Mitchell, Stuart, Michael OSullivan, and Iain, Dunning. 2011. PuLP: a linear programming toolkit for python. In *The University of Auckland, Auckland, New Zealand*. http://www.optimization-online.org/DB_FILE/2011/09/3178.pdf.
14. DBM Video Demonstration. <https://youtu.be/9nRitwtnBj8>.
15. Mansour, Chadi, et al. 2012. Dynamic bandwidth management for teleoperation of collaborative robots. In *2012 IEEE international conference on robotics and biomimetics (ROBIO)*, 1861–1866. New York: IEEE.
16. DBM Source Code. https://github.com/ricardoej/dynamic_bandwidth_manager.
17. Sheridan, Thomas B. 1992. *Telerobotics, automation, and human supervisory control*. Cambridge: MIT press.
18. Sorouspour, Shahin, and Peyman, Setoodeh. 2005. Multi-operator/multi-robot teleoperation: an adaptive nonlinear control approach. In: *2005 IEEE/RSJ international conference on intelligent robots and systems, 2005, (IROS 2005)*, 1576–1581. New York: IEEE.
19. French, Jon, Thomas G. Ghirardelli, and Jennifer, Swoboda. 2003. The effect of bandwidth on operator control of an unmanned ground vehicle. In *The interservice/industry training, simulation and education conference (IITSEC)*, Vol. 2003. NTSA.
20. Van Erp, Jan B.F., and Pieter Padmos. 2003. Image parameters for driving with indirect viewing systems. *Ergonomics* 46: 1471–1499.
21. DBM Wiki. http://wiki.ros.org/dynamic_bandwidth_manager.

Ricardo Emerson Julio Studied Computer Science at Federal University of Lavras (UFLA). He did his M.Sc. in Science and Computing Technology working in the System Engineering and Information Technology Institute (IESTI), Federal University of Itajuba (UNIFEI). Nowadays, he is a PhD Student in Electrical Engineering at UNIFEI. His research focuses on multi-agent systems, robotics, communication and ROS. He is an expert on software development and programming with 8 years of industrial experience working in mining area.

Guilherme Sousa Bastos Studied Electrical Engineering at Federal University of Itajuba (UNIFEI), M.Sc. in Electrical Engineering at UNIFEI, and PhD in Electronic and Computation Engineering at Aeronautics Institute of Technology (ITA), with part of doctorate done at Australian Centre for Field Robotics (ACFR). Nowadays, he is associate professor at UNIFEI and coordinator of Computer Science and Technology. He has experience in Electrical Engineering and Automation of Electrical and Industrial Processes, acting on the following subjects: electrical hydro plants, mining automation, optimization, system integration and modeling, decision making, autonomous robotics, machine learning, discrete events systems, and thermography.

Part IV

Service Robots and Fields Experimental

An Autonomous Companion UAV for the SpaceBot Cup Competition 2015

**Christopher-Eyk Hrabia, Martin Berger, Axel Hessler, Stephan Wypler,
Jan Brehmer, Simon Matern and Sahin Albayrak**

Abstract In this use case chapter, we summarize our experience during the development of an autonomous UAV for the German DLR Spacebot Cup robot competition. The autarkic UAV is designed as a companion robot for a ground robot supporting it with fast environment exploration and object localisation. On the basis of ROS Indigo we employed, extended and developed several ROS packages to build the intelligence of the UAV to let it fly autonomously and act meaningfully as an explorer to disclose the environment map and locate the target objects. Besides presenting our experiences and explaining our design decisions the chapter includes detailed descriptions of our hardware and software system as well as further references that

C.-E. Hrabia (✉) · M. Berger · A. Hessler · J. Brehmer · S. Albayrak
DAI-Labor, Technische Universität Berlin, Ernst-Reuter-Platz 7,
10587 Berlin, Germany
e-mail: christopher-eyk.hrabia@dai-labor.de
URL: <https://www.dai-labor.de/>

M. Berger
e-mail: martin.berger@dai-labor.de
URL: <https://www.dai-labor.de/>

A. Hessler
e-mail: axel.hessler@dai-labor.de
URL: <https://www.dai-labor.de/>

J. Brehmer
e-mail: jan.brehmer@dai-labor.de
URL: <https://www.dai-labor.de/>

S. Albayrak
e-mail: sahin.albayrak@dai-labor.de
URL: <https://www.dai-labor.de/>

S. Wypler · S. Matern
Technische Universität Berlin, Ernst-Reuter-Platz 7,
10587 Berlin, Germany
e-mail: s.wypler@online.de

S. Matern
e-mail: simon.matern@campus.tu-berlin.de

provide a foundation for developing own autonomous UAV resolving complex tasks using ROS. A special focus is given on the navigation with SLAM and visual odometry, object localisation, collision avoidance, exploration and high level planning and decision making. Extended and developed packages are available for download, see footnotes in the respective sections of the chapter.

Keywords Unmanned aerial vehicle · Autonomous systems · Exploration · Simultaneous localisation and mapping · Decision making · Planning · Object localisation · Collision avoidance

1 Introduction

In the German national competition SpaceBot Cup 2015 autonomous robot systems were challenged to find objects in an artificial indoor environment simulating space exploration. Two of these objects had to be collected, carried to a third object, and assembled together to build a device which has to be turned on in order to complete the task.¹

In the 2015 event the ground rover of team SEAR (Small Exploration Assistant Rover) from the Institute of Aeronautics and Astronautics of the Technische Universität Berlin [1] was supplemented by an autonomous unmanned aerial vehicle (UAV) developed by the Distributed Artificial Intelligence Lab (DAI).

The developed ground rover features a manipulator to perform all grasping tasks and is assisted by the UAV in exploration and object localisation. Hence, mapping the unknown environment and locating the objects are the main tasks of the accompanying UAV.

The aerial system can take advantage of its capabilities of being faster than a ground based system and having less issues with rough and dangerous terrain. For this reason it was the concept of providing an autonomous UAV, that rapidly explores the environment, gathering as much information as possible and providing it to the rover as a foundation for efficient path and mission planning.

The multi-rotor UAV is based on a commercial assembly kit including a low level flight controller that is extended with additional sensors and a higher-level computation platform. All intelligence and advanced software modules are used and developed within the Robot Operating System (ROS) environment.

The UAV executes its mission in a completely autonomous fashion and does not rely on remote processing at all. The system comprises the following features:

- Higher-level position control and path planning
- Monocular simultaneous localisation and mapping based on ORB-SLAM [2]
- Vision-based ground odometry based on an extended PX4Flow sensor [3]

¹Complete task description in German at <http://www.dlr.de/rd/Portaldatal/28/Resources/dokumente/rr/AufgabenbeschreibungSpaceBotCup2015.pdf>.

- Object detection using BLOB detection or convolutional neural networks [4]
- Collision avoidance with sonar sensors and potential fields algorithm [5]
- UAV attribute focused exploration module
- Behaviour-based planner for decision making and mission control

The above components have been realized by students and researchers of the DAI-Lab in teaching courses, as bachelor or master theses or part of PhD theses. Moreover, a regular development exchange has been carried out with the corresponding students and researchers of the Aeronautics and Astronautics department that worked on the ground rover on similar challenges.

In this chapter we are focusing on the UAV companion robot and present our work in a case study with detailed descriptions of our system parts, components and solutions including hardware and software. Furthermore, we describe our observations and challenges encountered during development and testing of the system, especially issues encountered with erratic components, failing hardware and components not performing as good as expected. Providing a reusable foundation for other researchers in order to support our goal of fostering further research in the domain of autonomous unmanned aerial vehicles.

We explicitly include information and references that is sometimes excluded from publications such as source code, the exact amount of autonomy and extend of remote processing (or lack of it).

The remainder of the chapter is structured as follows. Section 2 discusses related work in the context of the developed systems as well as the provided information of other authors. Sections 3 and 4 provide information about our hardware and software architecture. Here, the software architecture gives a general view on our system and should be read before going in details of following more specific sections. Section 5 explains our navigation subsystem consisting of SLAM (simultaneous localisation and mapping) and visual odometry. This section covers also the evaluation and selection process of a suitable SLAM package. The following Sect. 6 elaborates two alternative object detection and localisation approaches we have developed for the SpaceBot Cup competition. Section 7 presents background about our collision avoidance system. After, Sect. 8 goes into detail about several possible exploration strategies we have evaluated in order to determine the most suitable one for our use case. In Sect. 9 we briefly introduce a new hybrid decision making and planning package that allows for goal-driven, behaviour-based control of robots. This is followed by a short section about the collaboration and communication from our UAV with the other ground robot of our University team. After discussing the general results and limitations of our approach in Sect. 11, we summarise our chapter and highlight future tasks in the last Sect. 12.

2 Related Work

In the field of aerial robotics several platforms and software frameworks have been proposed for different purposes during the last years. In this section try to focus on platform and architecture descriptions for UAVs that have to autonomously solve similar tasks regarding exploration, object detection, mapping and localisation and touch on the essential components for autonomous flight and mission execution.

With the exception of [6], where ROS is only used as an interface, the works included here use ROS as a framework for their software implementation.

Tomic et al. [7, 8] describe a software and hardware framework for autonomous execution of urban search and rescue missions. Their descriptions are quite comprehensive giving a detailed view on their hardware and software architecture, even from different vantage points. They feature a fully autonomous UAV using a popular copter base, Pelican,² and describe specific implemented features like navigation by keyframes, topological maps and visual odometry as well as giving background and recommendations on most crucial aspects of autonomous flight such as sensor synchronisation, registration, localisation and much more. One highlight is that they employ stereo vision, speed up using an FPGA, so they do not rely solely on (2D/3D) LiDAR as many other solutions. Also it appears that all relevant design decisions are sufficiently motivated, many details needed to reproduce certain experiments are given, the mathematical background for key features is given and surprising or important results are stressed throughout. It one of the most complete works we could find, only it does not explicitly point to locations where the described modules can be acquired from, especially the custom ones, and so called mission-dependent modules (e.g. domain feature recognition) are not explained since they were presumably deemed to be out of focus.

A description of a more high-end hardware architecture is presented in [9]. The authors describe their hardware and software architecture for autonomous exploration using a sophisticated copter platform and high-end (for a mobile system) components. They give an overview of their hardware architecture, mention the most important components and give some experimental results on localisation and odometry. The software architecture is not described in depth and developed packages are not linked in the publication.

Loianno et al. [6] describe a system that is comprised of consumer grade products: A commercially available multicopter platform and a smartphone that handles all the necessary high level computation for mission planning, state estimation and navigation using the build-in sensors of the copter and the phone. The software is implemented as an application (“App”) for the Android smart phone. According to the publication ROS is only used as a transport (interface) variant for sending commands and receiving data.

During the first instance of the SpaceBot Cup in 2013, also at least two teams already deployed a UAV in attempt to aid their earthbound vehicles in their mission.

²manufactured by Ascending Technologies.

Although the systems are not described in detail and may not be fully autonomous, they are included here, since they were deployed for the same task. For 2013 The Chemnitz University Robotics Team [10] used the commercially available quadrotor platform Parrot AR Drone 2.0. Without hardware modifications this platform can only do limited on-board processing, so it was basically used as a flying camera, hovering above at a fixed position while streaming a video feed to a remote station on site that ran the mission logic and controlled the crafts position. Although the copter relies on a working communication link, it was designed to be purely optional, thus not mission critical if the link is unstable or failing. The image was also streamed back to the ground station for mission monitoring and intervention. Also another contestant, Team Spacebot 21, deployed a Parrot AR Drone during the contest in 2013 and apparently further prepared a hexacopter for the event of 2015, but no description of their systems could be found.

In summary, the level of detail for the descriptions varies greatly. This is may be due to the space constraints imposed by the publication format.

However, none of the surveyed publications did provide a full software stack in ROS that could easily be integrated and adapted for use on one's own copter platform(s). While for example [6] Loianno et al. describe a nice complete system comprised of consumer grade products, to our disappointment the authors do neither explicitly link to the advertised App (or even a demo) nor was it easily discoverable on the net.

Although there are some high quality, detailed descriptions of hardware and software platforms for autonomous UAVs available, often either some crucial implementation details are missing or the code for software modules that are described on a high level is not available.

3 Hardware Description

The main objective of our hardware concept was having a modular prototyping platform with enough payload for an advanced computation systems as well as a couple of sensors together with a reasonable flight time. Furthermore, we tried to reuse existing hardware in the lab, to limit expenses, and keep the system handleable in indoor environments. The system and its general setup is illustrated in Fig. 1.

Since our focus is not on mechanical or electrical engineering, we based our efforts on a commercially available hexacopter self-assembly kit from MikroKopter (MK Hexa XL³). The modular design simplified required extensions. We removed the battery cage and added 4 levels below the centre platform of the kit, separated with brass threaded hex spacers. It was required to replace the original battery cage, made from rubber spacers and thin carbon fibre plates, to have a more solid and robust base for the additional payload below. The level platforms were made from

³<http://wiki.mikrokopter.de/en/HexaKopter>.

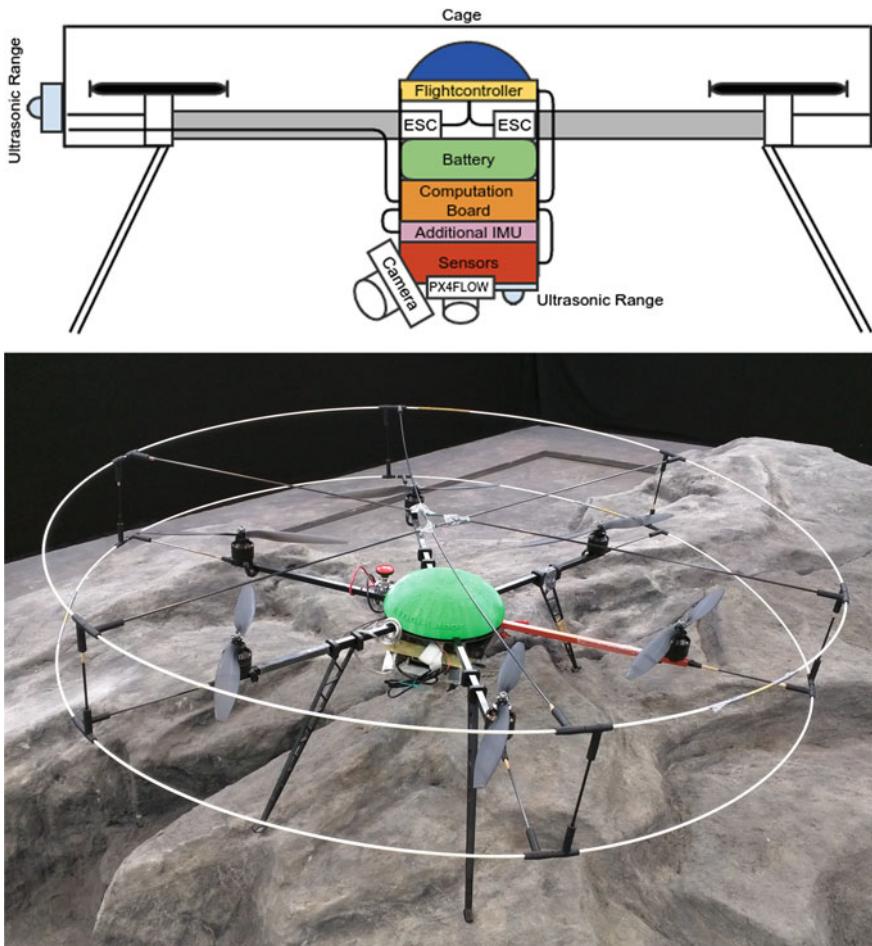


Fig. 1 The UAV hardware, setup and live in the competition environment

fiberglass for simple plain levels or 3d-printed for special mounts. In case of the computation level the platform is directly shaped by the board itself.

The first layer below the centre is still the battery mount, the second layer holds the computation platform, the third layer holds an additional IMU-sensor (Sparkfun Razor IMU 9DOF) and the lowest layer contains all optical sensors.

The original flight controller (Flight-Ctrl V2.5) is used for basic low level control of balancing the UAV and managing the motor speed controllers (ESC). The flight control is remotely controlled by a more powerful computation device. Since our research focus is not on low-level control, it is also suitable to use other available flight controllers like the 3DR Pixhawk. The only requirement is support for external control of pitch, roll, yaw and thrust, respectively relative horizontal and vertical motion, through an API.

During our research for a small scale and powerful computation board we came across the Intel NUC platform, which is actually made for consumer desktops or media centres. The NUC mainboards are small (less than 10 cm in square), comparable light-weight, provide decent computation power together with plenty of extension ports. Furthermore, they can be directly powered by a 4s-LiPo-Battery as they support power supplies providing 12–19 V. The most powerful NUC version available at that time was the D54250WYB, which provides a dual-core CPU (Intel Core i5-4250U) with up to 2,6 GHz in turbo mode. The current available NUC generation has even better performance and also includes a version with an Intel Core i7 CPU. We equipped our NUC board with 16 GB of RAM, 60 GB mSATA SSD and a mini-PCI-E Intel Dual Band Wireless-AC 7260 card.

Due to the required autonomous navigation capabilities, not relying on any external tracking system, the UAV has two visual sensors for autonomous navigation on the lowest layer. The first sensor provides input for the SLAM component and is a monocular industrial grade camera with global shutter and high frame rate (Matrix Vision mvBlueFox GC200w) with a 2 mm fisheye lens (Lensagon BF2M2020). The camera is attached to a tilttable mount allowing for a static 45° camera angle. The second sensor is looking to the ground and is a Pixhawk PX4FLOW module that provides optical odometry measurements [3]. The PX4FLOW sensor is accompanied with an additional external ultrasonic range sensor (Devantech SRF08) pointing towards the ground. The additional range sensor and IMU are required to compensate the weak performance of the PX4Flow, further details are given in Sect. 5.4.

The competition scenario should not have many obstacles in our intended flight altitude of ~2 m, actually almost exclusively pillars to support the roof structure, thus we decided to use three lightweight ultrasonic range sensors (Devantech SRF02) for collision avoidance. The sensors are attached to the protection cage of the UAV on the extension axis of the three forward facing arms. All ultrasonic range sensors are connected to the NUC USB bus with a Devantech USB I2C adapter. More demanding scenarios may require the use of more such sensors or it may be necessary replace or augment them with a 2D laser range scanner, to get more detailed information about surrounding obstacles. The protection cage of the UAV is build from kite rubber connectors, fiberglass rods for the circular structure and carbon fiber rods for the inner structure.

Our basic system without a battery and the protection cage weights 2030g. The protection cage adds 280 g and our 6600 mAh 4s LiPo battery 710 g, resulting in 3020g all in all, while providing approximately 15 min autonomous flight time.

4 Software Architecture

The abstracted major components of our system and the directed information flow is visualised in Fig. 2. The shown abstract components are consisting of several sub modules. The UAV is perceiving its environment through several sensor components that are handling the low-level communication with external hardware modules

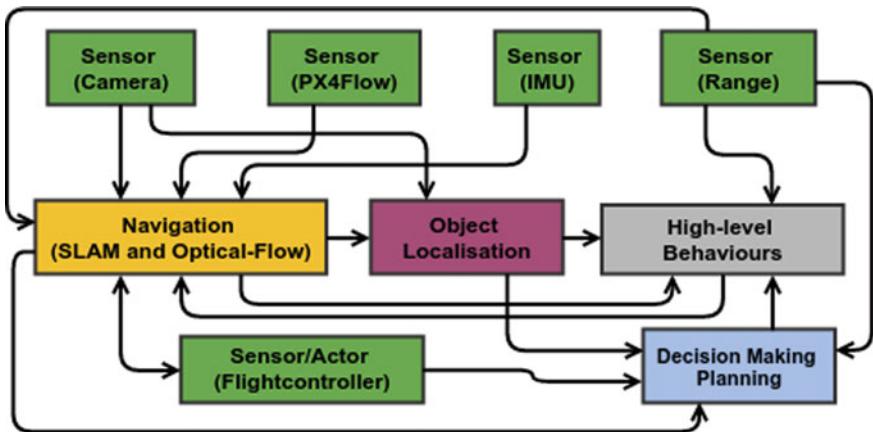


Fig. 2 The abstract UAV software architecture

and taking care of general post-processing. Most of the sensor data and its processing is related to the autonomous localisation and navigation. For this reason the navigation component is fusing all the available data after it was further processed in two distinct sub-systems for SLAM and optical-flow based odometry. The resulting map and location information as well as some of the sensor data is also used in the object localisation, high-level behaviour and decision making/planning components.

The object localisation is trying to recognize and locate the target objects in the competition environment. The decision making/planning component is selecting the current running high-level behaviour based on all available information from the navigation, object localisation, the flight controller and the range sensors, as well as from the constraints of the behaviours themselves. Depending on the executed behaviour, like collision avoidance, exploration or emergency landing the navigation component is instructed with new target positions. Hence, the navigation component is monitoring the current state and controlling the low-level flight controller with new target values for pitch, roll, yaw and thrust in order to reach the desired position. The actual motor control runs on the flight controller, while the motor speed is controlled by external speed controllers.

The abstract architecture is further detailed in the ROS architecture, see Fig. 3, showing the components including the relevant packages, mainly used topics, services and actions.

This architecture shows a common ROS approach of providing continuously generated information (e.g. sensor data) as topics, direct commands and requests (e.g. setting new targets) as services and long running requests (e.g. path following) as actions, using the ROS *actionlib*. All shown components and packages correspond to one ROS node. If nothing else is stated we used the ROS modules of version Indigo. Most information related to the core challenges of autonomous navigation and object localisation is exchanged and maintained using the *tf* package.

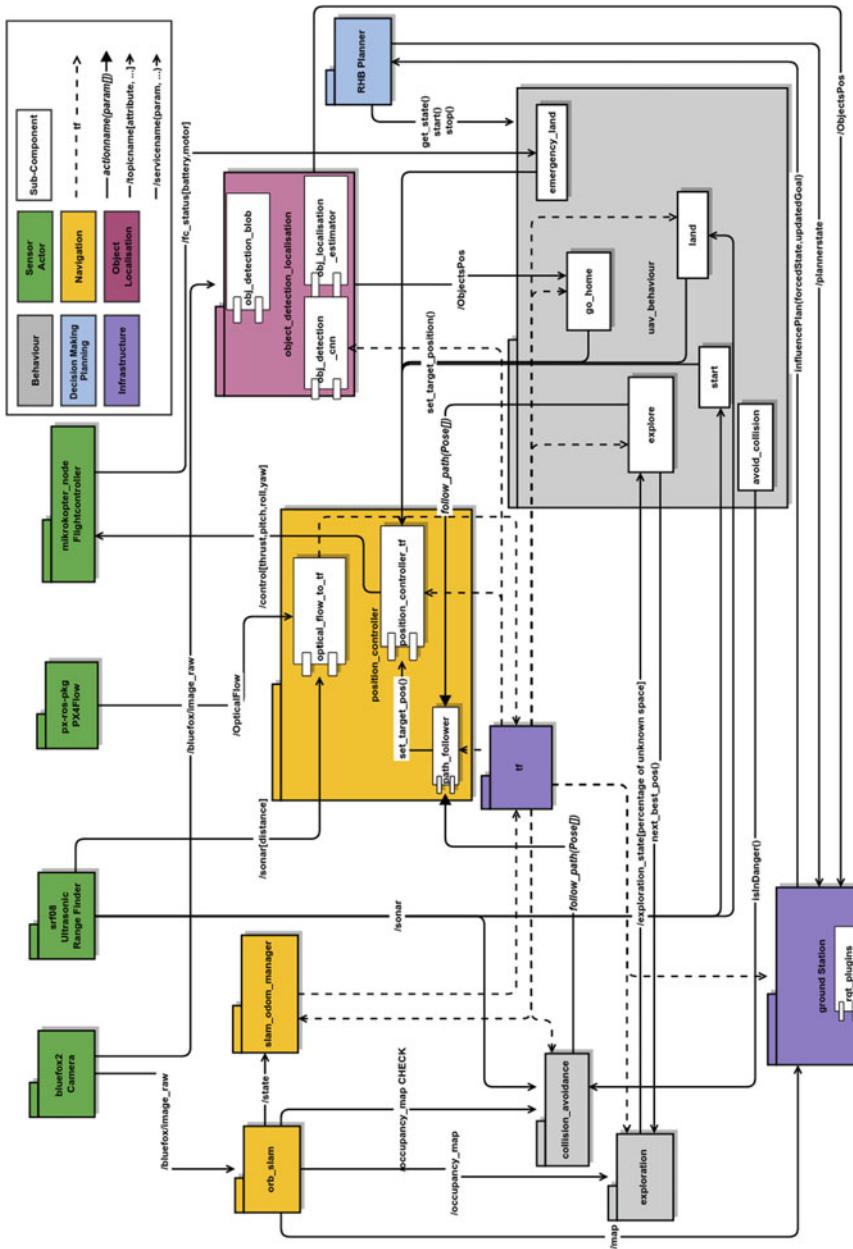


Fig. 3 The UAS ROS software architecture. Visualized are packages, components and service, action and topic interfaces. Each package or subcomponent corresponds to a node instance in the running system, except the behaviour subcomponents inside the *uav_behaviour* package

All higher-level behaviour is controlled by our decision making and planning framework RHB Planner (ROS Hybrid Behaviour Planner). The individual behaviours for start, land, emergency-landing, collision avoidance and exploration are using provided base classes of this framework and are running on one node. The RHB Planner is also supporting a distributed node architecture for the behaviours, but we did not take advantage of it because of the computationally simple nature of most behaviours. Nevertheless, the generalised implementations of the more complex tasks of collision avoidance and exploration are separated in own packages with corresponding nodes.

The external monitoring of the UAV is realised with rqt and its common plugins for visualisation and interaction as well as some custom plugins for controlling the *position_controller* and decision making and planning component. The provided controls are just enabling external intervention by the human but are strictly optional.

For accessing the monocular camera we are using a ROS package from the GRASP Laboratory.⁴

Independent of the ROS software stack running on x86 main computing platform is the software of the MikroKopter FlightCtrl and the PX4Flow module. Both embedded systems are interfaced through RS232 usb converters and their ROS wrappers in the packages *mikrokopter* and *px-ros-pkg*. Sensor data is pushed by the external systems after an initial request and collected by the ROS wrappers.

The MikroKopter FlightCtrl firmware already contains features for providing sensor information through the serial interface as well as receiving pitch, roll, yaw and thrust commands amongst other external control commands. We extended the existing firmware in several ways in order enable compilation in Linux environment, sensor debug stream without time limited subscription, added a direct setter for thrust and new remote commands for arming/disarming, calibration and beeping. Our fork of Version V2.00a and V2.12a is available online.⁵

Due to several problems with the PX4Flow sensor, see Sect. 5.1 for more details, we forked the firmware⁶ as well as the corresponding ROS package.⁷ We changed the PX4Flow firmware in order to get more raw data from the optical flow calculation, disable the sonar and process additional MAVLink messages provided by the PX4Flow. Furthermore, the firmware fork also includes our adjusted settings as well as an alternative sonar filtering.

For the communication with the ultrasonic range sensors of type SRF02 and SRF08 and the interaction with the MikroKopter flight controller we developed new ROS packages.⁸

⁴<https://github.com/KumarRobotics/bluefox2.git> and
https://github.com/KumarRobotics/camera_base.git.

⁵<https://github.com/cehberlin/MikroKopterFlightController>.

⁶<https://github.com/cehberlin/Flow>.

⁷<https://github.com/cehberlin/px-ros-pkg>.

⁸https://github.com/DAInamite/srf_serial and https://github.com/DAInamite/mikrokopter_node.

More details of the developed or extended modules and related ROS packages for navigation, autonomous behaviour (including decision making and planning) and object detection are given in the following sections.

5 Navigation

Autonomous navigation in unknown unstructured environment without any external localisation systems like GPS is one of the most challenging problems for UAV, especially if only onboard resources are available to the flying system.

Our approach is combining two methods for the localisation, we use the PX4Flow sensor module as a vision based odometry and a monocular SLAM for additional localisation information as well as creating a map. The advantage of this approach is that the odometry information, calculated from the downwards looking camera's optical flow, fused with the data from a gyroscope and an ultrasonic range sensor for scaling, is available on a higher frame rate and without initialisation period, but is prone to drifts in long-term. Whereas the SLAM provides more accurate information, but needs time for initialisation and can lose tracking. Hence, the odometry provides a backup system in case SLAM loses tracking. Moreover, using an external device (PX4Flow) for the odometry image processing has the advantage of reducing the computational load on the main computation system. An alternative solution could directly use one or more additional cameras for optical flow and odometry estimation without a special sensor as the PX4Flow. This could be done for instance by applying libviso2 or fovis through available ROS wrappers *viso2*⁹ and *fovis_ros*.¹⁰ However, such an approach would generate more processing load on the main system.

In our configuration with the PX4Flow and SLAM we achieved ~ 40 Hz update rate for the integrated odometry localisation and ~ 30 Hz for the SLAM localisation, while the map is updated with ~ 10 Hz.

Next we describe the required adjustments of the visual odometry sensor PX4FLOW in order to get suitable velocity data, continued by a section about the used SLAM library and our extensions, after explaining how we fused the localisation information from both methods and finally explaining how we have controlled the position of the UAV.

5.1 Odometry

Different than expected the PX4FLOW did not provide the promised performance and had to be modified in several ways to generate reasonable odometry estimates in

⁹<http://wiki.ros.org/viso2>.

¹⁰http://wiki.ros.org/fovis_ros.

our test environment. The implemented modifications are explained in the following.

In fact we received very unreliable and imprecise odometry measurements during our empirical tests on the UAV. For improving the performance we first replaced the original 16 mm (tele-) lens with a 6 mm (normal-) lens to get a better optical flow performance close to the ground. This is especially a problem during start and landing, as well as supporting faster movements. Second, we disabled the included ultrasonic range sensor and added another external ultrasonic range sensor (Devantech SRF08) with a wider beam and more robust measures, because of heavy noise and unexpected peak errors with the original one. In consequence, we modified the PX4FLOW firmware (see Sect. 4 for references) to provide all required information to calculate metric velocities externally. It would have also been possible to integrate the alternative sensor in the PX4Flow board itself, but due to time constraints and better control of the whole process, we decided to move this computation together with an alternative filtering (lowpass and median filter) of the range sensor, as well as the fusion with an alternative IMU, to the main computation board. This additional IMU board (Sparkfun Razor IMU 9DOF) was required because the PX4Flow did not provide a valid absolute orientation after integration, which is caused by the very simple filtering mechanisms of the only included gyroscope. Here, it was not possible to replace it with the sensors of MikroKopter flight controller, as the board does not include a magnetometer and would not have been able to provide accurate orientation data, too. Instead the additional IMU board enabled us to use 3D accelerometer, gyroscope and magnetometer for orientation estimation.

The resulting odometry information, that is velocity-estimates for all six dimensions (x, y, z, pitch, roll, yaw) and the distance to ground, is integrated over time into a relative position and published as transforms (tf) from the *optical_flow_to_tf* node of the *position_controller* package, see also Fig. 3 in Sect. 4. The coordinate frame origin is given by the starting position of the system.

5.2 Localisation and Mapping

One of the most challenging problems of developing an autonomous UAV that does not rely on any external tracking system or computation resources is finding an appropriate SLAM algorithm that can be executed on the very limited onboard computation resources together with all other system components. Since our focus is not on further advancing the state of the art in SLAM research we evaluated existing ROS solutions in order to find a suitable one that we can use as a foundation for our own extensions. In doing so we especially focus on a fast and reliable initialisation, a robust localisation with good failure recovery (recovery after lost tracking) in dynamic environment with changing light conditions and sparse features. All that is favoured over the mapping capabilities. This is motivated by our approach of enabling autonomous navigation in 6D (translation and orientation) in the first place, while in the second place the created map is considered. The created map has a minor priority since prior knowledge of the competition area would also allow for simple

exploration based on the area size, starting point and safety offsets. We evaluated the different algorithms with recorded sensor data of simulations as well as real experiments where we manually estimated the ground truth. All experiments have been executed with best knowledge from the provided documentation and calibrated sensors.

In comparison to many existing solutions, see also Sect. 2, we could not rely on a 2D laser scanner as an input sensor, because of the unstructured competition environment without surrounding walls near by. Using a 3D laser scanner was not possible due to financial limitations in our project. Furthermore, we determined that laser scanners have problems in detecting the black molleton fabric that was used to limit the competition area. In consequence it would only be possible to detect the border obstacles from max. 1m distance, independent of the actual maximum range of the sensor.

Hence, our initial idea was using a ASUS Xtion RGBD sensor together with the RGBDSLAM V2 [11] instead of a laser scanner or RGB camera. This was based on the selection and positive experience of the rover sub-team in the first execution of the competition in 2013. In this context we have also evaluated the alternative package *rtabmap* SLAM [12]. Both packages allow configuring different feature detection and feature matching algorithms. In comparison to RGBDSLAM V2, RTAB-MAP supports several sub maps that are created on a new initialisation after lost tracking. Such sub maps allow for a stepwise recovery and are fused by the algorithm later on. RGBDSLAM V2 would require getting back to the latest valid position and orientation for recovery. This is a clear disadvantage especially for an always moving aerial system. However, in our empirical test the pure localisation and mapping performance of both algorithms was similar after tuning the configuration appropriately.

Unfortunately we figured out that the required OpenNi 2 driver together with the corresponding ROS package for accessing the raw data of the RGBD sensor is generating very high load on our system. In fact, just receiving the raw unprocessed RGBD data from the sensor in ROS was giving 47.5% load on our two core system, in comparison the later selected RGB camera is just creating 7% CPU load. We did some experiments with different configurations and disabled preprocessing, but have not been able to reduce the load significantly. Due to the high load it was not possible to run any of the existing RGBD-based SLAM solutions with an appropriate frame rate and without quickly loosing tracking during motion. Though an RGBD SLAM solution would have been suitable for our indoor scenario, it is limiting the portability of the system to other applications in outdoor environment, since RGBD sensors are strongly affected by sunlight.

In consequence we tested other available SLAM solutions that are able to operate with RGB cameras as alternative sensor. The RGB sensor has also the advantage of having a higher detection range. Here, we especially looked upon monocular approaches since we expected less load, if only one image per frame has to be processed. Table 1 summarises general properties and our experiences with different algorithms. The direct gradient-based method used in LSD SLAM [13] has the advantage of generating more denser maps, while the indirect ORB SLAM [2]

Table 1 Comparison matrix of monocular SLAM packages. The numbers indicate the ranked position as qualitative comparison from 1–3, 1 being better

Package/ algorithm	Type	Map type	Initialisation	Recovery	Position quality	Orientation quality	System load	Map quality
ORB SLAM [2]	Indirect	Feature point cloud	1	1	1	1	2	2
LSD SLAM [13]	Direct	Semi-dense depth	2	2	3	3	3	1
SVO SLAM [14]	Semi-direct	Feature point cloud	3	3	2	2	1	3

and the semi-direct (feature-based combined with visual odometry) SVO SLAM [14] only provide very coarse point cloud maps build from detected features. However, both LSD SLAM and SVO SLAM had difficulties in getting a valid initialisation and have frequently lost tracking, resulting in bad position and orientation estimations. In contrast ORB SLAM is able to quickly initialise, small movements in hovering position are enough, and holds the tracking robustly, while recovering very fast once it is lost in situations without many features or very fast movements. Therefore, we selected the *ORB_SLAM* package [2] that was providing stable and fast localisation on our system, resulting in update rates of ~ 30 Hz for the SLAM localisation and ~ 10 Hz for the mapping. We empirically determined the following ORB SLAM configuration that differs from the provided default: 1000 features per image, 1.2 scale factor between levels in the scale pyramid, fast threshold of 10, enabling the Harris score and the motion model. Especially switching from the FAST score to the Harris score improved the performance in environments with sparse features and monotonic textures.

Even though the package provided a good foundation, it was missing several features. For this reason we developed some extensions, which are available online in a forked repository.¹¹ The original package is available, too.¹² Here, we incorporated an additional topic for state information, disabled the processing of topics if they are not subscribed (especially useful for several debug topics), improved the general memory management (the original includes several memory leaks) and integrated a module for extended map generation. The last extension allows for exporting octree-based maps and occupancy maps through topics and services. The octree representation is calculated from the feature point cloud of the internal ORB SLAM map representation. The occupancy map is created from the octree-based map by projecting all voxels in a height range (slice through the map) into a plane. In our case it was sufficient to use static limits to eliminate all points from ceiling and floor.

However, this extension has potential for many optimisations and extensions, for instance all calculation could benefit from caching and reusing former created maps instead of recalculating entire maps or a more sophisticated solution for detecting floor and ceiling would simplify a reuse in other scenarios. Additionally, an extension allowing to create new maps on lost tracking and fuse them later, like supported

¹¹https://github.com/cehberlin/ORB_SLAM.

¹²https://github.com/raulmur/ORB_SLAM.

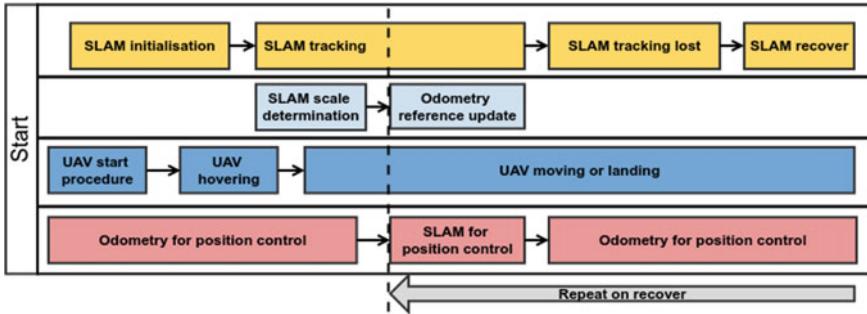


Fig. 4 The SLAM and odometry execution process flow

by RTAB-MAP, would increase the robustness and applicability of the algorithm. Furthermore, we plan to improve the new version 2 of ORB SLAM.¹³ The new version also supports stereo camera setups that may be manageable from the load perspective on the current NUC generation.

5.3 SLAM and Odometry

The used SLAM module and the position calculated from the integrated odometry data provide two distinct coordinate frames. The *slam_odom_manager* is listening to created tf-transformations of both navigation submodules and creates a fused transformation from them. This also requires valid static transformations from the sensors to the base frame of the robot. In this context, the *slam_odom_manager* is also monitoring the state of the modules in order to react to a changed SLAM state, e.g. successful initialisation, lost tracking or when tracking is recovered, by recording transformations between the distinct coordinate frames and switching or adjusting the currently used master coordinate frame that forms the base for the resulting output transformation of the *slam_odom_manager*. This transformation is the reference frame for the *position_controller*.

Due to the fact that we are only using a monocular SLAM the resulting position and map are not scaled in real world units. The problem can be addressed by determining the scale based on available metrical sensor data, as shown by Engel et al. [15]. In our solution the challenge is addressed by using the odometry data, which is scaled based on the absolute ground distance from the ultrasonic range sensor, as a reference in an initial stage of the mission in order to get a suitable conversion from the SLAM coordinate frame to the real world.

The execution flow of the process of the *slam_odom_manager* is illustrated in Fig. 4. Here, the shown parallel process execution is repeated from the dotted line

¹³https://github.com/raulmur/ORB_SLAM2.

once SLAM has recovered. The general idea is to get robust short-term localisation from odometry, while getting long-term localisation from SLAM. This is based on the experimentally verified assumption that the visual odometry of the PX4Flow has an increasing error over time due to repeated integration, but no initialisation stage. Whereas the SLAM is considered as less robust in short-term, since it can loose tracking and need a initialisation stage with a moved camera, but it is more robust in long-term, because it is able to utilize loop-closures. In order to avoid jumps in the position control of the UAV after the transition from SLAM localisation to odometry localisation and vice versa the position controller PID controllers are reinitialised at the handover.

5.4 Position Controller

The *position_controller* package is responsible for the high-level flightcontrol of the UAV. The module is available online.¹⁴ It finally generates pitch, roll, yaw and thrust commands for the flight controller based on the given input positions or path. The package is separated into three submodules or nodes. Already mentioned was the *optical_flow_to_tf* module that converts odometry information into tf-transformations. The *path_follower* is a meta-controller of the *position_controller_tf* that controls the execution of flight paths containing a sequence of target positions. The core module that calculates the flight control commands is the *position_controller_tf*. Therefore, it monitors the velocity, the distance to ground and the x-y-position of the system based on received tf-transformations. The control of the targeted positions in space is realised with a chain of two PID-controllers for acceleration and velocity for each of the four controllable parameters. Since the balance of the vehicle is maintained by the low-level flight controller, the controller does only influence roll and pitch in order to move in x-y-directions, while yaw is controlled in order to hold a desired heading. All controllers can be configured with several constraints for defining maximum change rates and velocities. The PID implementation and configuration make use of the *control_toolbox* package, but we used an own fork of the official code base¹⁵ to integrate some bugfixes as well as some own extensions that allow for an easier configuration using ROS services.

Furthermore, the controller includes a special landing routine for a soft and smooth landing, which is activated if the target distance to ground is set to 0. This routine calculates a target velocity based on the estimated exponential decreasing time-to-contact as presented in [16].

¹⁴https://github.com/DAInamite/uav_position_controller.

¹⁵https://github.com/cehberlin/control_toolbox.

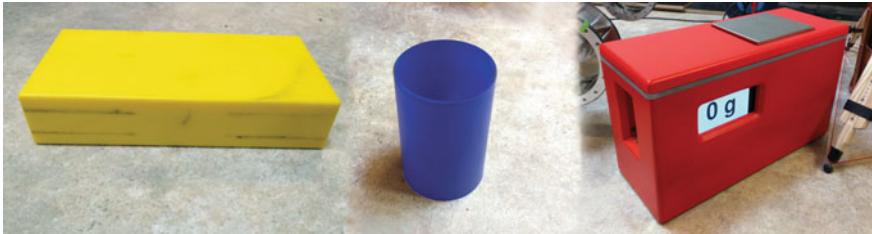


Fig. 5 The target objects. From *left* to *right*: battery pack, plastic cup and base object

6 Object Detection and Localisation

Besides autonomous flight and navigation in unknown terrain, another important key capability of the UAV is the detection of the mission's target objects. Providing additional knowledge of the terrain and the objects' positions to the ground vehicle enables quick and efficient path planning. This results in a faster completion of the search, carry and assembly tasks. In the scenario at hand three target objects had to be found. The objects are colour coded and their exact shape and dimensions are known: A yellow battery pack, a (slightly transparent) blue plastic cup and the red base object, see illustration in Fig. 5. Colours, dimensions and weights for these objects were known beforehand.

Two different approaches have been developed and evaluated to detect these objects: A simple blob detection and a convolutional neural network based detection and localisation. Besides that, other object recognition frameworks were evaluated regarding their applicability to the task: Tabletop Object Recognition,¹⁶ LINE-MOD¹⁷ and Textured Object Detection.¹⁸ Albeit, neither detection rates nor computational complexity allowed for their use on the UAV in the given scenario. The detection rates were generally not sufficient and tend to fail in case that vital constraints are violated (e.g. no flat surface can be detected, or the vertical orientation of trained objects is limited).

The lessons learned and two developed approaches are detailed in the following subsections. The discussed implementations are available online.¹⁹

6.1 Blob Detection

Motivated by the colour coded mission objects, a simple blob detection approach seemed admissible. Hence, we used a simple thresholding for the primary colours in

¹⁶<http://wg-perception.github.io/tabletop/index.html#tabletop>.

¹⁷<http://wg-perception.github.io/linemod/index.html#line-mod>.

¹⁸<http://wg-perception.github.io/tod/index.html#tod>.

¹⁹https://github.com/DAInamite/uav_object_localisation.

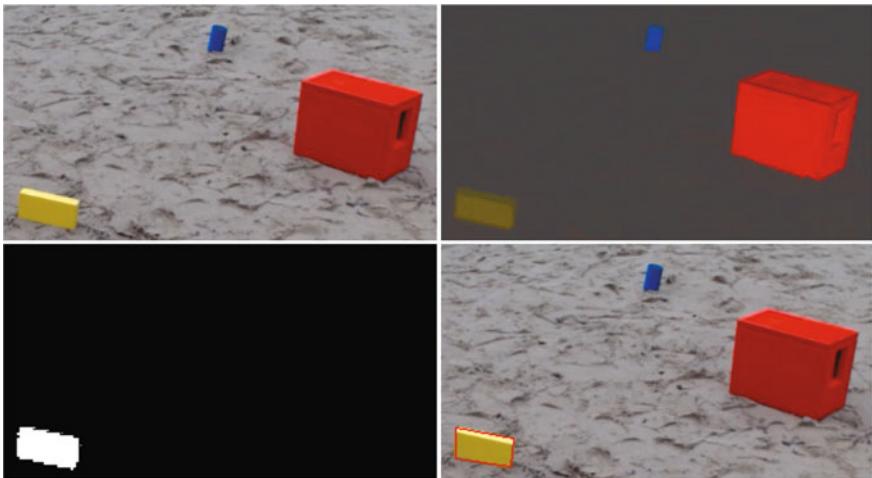


Fig. 6 Exemplary image showing prevalence of colours and intermediate processing result for the battery

the image to get regions of interest for the objects to be detected. Consecutive contours of the resulting connected areas are extracted and analysed, using some properties of the objects such as their expected projected shapes. The implementation is applying the OpenCV framework in version 2.4.8.

First the image is converted to HSV colour space. In the thresholding step, the image is simply clipped to the interesting part of the hue-channel associated with each object's expected colour. The thresholds for each object were manually tuned with a custom rqt interface during the preparation sessions of the competition.

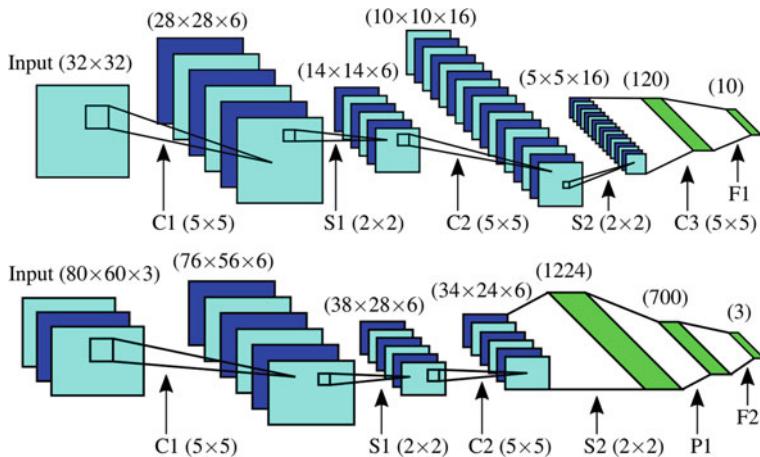
After the thresholding, the image is opened (erode followed by a dilate) to get rid of small artefacts and smoothen the borders of the resulting areas. Then the contours of the thresholded image are extracted (`findContours`) and the detected contours are simplified using the Douglas–Peucker algorithm (`approxPolyDP`). The resulting simplified contours are checked for certain properties to be considered a valid detection (Fig. 6).

Although the employed method is rather simplistic, with correctly tuned thresholds, it is able to detect a good amount of objects in our test sets (footage from recordings made during flight with `rosbag`²⁰) of about 99%, while only generating a low number of false positives (0.3%), see Table 2 for more details.

²⁰<http://wiki.ros.org/rosbag>.

Table 2 Results of blob based detection over image sets extracted from recorded test flights

Image set	Correctly found objects	Incorrectly found objects	False positives
Base	686	6	13
Base and cup	1303	2	0
Battery	1174	0	6
Batt. and base	1407	12	2
Batt., base and cup	1359	3	0
Batt. and cup	1069	0	8
Cup	524	0	8

**Fig. 7** Original network structure of LeNet-5 (*top*) and resulting network structure for the detection task (*bottom*)

6.2 Convolutional Neural Network

Convolutional neural networks gained a lot of popularity for generic object detection tasks. For the problem at hand, the tiny-cnn library²¹ has been selected. Basically the original LeNet-5 network architecture and properties were used (although layer F6 has been omitted), see Fig. 7.

LeNet-5 was originally designed to solve the MNIST [17] Optical Character Recognition (OCR) challenge where single handwritten digits had to be recognized. The images are grey-scale, 28×28 pixel in size and are usually padded to 32×32 pixel which is the default input size of the LeNet5. The example implementation employed Levenberg- Marquardt gradient descend with 2nd order update, mean-squared-error loss function, approximate tanh activation function and consisted of 6 layers: 5×5 convolution with 6 feature maps, 2×2 average pooling, partially

²¹<https://github.com/nyanp/tiny-cnn>.

connected 5×5 convolution with 16 feature maps, 2×2 average pooling and again a 5×5 convolution layer producing 120 outputs which are fed into a fully connected layer with 10 outputs, one for each digit. This approach was selected as it builds upon a time-proven and for today's standards quite small codebase allowing it to perform fast enough on the UAV. More sophisticated approaches for scene labeling or generic object detection methods may easily exceed the processing time requirements and computational constraints of the platform.

In order to adapt the LeNet-5 architecture to our use case we tested various modifications. At first the input colour depth was extended to support also RGB, HSV and YUV (as well as YCrCb) colour spaces or components thereof while keeping the rest of the network architecture as described above, except for the last layer output which was reduced to three neurons, one for each object class. Of course, increasing the input depth increases time needed for training as well as classification but this is most likely outweighed by the benefits that the colour channels provide, because colour is expected to be a key characteristic of the mostly textureless target objects.

Camera images with a resolution of 640x480 pixels are down-scaled and stretched vertically to fill the quadratic input. Stretching was preferred over truncating because it does not reduce the field of view and therefore maintains the search area as large as possible. Features extractors inside the network will be established during training so it can be expected that the vertical distortion will not affect the network performance because it is trained with similar stretched images. Apparently the objects do not incorporate enough structural information to learn a usable abstraction at down-scaled image size. Although the CNN was able to achieve high success rates (close to 100%) and high enough frame rates on the images from the test set, those results were overall highly over-fitted and not usable in practice, since apparently the classifier learned mostly features from the background instead of the depicted object. The image of the objects only cover a comparatively small number of pixels so that the background could have a similar pattern by accident. This fact is especially noticeable with grey-scale images under low light conditions and seemed to support the decision of using coloured images. Next, we determined the best-suited colour representation while increasing the networks's size only slightly from 36×36 to 48×48 pixels.

Results using three different popular colour spaces were quite similar - with a slight advantage of RGB and YUV over HSV. Though the resulting classifiers were still overfitting and training results remained unusable in practise. The advantages of the mentioned colour representations may arise from the fact that they use a combination of 2 (YUV) or 3 (RGB) channels to represent colour tone (instead of HSV having only one), which results in more features extracted from chromaticity. RGB and YUV might also be superior in this case because the object colours match different channels of those representations and thus result in good contrast in these respective channels which is beneficial for feature extraction and may aid learning. Also image normalization did not improve detection rate. In case of local contrast adjustment the results got worse, maybe because distracting background details were amplified. Combinations with more channels like HUV, RGBUV, and others resulted in only marginal improvements. In consequence, they could not justify the additional computation.

In order to make the objects easier to recognize they need to be depicted larger so that they contain at least a minimum amount of structure in the analysed image. Therefore, the input size has to be sufficiently large and networks with input image dimensions of 127×127 pixels were successfully tested. Overfitting was reduced drastically and the solution started to become much better with respect to correctness yielding false positive rates of less than 15%. Unfortunately, the performance degraded vastly so that this solution was not applicable on the UAV. During testing of larger and deeper networks with up to 4 convolutional layers and larger pooling cardinality it turned out that the performance did not improve any more but training time increased considerably.

Furthermore, shallower networks with less feature extracting convolutional kernels and layers were tried. Also, the input size was reduced to an amount that was computationally feasible on the UAV but still preserved the most significant features of objects like edges when they are not too small. 80×60 pixels appeared to be the best size. This is also preserving the camera image aspect ratio.

Experiments showed that two convolutional layers (C1 and C2, C2 partially connected) were sufficient (removing the third convolutional layer had no significant impact on accuracy) and that the Multi Layer Perceptron (MLP) can already be fed with a larger feature map from the second pooling layer. This MLP is now 3 layers deep having the first of them (P1) only partially connected to increase learning speed and break symmetry.

However, poor performance on background images without objects still remained. This was changed by reducing the last layer output size back to three neurons (one per object) and adding negative (empty) sample images to the training set. Previously at least one class was correct for each image (either base, battery, cup or background). This was changed so that when a background image was trained, none of the three classes got positive feedback but the expected value at each of them was set to be minimal (-1 in a range from -1 to 1). This seemed to counter the observed overfitting.

Nevertheless, testing the network that was trained with images containing only a single object or no object at all on images showing multiple objects belonging to different classes did not yield the desired results: Instead of showing high activation for each and every object present in the image, the network decided for one of them, leaving the activation for the second object in the picture not significantly higher than those corresponding to an object that is not present in the frame. A possible alternative, which we did not evaluate, would have been to have a binary classifier for each object and afterwards combine their results. We instead addressed this, by altering the training to support any combination of objects. The final network architecture is illustrated in Fig. 7 (bottom).

To further improve detection results, make them more robust and ease automatic evaluation, purely synthetic images were generated from the MORSE Simulation and added to the training sets as well as placing rendered objects on random structured images from websearches (mostly sand, rocks and similar textures). This way it was possible to get ground truth more easily since it is readily accessible in the generating context instead of manually adding object position labels to the captured images. Sample statistics after training are listed in Table 3 and overall similar

Table 3 Confusion matrix of CNN after 52 epochs of training (94.4% accuracy)

True\detected	None	Base	Battery	Base and batt.	Cup	Base and cup	Batt. and cup	All three	Sum
None	572	0	4	0	3	0	0	0	579
Base	1	557	0	0	0	11	0	0	569
Battery	14	0	566	0	0	0	0	0	580
Base and batt.	9	33	4	538	3	2	0	1	590
Cup	4	0	0	0	600	0	2	0	606
Base and cup	0	3	0	0	4	553	0	5	565
Batt. and cup	0	0	3	0	58	0	505	0	566
All three	0	1	0	22	9	55	9	505	601
Sum	600	594	577	560	677	621	516	511	

accuracy (~94%) was observed with a larger number of images from different sources (captures, rendered and composite images).

6.3 Object Localisation and Results

Due to the fact that the CNN only reports the presence of objects in the provided image and not its particular position, we implemented a sliding window approach to localise them. About 30% overlap of consecutive image regions was used for this purpose and splitting any region with a positive detection up into 9 overlapping subregions until the object is no longer found or a sufficient accuracy is reached. Even then the detection may be focused on any salient part of the object (not necessarily its centre) which introduces a certain error into the localisation. However, this process of localising the object within the image makes the approach too computationally complex to be effectively used on the UAV at this point, although it provides some room for trade-offs between time spent and resulting accuracy. Since the detection has to run integrated with all the other computations on the UAV platform and the localisation within the image is crucial for the map projection, we decided to mainly rely on the simpler but efficient blob detection approach.

The localisation of the objects can be estimated efficiently using the blob detection approach. The centre position of the detected object as a localisation within the image is given as a result of the detection. This centre position is further transformed into the world frame by projecting a ray from the camera origin, applying a pinhole camera model, into the xy-plane. The calculation is simplified through the assumption of having a flat ground and just considering the current altitude of the UAV. All frame conversations are based on the handy ROS *tf* package, here the projection vector is converted into the world frame resulting in the estimated object localisation in world coordinates.

Future work could explore automatic learning of the object positions using CNN in order to reduce the overhead of the image localisation as well as investigating a combination of CNN as a first stage and using the blob detecting for localisation in a second stage, or running them side-by-side if (likely) they make different errors and the detections can be combined or validated. Moreover, the actual localisation could benefit from a consideration of the terrain model provided by the SLAM map.

7 Collision Avoidance and Path Planning

In order to guarantee a safe flight during the competition the UAV needs methods for collision avoidance and path planning.

The term collision avoidance subsumes a number of techniques, which protect the UAV from direct threats. During the flight the sensors will recognize local obstacles and the collision avoidance will calculate a safe flight direction once a possible hazardous situation is determined.

Path planning algorithms usually use a map to plan a path to a desired target position. The calculated path consists of way points that can be headed for and is preferably optimal to save time and energy.

For unexplored and thus unpopulated regions in the map paths to the target locations are initially planned without considering possible obstructions. In this case local collision avoidance prevents the UAV from colliding with obstacles as they are encountered.

The available ROS navigation stack²² is targeting ground vehicle only and more complex as needed in the assessable navigation scenario of the SpaceBot Cup. For this reason a computationally lightweight solution was implemented, which is explained in the following subsections.

7.1 Collision Avoidance

Collision avoidance can be achieved using *Potential Fields*. The Potential Fields approach can be envisioned as an imaginary force field of attracting and rejecting forces that the UAV is surrounded with. The target position generates a force of attraction and obstacles push the UAV away. A new heading can be calculated by means of simple vector addition. The following equation is describing this relationship:

$$\mathbf{F}(q) = \mathbf{F}_{Att}(q) + \sum_{i=1}^n \mathbf{F}_{Rep_i}(q)$$

²²<http://wiki.ros.org/navigation>.

q is the actual position of the UAV, $\mathbf{F}_{Att}(q)$ is a vector, which shows the direction of the target, and $\sum_{i=1}^n \mathbf{F}_{Rep_i}(q)$ describes the repulsion of all obstacles in the environment.

$\mathbf{F}(q)$ is the resulting vector, i.e. the heading the UAV should use to move.

The following functions (from Li et al. (2012) [18]) can be used to calculate attracting and rejecting forces:

$$\mathbf{F}_{Rep_i}(q) = \begin{cases} -\eta \left(\frac{1}{\|q_{obs} - q\|} - \frac{1}{q_0} \right) \frac{1}{\|q_{obs} - q\|^2} \frac{q_{obs} - q}{\|q_{obs} - q\|} & \text{if } \|q_{obs} - q\| < q_0 \\ 0 & \text{if } \|q_{obs} - q\| \geq q_0 \end{cases}$$

$$\mathbf{F}_{Att}(q) = \begin{cases} \zeta(q_{goal} - q) & \text{if } \|q - q_{goal}\| \leq d \\ d\zeta \frac{q_{goal} - q}{\|q - q_{goal}\|} & \text{if } \|q - q_{goal}\| > d \end{cases}$$

q_0 , d , ζ and η are parameters that can be used to adjust attraction and rejection. q_{obs} represents the position of an obstacle and $\|\cdot\|$ is the Euclidean norm.

Hence, *Potential Fields* efficiently calculates a heading of the UAV that leads away from the obstacles and, at best, directly points to the target.

It may happen that attracting and rejecting forces cancel out. In this case the UAV is caught in a local minimum. In order to address this problem the potential fields approach can be combined with path planning, as presented in the next subsection.

7.2 Path Planning

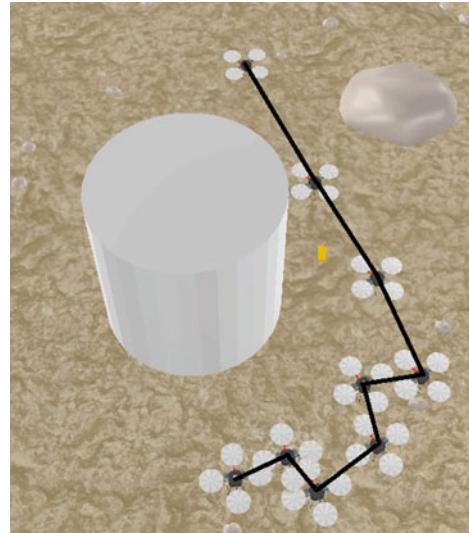
Having a map of the environment that splits the space into discrete segments allows applying search algorithms for path planning to find the shortest path in the graph. Suitable representations for this kind of purpose are *Occupancy Grid* [19], which divide the two-dimensional space into squares and rectangles, and *Octomaps* [20], which provide a volumetric representation of space in the form of cubes or voxels.

In our architecture we applied *D* Lite* as the path planning algorithm. *D** Lite is an incremental heuristic search algorithm that has been developed by Likhachev und Keonig [21]. *D* Lite* repeatedly determines shortest paths between the current position of the system and the goal position as the edge costs of a graph change while the UAV moves towards the goal.

7.3 Summary

For the considered scenario it is sufficient to use *Potential Fields* to successfully navigate the mission area. If the UAV moves to close to an obstacle, Potential Fields will calculate the necessary evasive manoeuvres. We can ignore the problems arising

Fig. 8 The behavior of the collision avoidance in simulation



from local minima since there is only a small number of obstacles in the altitude the UAV operates in and these obstacles are not having particularly complex shapes.

However, we apply D* Lite to always have a valid flight path under the assumption that the UAV's positions is adequately tracked. Changing the resolution of the map allows to adjust the computational load caused by D* Lite. Providing more spare computation time enables other modules to execute more intensive calculations.

The separation of local and global planner has also been proposed by Du et al. [22]. This results in faster reaction times and optimal path planning. As a consequence, collision avoidance and path planning are both passive in our architecture as long as no obstacle is detected. During this normal execution the targeted path is calculated by the exploration node and given to the *path_follower*. When the collision avoidance becomes active (i.e. the UAV is in the influence sphere of a potential field) the original path will be overwritten by a new pose, namely the avoidance heading, to master the urgent danger (see also Fig. 8). After, the path planning module is triggered, if the potential fields approach has not yet sufficiently resolved the approaching collision, in order to provide a suitable path to the target considering the obstacle.

8 Autonomous Exploration of Unknown Areas

A main task of the UAV is exploration, which means mapping the environment and navigating to unknown areas. This section describes the analysis and evaluation of a set of exploration strategies with the goal to use the best one in our UAV software. The quality of an exploration strategy can be measured in the time it needs to cover a specified area, the required computation power and the precision of the

resulting map. Since the UAV has a very limited time of flight of roughly 15 min, a rapid exploration with low computation requirements is preferred and map precision analysis is omitted.

8.1 Simulation

For the purpose of gathering performance data on the exploration strategies for comparison, a simulation environment has been developed, capable of representing the UAV and its field of view in a flat $40\text{ m} \times 30\text{ m}$ world without obstacles except the limiting outer walls. The UAV flies with a speed of 0.2 m s^{-1} and rotates 5° s^{-1} . Exploration modules can be switched flexibly due to a minimalistic, ROS-like interface: Exploration is a function getting the current robot configuration (`PoseStamped` via `TransformListener`) and a world representation (`OccupancyGrid` from the SLAM service) and returning the next best view, i.e. favoured robot configuration (`Pose`). The general handling of the exploration process, like further passing on the target poses to the `path_follower` is handled in a simple collision avoidance behaviour, which makes use of the package discussed in this section.

The strategy performance is logged in matters of map coverage over time, distance flown, rotations made and time needed to discover the whole world. Additionally, in each simulation run three objects are randomly placed in the world and their discovery times are logged.

8.2 Exploration Strategies

Five exploration strategies have been examined, of which a typical path is shown in Fig. 9 each. **Random flight** (9a) makes the UAV steer straight until a wall is reached. Then it proceeds in a random angle, until there are no more frontier cells. **Concentric circles** (9b) is a rather static strategy, which makes the UAV fly an enlarging spiral path around the starting point. If a wall is reached, the circular path is given up and replaced through a temporary wall-following behaviour. The **SRT** (9c) method presented in [23] builds a *Sensor-based Random Tree* by probing the local safe area for unexplored cells and returning to the previous one when there are none left. The **utility-based frontier approach** (9d) is similar to Yamauchi's frontier-based exploration [24], but uses a utility function to decide which frontier to visit based on distance and information gain. Similar work has been done in [25, 26]. A variant with penalised rotation has also been tested (9e), addressing the general problematic performance of determining or handling the orientation with SLAM and odometry. The last strategy uses a **genetic algorithm for sampling** (9f). The algorithm mutates and recombines a pool of randomly generated, frontier-based robot configuration samples over a fixed number of generations. The mutation creates a new sample by a normally distributed random alteration of position and orientation. The recombination yields a sample

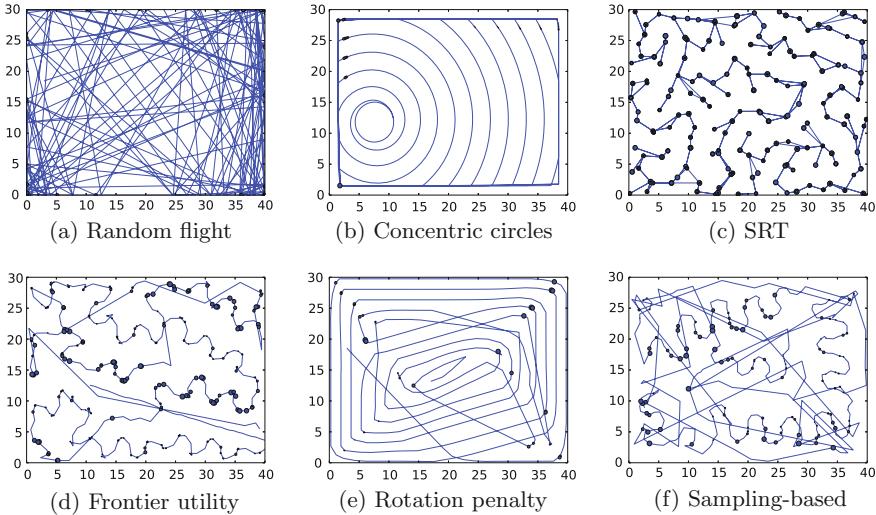


Fig. 9 Typical exploration paths

with position and orientation taken from two different random samples. In the end of each generation step, a fixed number of best samples is selected by following utility function U .

$$U := \frac{C}{\max(t_R, t_T)}$$

The function estimates the expected information gain (newly discovered cells C assuming flat ground and a given camera setup) per time needed to reach this configuration, with t_R and t_T being rotation time and translation time respectively.

8.3 Evaluation and Results

On each strategy data has been collected over 100 runs each with the same pool of random-generated starting scenarios, consisting of an initial robot configuration and three object positions. Figure 10 shows the average proportion of visited cells over time for all strategies.

The “ideal exploration” rate is the theoretical value of the UAV flying straight through unknown area at maximum allowed speed. The utility-based frontier approach finishes first taking 50 min on average. On the other hand, the sampling-based strategy shows the highest exploration rate in the early phase of exploration, finding the objects first and having covered the most area after 10 min.

Nevertheless, the evolutionary non-deterministic sampling algorithm requires more computational resources even if the processed generations are limited, see

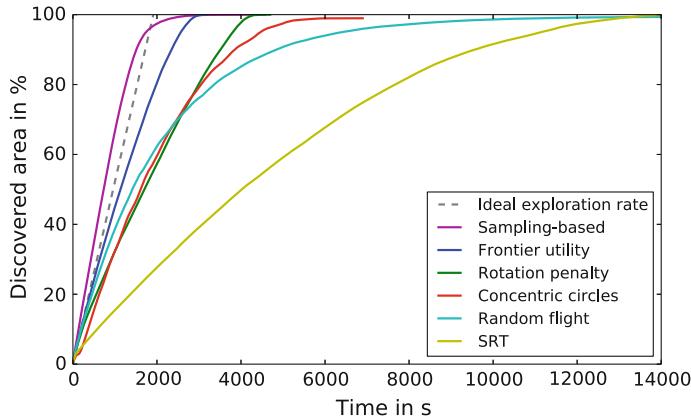


Fig. 10 Exploration progress over time

Table 4 Computation time for one waypoint in average over the full exploration area

Strategy	Average computation time in ms
Sampling-based	1271
Frontier utility	28
Concentric circles	0
Random flight	7
SRT	3

Table 4 for a comparison. Hence, the utility-based frontier approach is the preferred exploration strategy for our setup in SpaceBot Cup, because it combines a fast exploration with reasonable requirements of computation performance. The implemented ROS package *uav_exploration* including the specific simulation environment is available online.²³

9 Autonomous Behaviour

Developing systems that are able to react appropriately to unforeseen changes while still pursuing its intended goals is challenging. As discussed in [27], adaptivity in general, and fast and flexible decision making and planning in particular, are crucial capabilities for autonomous robots. Especially in the Robot Operating System (ROS) community [28] developers are so far mostly using pre-scripted, non-adaptive methods of describing the high-level robot behaviours or tasks. A popular package is *SMACH* that allows to build hierarchical and concurrent state machines (HSM) [29]. All kind of state machine based approaches have the problem that a decision

²³https://github.com/DAInamite/uav_exploration.git.

or reaction can only be given if a state transition was already modelled in advance. Behaviour trees, available in the *pi_trees* package [30], are an alternative that allows for more dynamic rules. More flexible is the BDI-based implementation *Cogni-TAO* [31] available in the *decision_making* package. The concept is more suitable for uncertain environments because the execution sequence is not fixed and the selection of behaviours (plans) is based on conditions. Nevertheless, it is still difficult to define mission or maintenance goals and there exist only simple protocols for plan selection.

In order to provide a flexible, adaptive and goal-driven alternative we are working on a new hybrid approach with tight ROS integration that incorporates features from reactive behaviour networks and STRIPS-like planning. Even though, such a system can perform less optimal, it will support execution in dynamic environments.

The behaviour network itself strongly supports the idea of an adaptive robotic system by being

- opportunistic and trying to perform the best-suited action at any time even if the symbolic planner cannot handle the situation and does not find a suitable plan;
- light-weight in terms of computational complexity;
- performing well in dynamic and partially observable environments under the assumption that actions taken at one point in time do not block decision paths in the future.

The first expansion stage of our concept, called *ROS Hybrid Behaviour Planner - RHPB*, is going to be advanced in future for instance with extended multi-robot support, incorporating learning and more hierarchical layers. The current architecture of our implementation contains three core layers: the behaviour network itself represented by its distributed components, the symbolic planner and a manager module. It is available online.²⁴

The manager module supports and manages the distributed execution of several behaviours on different machines within the robot. Furthermore, it is monitoring and supervising the behaviour network by interpreting the provided plan and influencing the behaviour network accordingly.

The following subsections provide required background in order to understand our framework and implement autonomous behaviour with it. Furthermore, the implemented UAV behaviours are presented.

9.1 Behaviour-Network Base

The behaviour network layer is based on the concepts of Jung et al. [32] and Maes et al. [33], but incorporates other recent ideas from Allgeuer et al. [34], in particular supporting concurrent behaviour execution, non-binary preconditions and effects.

²⁴<https://github.com/DAInamite/rhbp>.

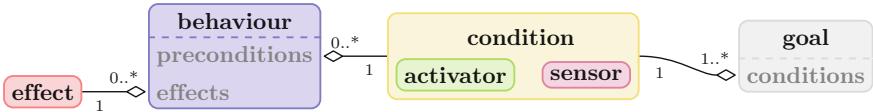


Fig. 11 Behaviour network components

The main components of the network are behaviours representing tasks or actions that are able to interact with the environment by sensing and acting. *Behaviours* and *goals* both use *condition* objects composed of *activator* and *sensor* to model their environmental runtime requirements, see Fig. 11. The network of behaviours is created from the dependencies encoded in wishes based on preconditions and effects.

Each behaviour expresses its satisfaction with the world state (current sensor values) with wishes. A *wish* is related to a sensor and uses a real value $[-1,1]$ to indicate both the strength and direction of a desired change, 0 indicates complete satisfaction. Greater absolute values express a stronger desire, by convention negative values correspond to a decrease, positive values to an increase.

Effects model the expected influence to available sensors (the environment) of every behaviour similar to wishes.

Goals describe desired conditions of the system, their implementation is similar to behaviours except that they do not have an execution state or model effects on the network. Therefore, goals incorporate conditions that allow for the determination of their satisfaction and express wishes exactly like behaviours do. Furthermore, goals are either permanent and remain active in the system as maintenance goals, or are achievement goals that are deactivated after fulfilment.

Sensors model the source of information for the behaviour network and buffer and provide the latest sensor measurements. Virtual sensors can also be used to model the world state, for instance the number of detected target objects. The type of the sensor value is arbitrary, but to form a condition a matching pair of sensor and activator must be combined.

Due to the fact that raw sensor values can be of arbitrary type they need to be mapped into the behaviour network by *activators*. Activators compute a utility score (precondition satisfaction) from sensor values using an internal mapping function. The separation of sensor and activator fosters the reuse of code and allows also the abstract integration of algorithms using more complex mapping functions like potential fields. Our implementation already comes with basic activators for expressing a threshold-based and a linear mapping of one-dimensional sensors. Multi-dimensional types can either be integrated by custom activators that provide a normalisation function or by splitting its dimensions into multiple one-dimensional sensors.

The key characteristics and capabilities of a behaviour network are defined by the way activation is computed from sensor readings and the behaviour/goal interaction. Behaviours are selected for execution based on a utility function that determines a real number behaviour score, called activation. There are multiple sources of activation, negative values correspond to inhibition. If the total activation of a behaviour

reaches the execution threshold and all preconditions are fulfilled the planner selects it for execution, several behaviours can be executed in parallel. The behaviour network calculation is repeated in fixed frequency that can be adjusted according to the application requirements.

At every iteration all activation sources are summed to a temporary value called activation step for every behaviour. After the activation step has been computed for every behaviour it is added to the current activation of the behaviour reduced by an activation decay factor. The decay reduces the activation that had been accumulated over time if the behaviour does not fit the situation any more and prevents the activation value from becoming indefinitely large.

After behaviour execution the activation value is reset to 0. Behaviours are not expected to finish instantaneously and multiple behaviours are allowed to run concurrently, if they are not having conflicting effects.

9.2 Symbolic Planner Extension

The activation calculation is influenced by the symbolic planner based on the index position of the particular behaviour in the planned execution sequence. In order to allow for a quick replacement of the planner we based our interface on the widely used Planning Domain Definition Language (PDDL) in version 2.1. Hence, a majority of existing planners can be used.

For our implementation we developed a ROS Python wrapper for the Metric-FF [35] planner, a version of FF extended by numerical fluents and in the current version also conditional effects. It meets all our requirements (negated predicates, numeric fluents, equality, conditional effects) and due to its heuristic nature favours fast results over optimality. In fact the wrapper is only responsible for appropriate result interpretation and execution handling.

The actual mapping and translation between the domain PDDL and the resulting plan is part of the *manager*. The PDDL generation on entity level is done automatically by the behaviour, activator and goal objects themselves through a defined service interface. Moreover, the manager monitors time constraints defined in behaviours, re-plans in case of timeouts, new available behaviours or if the behaviour network execution order deviates from the proposed plan. This ensures that replanning is only executed if really necessary and keeps as much freedom as possible for the behaviour network layer for fast response and adaption.

The manager also handles multiple existing goals of a mission by selecting appropriate goals at the right time depending on available information, for example if goals can not be reached in the moment.

9.3 ROS-Integration

All components of the RHBP are based on the ROS messaging architecture and are using ROS services and topics for communication. Every component of the behaviour network, like a behaviour or sensor, is automatically registered to the manager node and reports its current status accordingly, for details see Fig. 12. The application specific implementation is simplified through provided interfaces and base classes for all behaviour network components that are extended by the application developer and completed by filling hooks, like start and stop of a behaviour. The class constructor automatically uses registration methods and announces available components to the manager. The ROS sensor integration is inspired by Allgeuer et al. [34] and implemented using the concept of virtual sensors. This means sensors are subscribed to ROS topics and updated by the offered publish-subscribe system.

For each registered component a proxy object is instantiated in the manager to serve as data source for the actual planning process where the activation is computed based on the relationships arising from the reported wishes and effects. Besides the status service offered by behaviours and goals there are a number of management services available to influence the execution, see Fig. 12.

Due to the distributed ROS architecture the whole system works even across the physical boundaries of individual robots on a distributed system.

Furthermore, RHBP comes with generic implementations that directly support simple single dimensional topic types for numbers and booleans to enable direct integration of existing sensors by just configuring the topic name. Moreover, activators for some common ROS types are provided as well and are going to be extended in future.

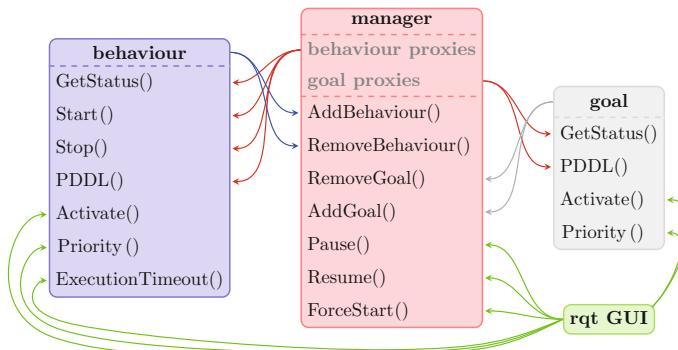


Fig. 12 ROS services used by the behaviour network components arrows indicate the call direction

9.4 SpaceBot Cup UAV Behaviours

In Sect. 4 the behaviours for addressing the SpaceBot Cup challenge have already been mentioned from the architectural point of view. As being said, the more complex algorithms and computations for exploration and collision avoidance are separated into own packages, already discussed in Sects. 7 and 8.

In order to implement the desired behaviour we implemented the behaviour model illustrated in Fig. 13. In order to do so the provided behaviour base class have been extended for the individual behaviours. Available sensors and abstracted information of the system have been integrated as virtual sensors into the RHBP framework. For that it was necessary to implement some special sensor wrappers, which extract the needed information from complex ROS message types, like poses (TFs). Furthermore, a special distance activator was implemented to determine the activation in the network based on a *geometry_msgs.msg.Pose* and a desired target pose. The exploration sensor is a wrapper for the exploration module that describes the completeness of the exploration.

The realised UAV capabilities are to take off and land (regularly at the landing zone after the mission is completed, the time is over, or the battery is depleted or anywhere else in emergency situations), select a position to move to (performed by exploration or return-to-home behaviour and overridden by the obstacle avoidance), and move

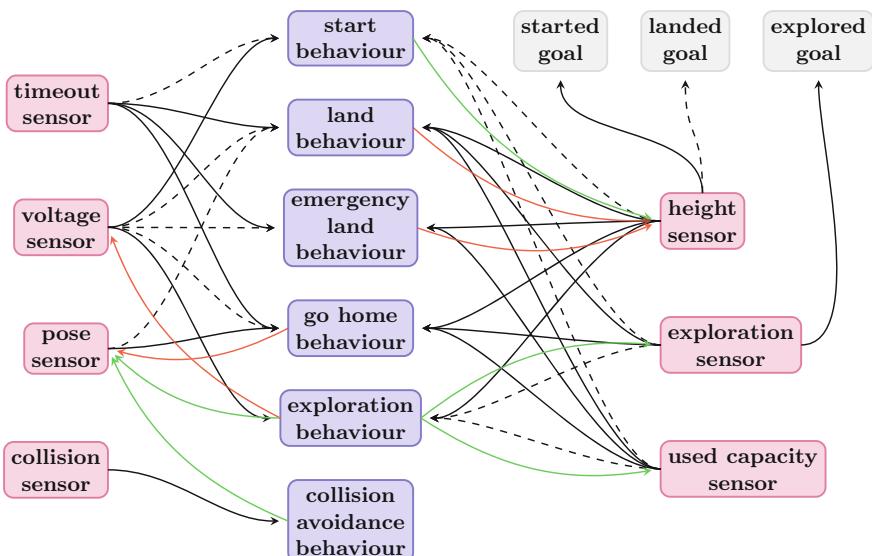


Fig. 13 SpaceBot Cup Planning Scenario Black arrows indicate (pre-) conditions. *Solid black arrows* (\longrightarrow) indicate desire for high value, *dashed arrows* ($- - \rightarrow$) desire for low value. *Green arrows* ($\textcolor{green}{\longrightarrow}$) mean positive correlation (increase, become true), *red arrows* ($\textcolor{red}{\longrightarrow}$) indicate negative correlation (decrease, become false). Pose sensor values encode distance from home

to the selected location while maintaining constant altitude over ground. While it is operating, the UAV continuously maps the terrain and searches for objects. These activities do not need to be turned on or off explicitly. Given the initial situation that the aircraft is fully charged, on the ground, at the landing zone (also referred to as “home”), and the mission starts, the network will activate the start behaviour first and then cycle between exploration (which retrieves a target location) and, if required, collision avoidance, (thereby mapping the terrain and scanning for objects) until it runs out of battery or completed its exploration mission. Finally, it will select the home location as target, move there and land.

10 Teamwork

The basic idea that the UAV is the flying eye of the rover only makes sense when both vehicles communicate about map disclosure and object positions. In an ideal both vehicles would send their map update to each other and each vehicle would merge the update with the own existing OctoMap.

Due to the limited computational power of the UAV and due to the fact that the UAV moves much faster than the rover, we finally decided to only let the UAV send map updates and known object positions to the rover.

Finally, we simplified our approach in the way, that the rover knows the initial offset to the UAV (where it has been positioned before start relative to the rover’s position). The UAV sends object positions in its own coordinate system. Using the tf transformation the rover can now mark the objects in its own map.

For the realization of the team communication we base on the *ROS multimaster_fkie* package together with the *master_discovery_fkie* and *master_sync_fkie* packages. This enables having several independent ROS cores in one local area network. The *master_discovery* node multicast messaging can be used as we are in the same subnet. Hence, the UAV is publishing the coordinates of discovered objects as tf, which can be received and transferred in the independent ROS system of the rover.

A still open challenge is the fusion of the two independent OctoMaps in order to iteratively update a common map of both robots. This is probably a computation intensive task and for this reason is planned to be realised within the ground station. An alternative approach would be merging the two-dimensional occupancy maps instead, for instance with the not further evaluated solution in the package *map_merging* [36].

11 Results

Several results specific to individual modules or insights gained during the development of them have been presented and discussed in the module related sections

before, for instance for navigation, exploration and object localisation. In the following we are considering the common capability set and limitations of our approach.

In general the developed UAV is able to autonomously start, land, hover on a position, follow given trajectories and detect the target objects of the mission. Moreover, the paths or trajectories are generated by the exploration module and collisions are avoided with the potential field approach, while the whole process is controlled by a high-level goal-oriented decision-making and planning component. The capabilities have been empirically tested in simulation, laboratory environment and the contest itself. The navigation and the object localisation performed robustly in the very unstructured environment without feature-rich textures.

The finally integrated system with the above presented components is successfully running onboard of our hardware platform, with the CPU having almost 100% load. However, the system is responsive and able to execute all modules in an appropriate refresh rate. Moreover, some modules, like SLAM are still having potential for performance improvements.

Table 5 provides a more detailed overview about the produced load and update rates on our two core system (max. 200% load). The memory usage can be neglected, the whole system consumes less than 1 GB with an initial map. However, the table illustrates that most CPU load is generated by the processing of the visual camera data in the object detection and localisation and mapping.

Before we tested our system on the hardware platform as well as for speeding up the development of individual modules we have extensively used the MORSE simulation environment in version 1.3-1 [37]. Due to the 3D engine and high level

Table 5 Comparison of node CPU consumption and update frequencies. The separated categories group the nodes into sensors/actors, navigation, object localisation and higher-level behaviour (top to bottom)

Node	CPU Load in %	Update frequency in Hz
mikrokopter	3	50
px4flow	2	40
sonar_sensors	2	25
bluefox_camera	7	30
slam_odom_manager	1	30
orb_slam	85	30
position_controller	6	24
optical_flow_to_tf	2	40
path_follower	1	5
object_detection_blob	78	3
object_localisation_estimator	4	3
exploration	6	5
collision_avoidance	2	25
uav_behaviours and planning	1	1

sensor interfaces of the simulation environment we have been able to even test the computer vision related SLAM and object localisation modules. For testing modules based on the PX4FLOW as well as the lower level MikroKopter control we implemented custom actuators and sensors that provide or receive data in the same ROS message formats as the originals. Accordingly, we have been able to remap ROS topics provided by the simulator to the actual names in our hardware configuration in order to simulate our mission. Due to the limited hardware capabilities of our UAV the complete ROS software stack can also be executed together with the simulation environment on a common business notebook (Lenovo Thinkpad T440s with Intel Corei7, 16 GB RAM, SSD and integrated graphics) providing similar performance as the actual hardware. MORSE has been favoured over alternative simulation environments, like Gazebo, because it has already been used by our partners implementing the rover robot. Furthermore, MORSE is very easy to extend, due to its Python and Blender origin, for instance complex 3D models can be imported and added easily in Blender.

As expected the system performs very well in simulation, because of noise-free sensor data and a less dynamical environment. Nevertheless, our real system has some limitations and open issues, we want to discuss in the following.

The altitude hold performance is suffering from the low resolution thrust control of the flight controller (8bit for the full motor speed range). Thus the PID controller running in the position controller has problems in keeping the altitude without ongoing regulations, since the thrust difference between two values can be too large.

The contest itself was executed in two stages having a qualification stage and the final competition. The qualification was held in a smaller arena with simplified and separated tasks for the robots. During the preparation as well as the two parts of the competition we experienced several defects and problems. In particular we had massive problems with the used flight controller, which had hardware problems on several of our boards resulting in temporary IMU acceleration inversions of the z-axis. In consequence our system needed to survive some heavy crashes, even one in the qualification run, which could be fixed by replacing rotors, arms and the 3D-printed platform parts.

In that sense, we do not recommend to use the MikroKopter FlightCtrl we have used for future developments and rather propose alternative solutions like the PIXHAWK platform.

Furthermore, we are not satisfied with the orientation estimation of our odometry subsystem based on the PX4Flow and the additional IMU. The integrated orientation from the PX4Flow gyroscope is drifting over time and the IMU together with the used package *razor_imu_9dof* is suffering from fast movements resulting in bad performance. Here, this could be simplified by replacing the flight controller with one that is already coming with a well configured orientation estimation. An alternative approach could also use more advanced sensor fusion and filtering in order to improve the orientation estimation with existing sensors, for instance by applying a Kalman filter.

As well related to the navigation subsystem is the SLAM and odometry combination, here our approach is working in general, but is sometimes not as robust

as required. In particular the scale estimation of the monocular SLAM is prone to errors resulting in drifts due to the uncertain reference from the odometry. Unexpected problems during the scale determination result in a deviated scale reference for the SLAM. Future improvements could consider a continuous scale update, using an ongoing feedback loop during successful SLAM tracking, special initialisation flight routines or even completely resolve the issue by replacing the monocular SLAM with a stereo-vision-based approach.

12 Conclusion

The intention of this chapter was to provide reference, insights and lessons learned on the development of an autarkic UAV on the basis of the ROS framework. The chapter exploits the DLR Spacebot Cup scenario for the exploration of unknown terrain without an external tracking system. The UAV is thought as assistance robot for an autonomous ground rover that is capable of grasping larger objects from the ground. The UAV is more agile than the rover thus acting as supplemental sense of the ground vehicle, aiming at map disclosure and object detection.

We use ROS on the UAV as middleware and at the same time make use of the rich ROS module repository together with own ROS modules to create our own architecture that is able to control the UAV in a mission-oriented way. Some ROS modules could be used out-of-the-box, but other existing modules have been extended and adapted to our needs (e.g. *ORB_SLAM* and *px-ros-pkg*). An instance of this architecture is deployed on extended but still limited hardware (in terms of CPU and RAM) on the UAV alone and bridges multiple sensor inputs, computational control to actuator outputs.

Our architecture makes extensive use of the ROS architecture patterns *topic*, *service* and *action*. Topics are generally used as intended as means for unidirectional data transfer, e.g. continuously reading ultrasonic sensors and streaming the data to the *collision avoidance*. Services provide responses, so we use them to change states or explicitly retrieve information from other nodes, e.g. for setting a new target and its confirmation or getting waypoints from the *exploration_node*. Long-running tasks with periodic feedback are realised with the *actionlib*, for instance the path following.

Except the necessary standard ROS packages, our architecture comprises of about 25 ROS packages including dependencies so far as shown in Sect. 4, classified as Sensor/Actor for hardware connectivity and control, Behaviour and Planning for high-level control, navigation for UAV flight control, Object Localisation as mission-specific code and Infrastructure for commonly used functionality.

As the focus in this book is on ROS we can summarize our experience with ROS as extremely satisfying when executing our system (also from the background that we have used other frameworks and developed our own agent-oriented middleware earlier already). The ROS-based system runs stable and works as expected. We have

been able to prove that it is possible to develop an autarkic UAV with higher level capabilities using the ROS ecosystem.

However, always when we are confronted with hardware and the real world more problems arise. Although the flying base is robust we could not prevent it from transport damage or from mischief during crashes. Although we are using sensors that are embedded a thousand times in other technical systems they will conk out when mounted on a UAV. Although, many people work with PIDs and develop and use SLAM algorithms, there is still a lot to be done, e.g. tuning a PID takes time and in terms of SLAM the jury is still out.

For future work we will focus on high-level and mission-guided control as well as further advancing our autonomous navigation capabilities, while addressing new application beyond the SpaceBot Cup.

Meanwhile, we can assess that multi-rotor systems and other smaller aerial vehicles can fly. But this is only a small part of what the customers want in all application areas of UAV, from agriculture to logistics. Mission-guided control from our point of view means the user can concentrate on the parameters, goal and success of a mission, without struggling with collision avoidance and stability during flight. In terms of ROS, this means that we are working on ROS packages that contain a generic extension for mission specific tasks, that can easily be integrated into behaviour planning, execution and control, and that can easily monitored by human operators.

Acknowledgements The presented work was partially funded by the German Aerospace Center (DLR) with funds from the Federal Ministry of Economics and Technology (BMWi) on the basis of a decision of the German Bundestag (Grant No: 50RA1420).

References

1. Kryza, L., S. Kapitola, C. Avsar, and K. Briess. 2015. Developing technologies for space on a terrestrial system: A cost effective approach for planetary robotics research. In *1st symposium on space educational acitivities*, Padova, Italy.
2. Mur-Artal, R., J.M.M. Montiel, and J.D. Tardós. 2015. ORB-SLAM: A versatile and accurate monocular SLAM system. CoRR. [arXiv:abs/1502.00956](https://arxiv.org/abs/1502.00956).
3. Honegger, D., L. Meier, P. Tanskanen, and M. Pollefeyns. 2013. An open source and open hardware embedded metric optical flow CMOS camera for indoor and outdoor applications. In *2013 IEEE international conference on robotics and automation (ICRA)* 1736–1741.
4. Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11): 2278–2324.
5. Li, G., A. Yamashita, H. Asama, and Y. Tamura. 2012. An efficient improved artificial potential field based regression search method for robot path planning. In *2012 international conference on Mechatronics and automation (ICMA)*, 1227–1232.
6. Loianno, G., Y. Mulgaonkar, C. Brunner, D. Ahuja, A. Ramanandan, M. Chari, S. Diaz, and V. Kumar. 2015. Smartphones power flying robots. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 1256–1263.
7. Tomic, T., K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. Grix, F. Ruess, M. Suppa, and D. Burschka. 2012. Toward a fully autonomous UAV: Research platform for indoor and outdoor urban search and rescue. *IEEE Robotics Automation Magazine* 19 (3): 46–56.

8. Schmid, K., P. Lutz, T. Tomić, E. Mair, and H. Hirschmüller. 2014. Autonomous vision-based micro air vehicle for indoor and outdoor navigation. *Journal of Field Robotics* 31 (4): 537–570.
9. Beul, M., N. Krombach, Y. Zhong, D. Droschel, M. Nieuwenhuizen, and S. Behnke. 2015. A high-performance MAV for autonomous navigation in complex 3d environments. In *2015 international conference on unmanned aircraft systems (ICUAS)*, 1241–1250. IEEE: New York.
10. Sunderhauf, N., P. Neubert, M. Truschinski, D. Wunschel, J. Poschmann, S. Lange, and P. Protzel. 2014. Phobos and deimos on mars - two autonomous robots for the DLR spacebot cup. In *The 12th international symposium on artificial intelligence, robotics and automation in space (i-SAIRAS'14)*, Montreal, Canada, The Canadian Space Agency (CSA-ASC).
11. Endres, F., J. Hess, J. Sturm, D. Cremers, and W. Burgard. 2014. 3-d mapping with an RGB-d camera. *IEEE Transactions on Robotics* 30 (1): 177–187.
12. Labbe, M., and F. Michaud. 2014. Online global loop closure detection for large-scale multi-session graph-based SLAM. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, 2661–2666.
13. Engel, J., T. Schöps, and D. Cremers. 2014. LSD-SLAM: large-scale direct monocular SLAM. In *Computer vision – ECCV 2014: 13th European conference, Zurich, Switzerland, September 6–12, 2014, Proceedings, Part II*, 834–849. Springer International Publishing, Cham.
14. Forster, C., M. Pizzoli, and D. Scaramuzza. 2014. SVO: Fast semi-direct monocular visual odometry. In *IEEE international conference on robotics and automation (ICRA)*.
15. Engel, J., J. Sturm, and D. Cremers. 2014. Scale-aware navigation of a low-cost quadrocopter with a monocular camera. *Robotics and Autonomous Systems* 62 (11): 1646–1656.
16. Izzo, D., and G. de Croon. 2012. Landing with time-to-contact and ventral optic flow estimates. *Journal of Guidance, Control, and Dynamics* 35 (4): 1362–1367.
17. Deng, L. 2012. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine* 29 (6): 141–142.
18. Li, G., A. Yamashita, H. Asama, and Y. Tamura. 2012. An efficient improved artificial potential field based regression search method for robot path planning. In: *2012 international conference on Mechatronics and automation (ICMA)*, 1227–1232. New York: IEEE
19. Thrun, S., D. Fox, and W. Burgard. 2005. *Probabilistic robotics*. Cambridge: The MIT Press.
20. Hornung, A., K.M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. 2013. Octomap: An efficient probabilistic 3d mapping framework based on octrees. *Autonomous Robots* 34 (3): 189–206.
21. Koenig, S., and M. Likhachev. 2005. Fast replanning for navigation in unknown terrain. *IEEE Transactions on Robotics* 21 (3): 354–363.
22. Du, Z., D. Qu, F. Xu, and D. Xu. 2007. A hybrid approach for mobile robot path planning in dynamic environments. In *IEEE international conference on robotics and biomimetics, 2007. ROBIO 2007*, 1058–1063. New York: IEEE.
23. Oriolo, G., M. Vendittelli, L. Freda, and G. Troso. 2004. The SRT method: Randomized strategies for exploration. In *2004 IEEE international conference on robotics and automation, 2004. Proceedings. ICRA'04*, vol. 5, 4688–4694. New York: IEEE.
24. Yamauchi, B. 1997. A frontier-based approach for autonomous exploration. In *Proceedings of the 1997 IEEE international symposium on computational intelligence in robotics and automation, 1997. CIRA'97*, 146–151. New York: IEEE
25. Surmann, H., A. Nüchter, and J. Hertzberg. 2003. An autonomous mobile robot with a 3D laser range finder for 3D exploration and digitalization of indoor environments. *Robotics and Autonomous Systems* 45 (3): 181–198.
26. Tovar, B., L. Munoz-Gómez, R. Murrieta-Cid, M. Alencastre-Miranda, R. Monroy, and S. Hutchinson. 2006. Planning exploration strategies for simultaneous localization and mapping. *Robotics and Autonomous Systems* 54 (4): 314–331.
27. Hrabia, C.E., N. Masuch, and S. Albayrak. 2015. A metrics framework for quantifying autonomy in complex systems. In *Multiagent System Technologies: 13th German Conference, MATES 2015, Cottbus, Germany, September 28–30, 2015, Revised Selected Papers*, 22–41. Springer International Publishing, Cham.

28. Quigley, M., K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A.Y. Ng. 2009. Ros: An open-source robot operating system. In *ICRA Workshop on Open Source Software* 3 (3.2): 5. Kobe.
29. Bohren, J., and S. Cousins. 2010. The SMACH high-level executive [ros news]. *IEEE Robotics Automation Magazine* 17 (4): 18–20.
30. Goebel R.P. 2014. ROS by example: Packages and programs For advanced robot behaviors. Pi Robot Production, vol. 2, 61–88. Lulu.com.
31. CogniTeam Ltd. Cognitao (think as one). [Online]. Available: <http://www.cogniteam.com/cognitao.html>.
32. Jung, D. 1998. An architecture for cooperation among autonomous agents. PhD thesis, University of South Australia.
33. Maes, P. 1989. How to do the right thing. *Connection Science* 1 (3): 291–323.
34. Allgeuer, P., S. Behnke. 2013. Hierarchical and state-based architectures for robot behavior planning and control. In *Proceedings of 8th Workshop on Humanoid Soccer Robots, IEEE-RAS International Conference on Humanoid Robots, Atlanta, USA*.
35. Hoffmann, J. 2002. Extending FF to numerical state variables. In *Proceedings of the 15th European conference on artificial intelligence*, 571–575. New York: Wiley.
36. Yan, Z., L. Fabresse, J. Laval, and N. Bouraqadi. 2014. Team size optimization for multi-robot exploration. In *Proceedings of the 4th international conference on simulation, modeling, and programming for autonomous robots (SIMPAR 2014), Bergamo, Italy (October 2014)*, 438–449.
37. Echeverria, G., N. Lassabe, A. Degroote, and S. Lemaignan. 2011. Modular open robots simulation engine: MORSE. In *Proceedings of the 2011 IEEE international conference on robotics and automation*.

Author Biographies

M.Sc. Christopher-Eyk Hrabia received a degree in computer science from the Technische Universität Berlin (TUB) in 2012. After he gained some international experience as a software engineer, he started his scientific career at DAI-Lab of TUB. He researches in the field of multi-agent and multi-robot system with a focus on high-level control of autonomous, adaptive and self-organizing unmanned aerial vehicles. Moreover, he developed and contributed to several ROS packages and is using ROS for student courses conducted by him. Together with Martin Berger he led the SpaceBot Cup UAV-Team of the DAI-Lab.

Dipl. Ing. Martin Berger after receiving his diploma in computer science in 2012, started as research assistant at DAI-Lab. He is involved in several student projects that teach practical application of robotics in competitive settings. He is a member of the RoboCup team DAInamite and frequently participates in international and national robot competitions.

Dr. Axel Hessler is head of the Cognitive Architectures working group at DAI-Lab. He received his doctor degree in computer science for research in intelligent software agents and multi-agent systems and how they can fast and easily developed and applied in various applicational areas. Currently he is investigating the correlation between software agents, physical agents and human agents.

M.Sc. Stephan Wypler is working as a software engineer in the industry, after he finished his Computer Science (M.S.) at the TUB in 2016. During his M.S. degree he was a core member of the SpaceBot Cup UAV-Team of the DAI-Lab and developed core modules for the higher-level planning, autonomous behaviour and object localisation.

B.Sc. Jan Brehmer is currently studying Computer Science (M.S.) at the TUB. For his B.S. degree he researched autonomous exploration strategies for UAVs at the DAI-Lab. Formerly, he assisted as a tutor at the department for software engineering and theoretical computer science.

B.Sc. Simon Matern is currently studying Computer Science (M.S.) at the TUB. He was working on the collision avoidance algorithms for the UAV in his final project of his B.S. degree.

Prof. Dr.-Ing. Habil. Sahin Albayrak is the head of the chair Agent Technologies in Business Applications and Telecommunication. He is the founder and head of the DAI-Lab, currently employing about one hundred researchers and support staff.

Development of an RFID Inventory Robot (AdvanRobot)

**Marc Morenza-Cinos, Victor Casamayor-Pujol, Jordi Soler-Busquets,
José Luis Sanz, Roberto Guzmán and Rafael Pous**

Abstract AdvanRobot proposes a new robot for inventorying and locating all the products inside a retail store without the need of installing any fixed infrastructure. The patent pending robot combines a laser-guided autonomous robotic base with a Radio Frequency Identification (RFID) payload composed of several RFID readers and antennas, as well as a 3D camera. AdvanRobot is able not only to replace human operators, but to dramatically increase the efficiency and accuracy in providing inventory, while also adding the capacity to produce store maps and product location. Some important benefit of the inventory capabilities of AdvanRobot are the reduction in stock-outs, which can cause a drop in sales and are the most important source of frustration for customers; the reduction of the number of items per reference maximizing the number of references per square meter; and reducing the cost of capital due to overstocking [1, 7]. Another important economic benefit expected from the inventorying and location capabilities of the robot is the ability to efficiently prepare

M. Morenza-Cinos (✉) · V. Casamayor-Pujol · J. Soler-Busquets · R. Pous

Universitat Pompeu Fabra, Roc Boronat 138, 08018 Barcelona, Spain

e-mail: marc.morenza@upf.edu

URL: <http://ubicalab.upf.edu>

V. Casamayor-Pujol

e-mail: victor.casamayor@upf.edu

URL: <http://ubicalab.upf.edu>

J. Soler-Busquets

e-mail: jordi.solerb@upf.edu

URL: <http://ubicalab.upf.edu>

R. Pous

e-mail: rafael.pous@upf.edu

URL: <http://ubicalab.upf.edu>

J.L. Sanz

Keonn Technologies S.L., Pere IV 78-84, 08005 Barcelona, Spain

e-mail: jlsanz@keonn.com

URL: <http://www.keonn.com>

R. Guzmán

Robotnik Automation S.L.L., Ciudad de Barcelona 3-A, 46988 Valencia, Spain

e-mail: rguzman@robotnik.es

URL: <http://www.robotnik.eu>

on-line orders from the closest store to the customer, allowing retailers to compete with the likes of Amazon (a.k.a. omnichannel retail). Additionally, the robot enables to: produce a 3D model of the store; detect misplaced items; and assist customers and staff in finding products (wayfinding).

Keywords Professional service robots · Inventory robots · Autonomous robots · RFID · ROS

1 Introduction

In this chapter a solution for smart retail that combines robotics and Radio Frequency IDentification (RFID) technology is presented.

Traditional retail (a.k.a. “brick-and-mortar” retail) is facing fierce competition from on-line retail. While traditional retail still keeps some advantages (e.g. physical contact with the products or immediate fulfillment), on-line retail continues to offer more and more advantages that are increasingly appealing to customers (e.g. easy to find products, no stock outs, in depth information, recommendations, user opinions, social networks integration, etc.) Also, on-line retail has at its disposal a wealth of data about its customers’ clickstream that can be leveraged through sophisticated analysis to offer personalized and targeted websites against which a generic “one model fits all” physical stores cannot compete. As a result, on-line retail is growing with double digits, while many retailers continue to close physical stores.

RFID offers an opportunity for traditional retailers to fight back. If every product in the retail store is tagged with RFID, it is given an Electronic Product Code (EPC), which is universally unique for each item. By placing RFID equipment at the store every relevant event of the product can be detected. Many leading retailers such as Kohl’s, Decathlon, Inditex or Marks and Spencers have already deployed RFID technology in their stores.

The most obvious, and most common, application of RFID in the store is for inventory. Most commonly, RFID-based inventories are done by using handheld RFID readers, that store associates use to scan every shelf, rack and fixture in the store. A typical fashion store in a typical shopping mall of about 1.000 m^2 , and about 10.000 items can be completely inventoried by a single associate in under 60 minutes. The same process using barcode technology would typically require a team of 3–5 persons working for one or two full days (double or triple counts are typically necessary in this case to reach an acceptable accuracy).

However, although the theoretical accuracy of RFID inventory using handheld readers is above 99%, retailers using this method report actual accuracies of between 80 and 90%. The difference lies in human errors. Inventory taking is a very tedious process, and the staff doing it frequently forget a shelf, an aisle, or an entire section. The layout of a retail store is typically not regular, and it is very easy, especially in larger stores, for associates to get confused and believe that they have already scanned a part of the store when they in fact have not. Also, the repetitive movements involved

in scanning the store have been linked to injuries, raising an issue of health in the work place.

Whenever humans are faced with tedious repetitive physical tasks, robots are the ideal candidates to overtake them, especially when there are health risks involved. Keonn Technologies, a manufacturer of RFID solutions for the retail sector, had the idea to combine a standard robotic base with an RFID payload composed of standard components to create a robotic inventory system for the retail store. The idea included some important insights on how to couple the navigation and the RFID systems to increase efficiency and the accuracy of the inventory process. In 2013 Keonn filed a patent and presented the first commercial prototype of an RFID inventory robot at the RFID Journal Live 2013 show, where it was selected as the winner of the “Coolest demo award” [15]. Since 2013 Keonn has taken the latest versions of AdvanRobot to the same show, where it has raised a lot of interest among retailers and in the RFID industry in general. During this time, Keonn, with the collaboration of the Ubiquitous Computing Applications Lab (UbiCA Lab) at Pompeu Fabra University, and the robotics company Robotnik has continuously improved the product, tested it extensively in large retail stores around the world, and established agreements with some of the most important players in the RFID for retail industry. AdvanRobot is the only robotic system for inventory designed by a multidisciplinary team of RFID specialists, robotics specialists, and academia. The resulting product, AdvanRobot, is now a part of Keonn’s portfolio of RFID solutions for retail.

AdvanRobot is able to inventory a very large store during the 10–12 h in which a store is normally closed. For the same job, at least 4 associates with RFID handheld readers would be required, making the Return on Investment (ROI) very high. Additionally, AdvanRobot never forgets a shelf, an aisle or a section, and the measured accuracy has always been above 99.5% of all tagged products in the store. In fact, AdvanRobot is the most accurate instrument to perform an inventory.

In addition, AdvanRobot is able to provide not only inventory but also the location of the products on the store layout and a 3D model of the store. This is considered of high value for retailers to detect misplaced items, to help customers find products and associates fulfill on-line orders.

In this chapter, the following topics are covered:

- First, we present a background section on RFID technology and its applications to retail.
- Second, AdvanRobot’s overview is given including its design and architecture, analyzing the specific navigation strategies for inventorying a store and finishing with the human-robot interaction.
- Third, a short introduction to AdvanRobot simulation is provided.
- Fourth, we describe the results of the tests carried out in actual retail environments.
- Fifth, ongoing developments are explained. We introduce a framework for the exploration and mapping of 3D environments. The ROS package is named *cam_exploration* and the code is publicly available at UbiCALab’s github account.¹

¹https://github.com/UbiCALab/cam_exploration.

- Sixth, we discuss the developments considered for future versions of AdvanRobot.

Due to the nature of the project some of the core packages of AdvanRobot are not publicly available. Nonetheless, a set of packages for a basic simulation are available.²

A video³ of AdvanRobot in a store introduces its operation and main features.

2 Background

2.1 RFID Technology

The central component of RFID technology is the RFID tag, composed of a chip mounted on a low cost antenna (usually made from etched aluminium). When an RFID reader sends an interrogating wave, all tags within the reach of the reader's antenna respond with a unique code, which the reader communicates through a wired or wireless network to an information system that makes use of this code for a given application. Figure 1 illustrates the different components of a typical RFID system.

In most cases, RFID tags are passive, and obtain the energy to respond by rectifying the interrogating signal wave. In some cases, besides a code, the RFID tag has a limited amount of user memory.

RFID technology has been around for decades now. However, it was not until 1999 when the AutoID Center at MIT redefined the frequency, the protocols and the code standards [16], that RFID started to become a widely adopted technology, especially in the retail sector. These standards were later acquired and are now managed by GS1,⁴ the global organization also managing commercial barcode standards.

As opposed to previous standards that use the low and high frequency bands (LF and HF), the new standard uses the ultra high frequency band (UHF) [6]. The band in Europe (ETSI standard) is from 865.6 to 867.6 MHz, and in the USA (FCC standard) from 902 to 920 MHz.

An RFID reader in the UHF band can read tags at a distance of up to 10 m as opposed to less than 2 m in the LF and HF bands. RFID antennas have beam widths normally between 30 and 90°. In a typical scenario the reader can identify hundreds, sometimes thousands of tags simultaneously, for which the Gen2 protocol [4] incorporates anticollision protocols allowing read rates of hundreds of tags per second.

The weight and battery constraints of handheld RFID readers limit the maximum power and in consequence the read range to between one and two meters. A robotic system, on the other hand carries a high capacity battery, which can operate several readers at full power, each connected to several antennas, each of them with a read

²<https://github.com/UbiCALab/advanrobot>.

³<https://youtu.be/V72Ep4s9T4o>.

⁴<http://www.gs1.org/>.

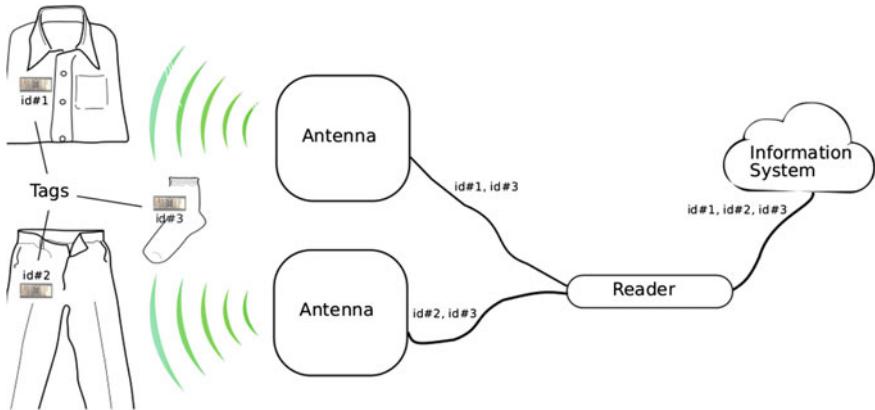


Fig. 1 Components of a typical RFID system. The Reader interrogates the environment by sending an RF signal through the Antennas. Tags, attached to products, reply with their unique identifier. The Reader communicates the data to an Information System for its exploitation

range of between 2–5 m. In consequence, a robotic RFID inventory system can have the equivalent reading capability of 10–20 handheld readers.

2.2 *Inventory Systems*

Due to errors, theft, misplacements and other reasons, actual inventories diverge significantly from theoretical inventories in the shop floor, typically by 10–20% [8]. Most retailers will use barcodes for inventorying, but inventories based on barcodes are expensive, disruptive, can only be done every few months, and their accuracy is typically no higher than 95%. This situation results in frequent stock outs, frustrated customers, expensive preventive overstocking, and in the impossibility to source on-line orders directly from the stores, which would allow retailers to effectively compete with online retailers [14]. In contrast, RFID-based inventories are much more affordable, non disruptive, and the accuracy is usually above 99%.

There are several options to inventory a store based on RFID technology. First, handheld RFID devices may be used to accurately take inventory of objects tagged with RFID tags. Second, ceiling mounted readers with fixed or steerable beam antennas can be used to inventory RFID-tagged objects. Third, smart shelves or fixtures, incorporating RFID antennas and readers can be used to continuously inventory the objects they contain, as long as they are tagged with RFID. And fourth, autonomous robots [13] or UAVs [17] can be used to inventory all objects in a space, also using RFID tagging.

Handheld readers cannot, by themselves, provide any information about the location of the objects within the space being inventoried, while the other three methods

Table 1 Comparison of RFID inventory methods

Method	Location accuracy	Inventory frequency	Fixed infrastructure	Hardware cost	Labor cost
Handheld reader	No location	Every few days or weeks	No	Very low	High
Ceiling mounted readers	2 m	Every few minutes	Yes	High	None
Smart shelves and fixtures	50 cm	Every few seconds	Yes	Very high	None
Autonomous robot	2 m	Every day	No	Low	Very low

can provide location with different degrees of accuracy. The first and fourth methods can provide frequent but non continuous inventory, while the second and third can provide quasi real time inventory (a.k.a. “near” time inventory). The first and fourth methods do not require any fixed infrastructure installation, and the second and third methods do. The four methods also differ in the cost of hardware and the cost of labor they require. Table 1 summarizes the above comparison.

3 AdvanRobot Overview

AdvanRobot is an autonomous mobile robot that takes inventory of RFID labeled products in large retail stores. Therefore, by using AdvanRobot, taking inventory becomes an automatized task. In addition, complementary features revealed after its initial concept, for instance the location of the RFID labeled products and the generation of 3D maps of the environment. This section is developed as follows: first, the AdvanRobot is described detailing its design and characteristics; second, a high-level overview of the system architecture is given; third, the navigation strategies for inventorying a store are defined; and finally, the human-robot interaction is detailed.

3.1 Design

AdvanRobot is designed in two main systems: the robotic base and the payload. Briefly, the robotic base is a ROS based autonomous mobile robotic base that is in charge of satisfying all the requirements that the payload needs for inventorying. It provides power, a safe and reliable navigation, and connectivity with the environment. The payload is the system in charge of performing the main task of the robot which is taking inventory.

In addition, it incorporates a web interface for the human robot interaction that can run in any web browser. This interface allows the interaction with the user from a very high-level simplifying all tasks and ensuring that everything runs as required. The Human-robot interface is detailed in Sect. 3.4.3.

3.1.1 Robotic Base

The robotic base is the model RB-1 manufactured by Robotnik.⁵ It is a circular base with differential wheels allowing an excellent maneuverability in narrow aisles since its turning radius is 0. Moreover, it has a load capacity of 50 kg and provides high stability and damping. However, note that the RB-1 base used includes ad-hoc modifications. The base consists of three subsystems: the traction subsystem; the brain; and the power subsystem.

The traction subsystem includes two motorized and encoded wheels powered by servomotors, three omni-directional wheels, and two dampers for stability and overcoming floor irregularities. Due to its traction configuration it has differential drive capabilities. In addition, it has an emergency push (e-stop) button that cuts the servomotors power and immediately stops the robot.

Secondly, the brain, composed by the computer and electronics subsystem is in charge of controlling and connecting all the robot parts and providing all the intelligence required. Its main devices are the computer, a router that provides an access point for external connections and the sensors. The only embedded sensor is an IMU with two gyroscopes, an accelerometer and a compass. The RB-1 base also uses peripheral sensors: an optical (RGB) and depth (D) camera (RGBD camera) placed on top of the payload in order to detect obstacles, and a laser range finder. Finally it is also prepared for the installation of sonars in order to avoid those obstacles that can not be detected by the laser range finder or the RGBD camera such as mirrors, black surfaces and highly translucent materials which can be present in target environments. All the peripheral sensors connectors are easily accessible for their connection and disconnection.

Finally, the power subsystem consists of a lithium iron phosphate battery that provides more than 11 h of autonomy. It also includes a battery management system (BMS) which controls the charging and discharging of the battery, and the electronics for recharging on a charging station.

⁵http://wiki.ros.org/Robots/RB-1_BASE.

3.1.2 RFID Payload

The RFID payload is the system in charge of taking inventory. It consists of three main parts:

First, AdvanRobot is equipped with 3 RFID readers,⁶ which control 4 antennas each. However the robot can work with different configurations, using 1–3 readers combined with RFID multiplexers to control all the antennas.

Second, AdvanRobot mounts 6 RFID antennas per side, summing up a total of 12 RFID antennas.⁷ The antennas are placed side by side in a way such that their reading areas overlap. In this fashion, there is a minimization of blind spots in what regards RFID readings. Hence, there is a degree of redundancy in the RFID subsystem. This is paramount in order to ensure a critical inventory accuracy. In the configuration shown in Fig. 2, AdvanRobot is equipped with the RFID antennas aforementioned. However, other types of antennas can be used in order to achieve different reading behaviors and scanning patterns.

Last, the structural subsystem, with the main aim of providing a physical support to the former: the RFID antennas and the RFID readers, in addition, the structural subsystem is foldable. AdvanRobot has been designed to read tags up to 2.75 m. As a result its height is slightly above 2 m. Therefore, the possibility of being folded implies the ability to traverse any door at the same time as being high enough to read all the products in a store.

Besides, the RGBD camera is linked to the top of the RFID payload. The reason for that is the maximization of the usable field of view of the camera. In what regards obstacle avoidance, this is crucial. For a safe navigation, AdvanRobot should detect any obstacle up to its height. This is better achieved observing the environment from the uppermost location of the payload.

3.1.3 Interconnection

Both parts are interconnected by two USB ports for the RGBD cameras; an RJ45 port to interface the RFID system; and a connection for power supply. These connections are accessible through a small door in a side of the payload making the assembly and disassembly a very easy procedure.

3.1.4 Comparison with Other RFID Robots

A comparative analysis is done based on the information available regarding other commercial RFID-equipped robots. At the moment, such information is not extensive. The comparison highlights the features that are explicitly different between the involved robots, which to our knowledge are:

⁶<http://keonn.com/rfid-components/readers/advanreader-150.html>.

⁷<http://keonn.com/rfid-components/antennas/advantenna-p22.html>.



Fig. 2 AdvanRobot in operation in a store. At the *bottom*, the lower circular part is the robotic base. On *top* of it, the RFID system, which is foldable. Finally, at the *top* of the RFID system the RGBD camera for obstacle detection

- Tory⁸ manufactured by *MetraLabs*
- StockBot⁹ manufactured by *PAL Robotics*

From a structural point of view, both, AdvanRobot and Tory have a circular footprint of 25 cm radius, however, StockBot's footprint is not circular and its equivalent footprint radius is 35 cm. This directly impacts the minimum size of the aisles where the robot can navigate, limiting its versatility in some environments. Regarding height, AdvanRobot is above the others, but it can be folded.

With respect to the battery autonomy, Tory states providing 14 h, Stockbot 8 h and AdvanRobot 11 h. In addition, Advanrobot is the one that recharges the battery in a shorter amount of time, it requires 2 h while the StockBot needs 4 h and Tory from 3 to 6 h for a complete charge. The robots operational availability is defined as the ratio between the time the robot is working and the total robot time (working plus charging time). Therefore, AdvanRobot accounts for an operational availability of 84.6%, Tory of 70% and Stockbot of 66,6%. For instance, AdvanRobot would be operative for 84.6 h over a complete period of 100 h. The remaining 15.4 h would be used for battery charging.

Focusing on the RFID system, StockBot has 8 integrated RFID antennas, Tory only mentions that it has integrated RFID antennas. AdvanRobot uses 12 antennas which

⁸<http://www.metralabs.com/en/shopping-rfid-robot>.

⁹<http://pal-robotics.com/ca/products/stockbot>.

Table 2 Summary of the comparative analysis of commercial inventory robots

	AdvanRobot	Tory	StockBot
Height (cm)	208	150	190
Equivalent footprint radius (cm)	25	25	35
Battery autonomy (h)	11	14	8
Charging time (h)	2	6	4
Operational availability (%)	84.6	70.0	66.6
RFID system	1–3 readers 12 antennas	Not specified	2 readers 8 antennas

characteristics can be selected among different options and from 1 to 3 readers upon application and user request. Thus, AdvanRobot provides an excellent versatility to adapt the RFID system to the environment.

Finally, to the best of our knowledge and from an operational point of view, AdvanRobot is the only one that is prepared to work by zones inside a shop floor. Such feature is explained in Sect. 3.4. Table 2 summarizes the analysis.

3.2 Architecture

The high-level architecture of the robot consists of 5 main blocks: User; Interface; Task manager; Navigation; and RFID. Figure 3 shows a schematic of the architecture. The Interface allows the communication between the user and the robot, it is the main component of the human-robot interaction. The *Task manager* basically translates the user requests into a set of actions. The Navigation block receives the actions and transforms them in commands for the movement of the robot. At the same time, the RFID system reads surrounding tags. The Navigation and RFID blocks interact in order to succeed in the selected task. The details of Navigation and RFID are explained in Sect. 3.3. Besides, the architecture follows the modularity of the robot's

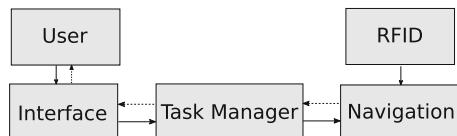


Fig. 3 System's high-level architecture. *Solid arrows* indicate control while *dashed arrows* indicate control feedback. RFID plays a critical role in Navigation decisions in order to accomplish accuracy and time constraints

design. The Navigation block corresponds to the autonomous base control and the RFID block to the RFID payload.

3.2.1 Task Manager

The Task Manager is implemented in a node named *task_manager*. It is the middleware that translates high-level user orders to lower-level control commands. It has been created in order to dispatch and monitor the tasks that the robot has to perform from a user perspective, meaning that it operates at a higher-level than the Navigation and RFID blocks.

task_manager communicates with the interface via ROS Services,¹⁰ in which the parameters of the service are the selected task and its options. Shortly, the node performs two main operations.

First it keeps the state for the robot, which is communicated to the user. Hence, the user knows the selected action status and progress. Additionally, the state prevents any interaction through the interface that could interfere with the current task. This state assignation allows the robot to work as a simplified finite state machine.

Second, it executes the selected task actions, using the ROS actionlib stack.¹¹ By means of actionlib, the node monitors the task, and if required it can also preempt the actions.

Also, *task_manager* is subscribed to other nodes that publish the state of relevant parts of the robot. This has two main benefits. On one hand, ROS message passing facilitates monitoring all the relevant information, from the state of the RFID readers to the temperature of the motors. On the other hand, the *task_manager* is the node that centralizes all the information and presents it to the user in a comprehensible way via the interface.

3.3 Navigation

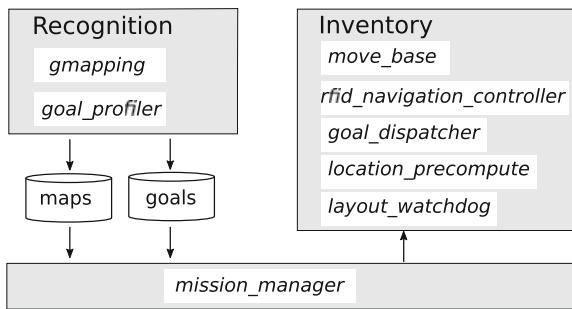
AdvanRobot uses the ROS navigation stack¹² for a safe navigation in retail environments. AdvanRobot is configured to navigate in any wheelchair accessible retail space which are those comprised by aisles equal or greater than 70 cm [5, 9]. It uses a range laser finder for simultaneous localization and mapping, widely known as SLAM [2] and an additional RGBD camera for obstacle detection in 3 dimensions. It is sonar-ready for the detection of mirrors and materials not reflective to light-spectrum. Before deploying AdvanRobot, an environment survey identifies potential risks for navigation and determines the need to use sonars, which can be plugged and played to clear the identified risks.

¹⁰<http://wiki.ros.org/Services>.

¹¹<http://wiki.ros.org/actionlib>.

¹²<http://wiki.ros.org/navigation>.

Fig. 4 Building blocks of AdvanRobot's navigation. In Italic, the names of the ROS nodes involved in each of the sub-blocks



The navigation consists of a preparatory human assisted stage and a fully autonomous stage. The first is needed to get a baseline of the environment and it is called *Recognition stage*. During the *Recognition stage* AdvanRobot generates a map and records key spots for later navigation. Once the *Recognition stage* is completed successfully AdvanRobot is ready for the *Inventory stage* when the actual autonomous inventory taking is performed. In both stages RFID observations are used as inputs to support the optimal performance of AdvanRobot. Figure 4 shows schematically the Navigation parts explained next.

3.3.1 Recognition Stage

The aim of the recognition stage is providing a guided observation of the environment to AdvanRobot. By doing so, AdvanRobot learns the map of the zone intended for inventory and, by listening to the RFID readings, records the key spots where products are present. In practice, an operator brings AdvanRobot to a zone's initial spot and using a remote control moves AdvanRobot close to the products to inventory. At the same time, a map is generated using ROS gmapping¹³ and key spots are recorded by a purpose-developed ROS package: the *goal_profiler*.

3.3.2 Inventory Stage

AdvanRobot performs inventory taking during the *Inventory stage*. Simultaneously, it does a pre-computation of RFID reads as a preparation for the latter offline location computation. The *Inventory stage* is triggered and controlled by the *mission_manager* node.

Following the trigger of an inventory, the key spots recorded during the *Recognition stage* start to be dispatched in the form of navigation goals. In order to optimally dispatch navigation goals during the inventory stage, a ROS node, the *goal_dispatcher*, monitors the progress of navigation and rearranges the goals online.

¹³<http://wiki.ros.org/gmapping>.

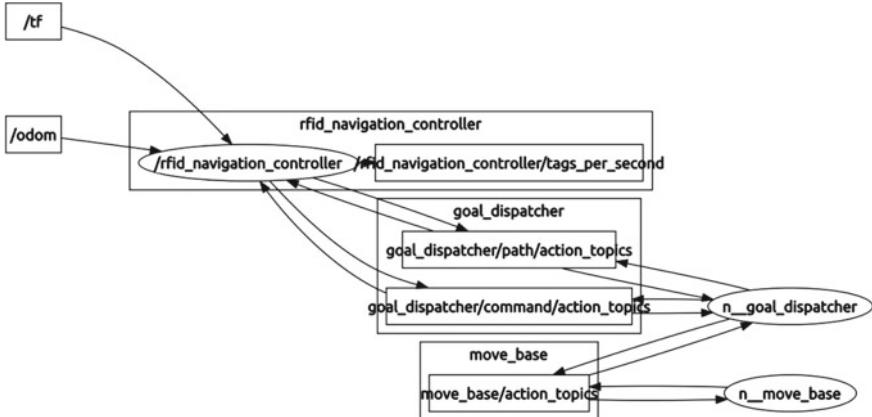


Fig. 5 Rqt_diagram of the navigation controller node

For instance, if a key spot is visited unexpectedly due to path re-planning its corresponding navigation goals are cleared and not dispatched again.

Given a navigation goal, linear and angular velocities sent to the motors controller are commanded by the *rfid_navigation_controller* node, which monitors the progress of RFID reads in order to compute the following velocities. The *navigation_control* node modulates the output of *move_base*¹⁴ in order to get the best inventory accuracy in the least time possible. For instance, if AdvanRobot moves at its maximum speed, which is 0.4 m/s, in an environment with a high density of RFID labeled products, the RFID system has no time to identify all the products. Thus, the navigation needs an added control layer that takes into account the progress of RFID reads. Otherwise, inventory accuracy requirements are not met. Figure 5 shows the rqt diagram that relates the *navigation_control* node to the the *move_base* node. In addition, the node *location_precompute* matches RFID reads to the corresponding identification antenna pose using ROS transforms¹⁵ lookups. Each RFID read is stored along with the antenna pose at the time of the identification for a posterior location computation. The computation of location is an ongoing development explained in Sect. 6.2

3.3.3 Integration

Linking properly the output of the *Recognition stage* to the subsequent *Inventory stages* is addressed by the *mission_manager* node. A key system design feature is that AdvanRobot follows a divide and conquer strategy to conduct inventory missions. An important learning from field experience is that it is preferable to define zones of less than 1000 m² instead of working with a single larger zone. This is explained by

¹⁴http://wiki.ros.org/move_base.

¹⁵<http://wiki.ros.org/tf>.

three main reasons. The first, the ease of operation by the user in case an inventory of a specific zone or set of zones is needed. This has been validated with users during on-site pilots. The second, a less demanding computational cost. Working with big and precise maps implies a high computational cost. The third, a convenient modularity facing considerable layout changes and the consequent need of a re-recognition. Hence, AdvanRobot is designed to work by zones, following a divide and conquer strategy.

Given a set of zones, any combination is eligible and is defined as an *Inventory mission*. An *Inventory mission* comprises a set of consecutive *Inventory stages*. In this way, the user can select the zones to inventory as needed. Working with a set of zones requires a proper management of the *Recognition stage* output, which are maps and goals. For each zone, a pair map and set of goals is kept. Thus, an *Inventory mission* requires dispatching the proper map and goals in the appropriate order and timing to the navigation layer. The *mission_manager* is the ROS node that performs the tasks of triggering and monitoring *Inventory stages* according to a defined *Inventory mission*.

The divide and conquer strategy is implemented placing a zone identifier at the beginning of each defined zone. The zone identifier is the reference for AdvanRobot to know the actual zone. At the moment, the zone identifier is a QR code which is detected by the RGBD camera using the ROS package *ar_track_alvar*.¹⁶ The automatic identification of zones enables the user to perform the recognition stage and afterwards launch an inventory mission without assistance. As well, zone identifiers are used by AdvanRobot to perform map transitions autonomously, commanded by the *mission_manager*, since they define the relations between zones. Furthermore, QR codes can be used as a support to AdvanRobot's location recovery and correction, which is discussed in Sect. 3.3.4.

An essential feature for a user is knowing when the *Recognition stage* needs to be rerun. By using ROS navigation stack properly, AdvanRobot is able to adapt to changes in the layout. However, if layout changes are significant, AdvanRobot may not be able to output a reliable localization and *Inventory missions* can fail. With the purpose of computing the need of rerunning a *Recognition stage* an algorithm is run by the node *layout_watchdog*. The algorithm uses as inputs mainly but not only the success and time to reach goals and the reliability of the localization during the mission.

3.3.4 Challenges

Exploration. The main challenge of the *Recognition stage* is suppressing the need of human assistance. For that, exploration has been considered and preliminary tests conducted. However, in retail environments, which are generally a big extension of interconnected aisles, the time to complete an unassisted exploration is prohibitive. Compared to an unassisted exploration, the actual approach has two main advantages:

¹⁶http://wiki.ros.org/ar_track_alvar.

optimizing the time it takes to recognize a zone; and empowering a user with an easy way to define the interesting zones for inventory. The latter, if doing unassisted exploration would require a posterior manual intervention or the addition of beacons for the robot to identify the interest zones. The ideal case would be that of a non human assisted exploration, but assisted by beacons or other technologies. Possible means for assisting explorations without human intervention include those discussed next for supporting localization.

At the moment, exploration is under testing, see Sect. 6.1. The exploration is being developed with the combined aim of granting AdvanRobot enhanced autonomy and producing 3D maps.

Localization Robustness. AdvanRobot requires a good accuracy in localization in both the *Recognition stage* and the *Inventory stage* in order to complete its tasks. During the *Recognition stage* AdvanRobot does not know the map of the environment, hence, it is executing a SLAM algorithm. It has been noticed that in very regular environments and large open spaces the localization of AdvanRobot is not reliable enough to generate faithful maps. Moreover, at the beginning of the *Inventory stage*, AdvanRobot needs to deal with the problem known as kidnapping [3]. To cope with this issue landmarks can act as absolute positioning references. Accordingly, the actual implementation makes use of QR codes. However, the detection of QR codes relies on a direct line of sight and lighting. Alternative means to support localization include laser reflectors, Bluetooth beacons and Battery-Assisted Passive (BAP) RFID tags.

Robustness to layout changes. Significant layout changes (or the accumulation of minor layout changes) can impact significantly the performance of AdvanRobot. In front of such changes AdvanRobot may not be able to reliably localize itself in the environment and fail to complete a mission. When this happens, not only the mission failure consequences have to be assumed but also the *Recognition stage* needs to be rerun. An interesting challenge is granting AdvanRobot with the capability of modifying the maps and goals of a zone at the same time it is running the *Inventory stage*. Hence, after several inventory iterations of a zone there would be no divergence from the original observations to the actual layout, which is the case at the moment. In practice, at every *Inventory stage*, the zone's *Recognition stage* observations would be updated and the impact of cumulative layout changes minimized. This would be equivalent to an assisted exploration, being the assistance the previous observations of the zone. In this way, there would only exist the need for a very first human assisted *Recognition stage*.

Inventory and Location Navigation Strategies. At the moment, the navigation is optimized for the compromise between time and inventory accuracy. However, a precise RFID location requires constraints to be met in terms of navigation [11]. Combining inventory and location constraints in a single navigation strategy is one of the main challenges to navigation control. After, it follows the addition of constraints for a complete 3D mapping of the environment. Combining optimally the

constraints for inventory, location and 3D mapping would optimize the valuable outputs of AdvanRobot and, at the same time, minimize the time invested in the commission.

3.4 Human-Robot Interaction

Human-AdvanRobot interaction mainly consists of two operational procedures that simplify the user experience and minimize human errors. Both operational procedures are guided and executed by means of a control interface described next.

The first procedure empowers the user to launch a *Recognition stage* in order to create a map of the environment and get a set of indicative goals to follow when doing inventory. The second procedure lets a user launch the inventory of a sequence of selected zones, called *Inventory mission*. The second procedure can be scheduled and managed remotely.

The specific challenges and specifications related to human-robot interface are explained. AdvanRobot is a system that is specially suitable for large stores. Nevertheless, taking inventory of the whole store in a single mission is not always possible due to time constraints. Moreover, the user might request to take the inventory of only a collection of specific products. To cope with this, two key aspects of AdvanRobot operations are introduced:

- The division of the shop floor.

The shop floor is separated in zones complying with the following:

- Zones should contain a family or a set of related families of products.
- Zones should be between 750 m² and 1500 m².
- Zones should encompass easily identifiable architectonic features. For instance, it is no recommendable defining a zone as an island of hangers in the middle of a store. This is due to the problems that can arise in referencing the zone and the robot localization robustness.

Therefore, for each of the zones the robot keeps a separate map interlinked with the other zones' maps. The reason for this has been discussed previously in Sect. 3.3.

- Zones are identified using visual cues, in this case QR codes.

The linking and identification of defined zones is done using QR codes placed at the start and at the end of each zone. Noteworthy, the end of a zone is always the beginning of the following zone. Consequently, all the zones are interlinked and any sequence of zones can be selected for inventorying.

This two key aspects allow the user to easily select zones for inventorying and recognition. If the layout of the shop floor has changed considerably and the re-recognition of a zone is required, only the specific zone will need to be re-recognized saving AdvanRobot time as opposed to having to re-recognize the whole area (the sum of all the individual zones). And the user can identify such zone easily since it is marked at its beginning and end by a QR code.

3.4.1 Launching the *Recognition Stage*

In order to launch the *Recognition stage* the user sequence of steps to follow is:

- Place the robot in front of the starting QR code of the zone. Doing so, AdvanRobot recognizes which zone is about to be recognized and informs the user on the interface for confirmation.
- By means of the human-robot interface the recognition is launched pressing the button *Start Recognition*.
- Guide the robot through the zone's interest spots, those intended for inventorying. While it is being guided, AdvanRobot records key spots where it is reading RFID tags. Later, the key spots are used to guide the inventory mission.
- When the RGBD camera detects the final QR of the zone, the interface pops up the options *Finish Recognition* and *Continue Recognition*. The first allows the user to end the process, the second is used to resume the guiding process in case, even the final QR has been detected, it is not yet over.
- If the option *Finish Recognition* is pressed, the map and key spots are stored and the *Recognition stage* processes stopped.

3.4.2 Launching an *Inventory Mission*

There are two procedures to start an *Inventory mission*. The first, which minimizes the user intervention, starts and ends AdvanRobot at its charging station. Given that the human-robot interface is a web application it can be accessed remotely. This empowers the user to launch *Inventory missions* programatically and remotely. The second is used when the user requires starting an inventory at a specific zone:

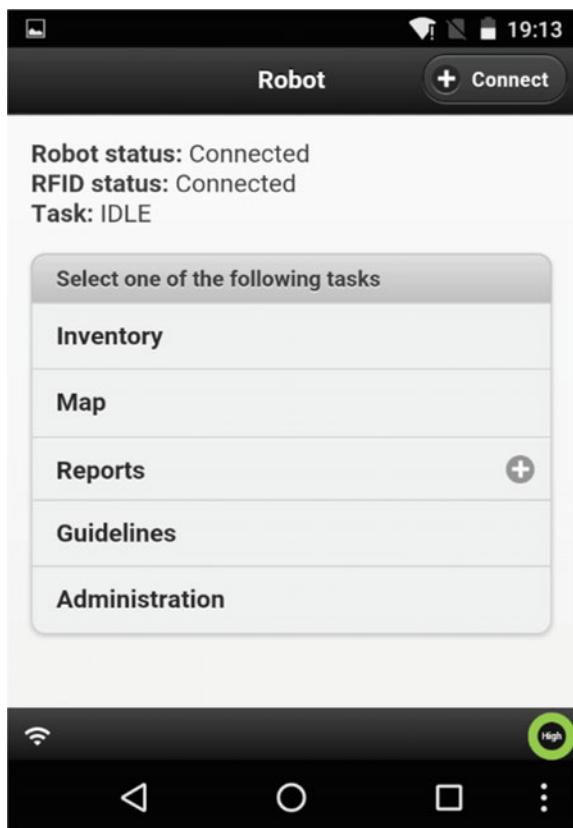
- Place the robot in front of the starting QR code of the first zone.
- Select the list of zones that AdvanRobot that comprise the *Inventory mission*.
- By means of the human-robot interface the Inventory mission is launched pressing the button *Start Inventory*.
- From this moment AdvanRobot is completely autonomous and its status and progress can be monitored on the human-robot interface.

3.4.3 Human-Robot Interface

The robot's interface allows the user to interact with the robot in an intuitive and painless manner and it provides feedback of its status and progress. Figure 6 shows a snapshot of the human-robot interface.

The interface guides the user along a sequence of steps that ensure the robot is prepared for the selected task. For instance, it gives guidelines to the user about the proper placement of the robot in front of a QR code, or it notifies the user to pull the e-stop button when it is needed.

Fig. 6 AdvanRobot interface



Once all the steps have been successfully completed by the user following the interface guidelines, the specific selected task and its parameters are communicated to the *task_manager* node (see Sect. 3.2.1) by means of ROS Services. During the task commission, the interface provides feedback by showing the progress of the task to the user. For instance, during the *Recognition stage* the interface shows the map together with the key spots that are being recorded; and when the robot is performing an *Inventory mission* it shows the progress of RFID readings and the progress of the mission itself.

In addition, the interface includes other relevant indicators. For instance, the snapshot of the interface, shown in Fig. 6, indicates that the robot is fully charged with the green circle on the bottom right corner and properly connected to internet with the white symbol on the bottom left corner. Also, it shows that the robot and the RFID systems are connected, and the robot status is IDLE, hence, any task can be triggered by the user.

The human-robot interface is a web application built using *HTML5* and *Javascript*. The communication with the ROS Master is accomplished using *rosbridge_suite*,¹⁷ a meta-package that provides the definition and implementation of a protocol for ROS interaction with non-ROS programs. *Rosbridge* is implemented using WebSocket as a transport layer and provides an API which uses JSON for data interchange. Finally, it also uses the ROS package *web_video_server*¹⁸ for streaming the video of the RGBD camera.

3.4.4 Challenges

In a large retail store there are WIFI blind spots. Therefore, the connectivity with the robot through an infrastructure network can be lost unexpectedly. In case a user needs to interface the robot and the infrastructure connection is not available, without a backup connection the interaction becomes impossible.

In order to guarantee a responsive connection at any store location, the robot includes two wireless links: one as client and the other as an access point. Usually, the robot is linked to the infrastructure network as a client, enabling its remote access and control. Moreover, the robot periodically uploads to the cloud relevant mission and status data, keeping a historical log that can be reviewed even when the robot is not online. In case the infrastructure network is not available in order to control or to know the robot status, AdvanRobot can be interfaced by means of its access point. As opposed to the infrastructure connection, this is available as long as the user stays within the robot's WIFI range. The roaming between links is performed automatically by the robot's interface, giving always priority to the robot's access point.

Hence, there is a valuable degree of redundancy in the robot's connectivity.

4 Simulation

The end-to-end simulation of the system has been set up using ROS and Gazebo. Simulation has been used for the validation of functionality in terms of navigation, operations and human-robot interaction. Yet, a realistic simulation of RFID is not available given its physical model is complex. RFID electromagnetic propagation suffers strongly from multipath effect, which means the RFID signal rebounds and is attenuated multiple times depending on the characteristics of the scenario. Modeling such behavior means taking into account each and every item and its characteristics within the reach of every single electromagnetic wave. At RFID frequencies not even raytracing produces satisfactory results. Only a full finite-element simulation of the entire environment, prohibitive from a computational cost point of view, would output reasonable simulation results.

¹⁷http://wiki.ros.org/rosbridge_suite.

¹⁸http://wiki.ros.org/web_video_server.

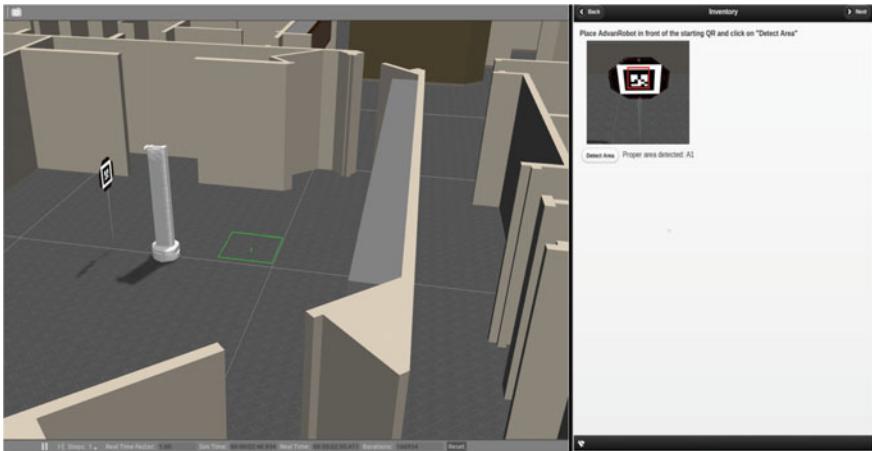


Fig. 7 Simulation of AdvanRobot at the moment of initiating an Inventory mission. On the *left*, a view of AdvanRobot standing in front of an initial QR. On the *right*, the corresponding view on the interface

Even though an RFID sensor is included for simulation in Gazebo, it is not implemented considering all the RFID simulation complexities. In conclusion, it is not possible with the available tools to simulate a realistic target scenario for the use case.

Accordingly, the simulation engine used for the RFID reads is not a physical but a probabilistic one. In this manner, only the throughput of RFID reads can be set to behave analogous to reality, which works for the validation of the coding of navigation strategies but not for the validation of their convenience regarding inventory accuracy and time. Hence, the validation and tuning of navigation strategies for an optimal compromise between inventory accuracy and time can only be performed in actual physical scenarios.

A set of packages for a basic simulation of the system can be found in the *UbiCALab* github repository¹⁹ and a snapshot of the simulation is shown in Fig. 7.

5 Experimental Results

AdvanRobot has been tested periodically in retail environments in every design iteration. The last version of AdvanRobot has been validated for a duration of 2 months at a retailer's facility as the preparation for a subsequent pilot. The validation targeted AdvanRobot's navigation on the shop floor; AdvanRobot's RFID identification accuracy; and the operation by store associates after a training.

¹⁹<https://github.com/UbiCALab/advanrobot>.

Table 3 Maximum and minimum inventory times of the complete store throughout several iterations

Minimum time	23:41:33
Distance (m)	4,485
Effective speed (m/s)	0.052
Maximum time	31:50:51
Distance (m)	5,422
Effective speed (m/s)	0.047

5.1 Navigation Validation

The navigation in a retail environment is not trivial due to the characteristics of shop floors. The main concerns at start have been the validity of the layout for a robust localization of the robot; floor materials and discontinuities; the ability to plan optimally paths; the effective speed of an inventory given the intricate configuration of aisles; and the effectiveness and negotiation of navigation commands. AdvanRobot has been in operation 8 h per night for 40 nights in a 7500 m² store. Table 3 shows the maximum and minimum inventory times of the complete store throughout the iterations. The effective speed of AdvanRobot at inventorying is roughly 0.05 m/s, which is satisfactory but still the main figure to improve. The effective speed is compromised by the constraints of the RFID system and, at the same time, by the complexity of the layout for navigation.

None of the initial concerns were found critical in completing inventory missions. However, the robustness of localization is sometimes compromised by the lack of structural features to robot's observations reach. This is not a matter of installing more powerful sensors for location - a longer range laser - since it is usual for the robot on the shop floor to end up surrounded by expositor furniture. While the impact of this has not been critical to the day, it is considered a key aspect to improve. There are two main approaches to tackle localization robustness. On one hand, improving localization algorithms. On the other hand, providing localization algorithms with additional observations of the environment.

5.2 RFID Identification Accuracy

Measuring inventory accuracy requires a baseline for comparison. The ideal case is that of manually counting each RFID label, which is known in retail as fiscal inventory. This seldom happens throughout a year given the workforce needed to count up to hundreds of thousands of items, a usual amount in a big retail mall. Consequently, a less demanding baseline in terms of man-hours is used. Currently, retailers use RFID handheld inventory devices for stock counting. Therefore, one of the references to compute the robot's accuracy is the output of handheld devices. Moreover, retailers generally keep an inventory record which is an estimation based

Table 4 AdvanRobot and handheld comparative accuracy

Product type	AdvanRobot's accuracy	Handheld accuracy (%)	Amount of RFID labels
Men's wear	99.80	77.90	39,671
Women's wear	99.90	44.10	22,277
Women's underwear	99.54	72.77	8,778
Men's underwear	99.43	88.91	1,055
Jeans	99.85	99.51	2,027

Table 5 AdvanRobot accuracy comparing to estimated inventory record

Estimated inventory count	Robot count matching	Robot count in excess	Accuracy (%)
209,465	162,335	12,025	78.7

on items inputs and outputs but not on actual counts of stock on the shop floor. Even such kind of estimated inventory records diverge from reality quickly over time, they are still a good baseline for an arbitrated accuracy comparison. In conclusion, the two baselines that are used to measure the robot's accuracy are handheld devices and estimated inventory records.

In order to measure the accuracy using the output of handheld devices, the baseline is computed as the sum set of items identified by AdvanRobot and by the handheld in a given zone. Table 4 shows results for a set of tests at selected zones. It is noticeable a higher AdvanRobot accuracy in all the compared cases. Interestingly, in the case of *Women's wear* the handheld accuracy is significantly lower. Likely, the explanation for that is human error during handheld inventory taking. One of the main advantages of using a robot for inventory taking is preventing such oversights.

Besides, AdvanRobot's accuracy was measured using an estimated inventory record of the whole retail store as baseline. In this case, the baseline itself is less accurate, which has an impact on the robot's measured accuracy. One of the main reasons are items that are reported to be at the shop floor but are actually at back stores not visited by the robot. Note that estimated inventory records report stock keeping units (SKU's) and quantities (count) instead of unique item identifiers. This means that the comparison is not direct. Table 5 shows the accuracy of AdvanRobot measured at a store with more than 200.000 RFID labeled items. The column *Robot count matching* shows the count of product SKU's identified by the robot matching the estimated inventory count. The column *Robot count in excess* shows the amount of references identified by the robot with a count higher than estimated. The accuracy measure is considered satisfactory by the retailer given the nature of the baseline, which is itself divergent from actual stock. Interestingly, the data set showed an excess of 19.938 references identified by the robot and not present in the estimated

inventory record. This means that the robot identified product references that were not known to the estimated inventory record for some unidentified reason.

In conclusion, the validation of RFID identification accuracy results successful in all the cases. Noticeably, AdvanRobot always outputs an accuracy above 99.5%. This is even more remarkable given the environment is highly dense in terms of RFID labeled products. In the exposed case there were over 200.000 RFID labeled items on a 7500 m² surface.

5.3 *Operation by Store Associates*

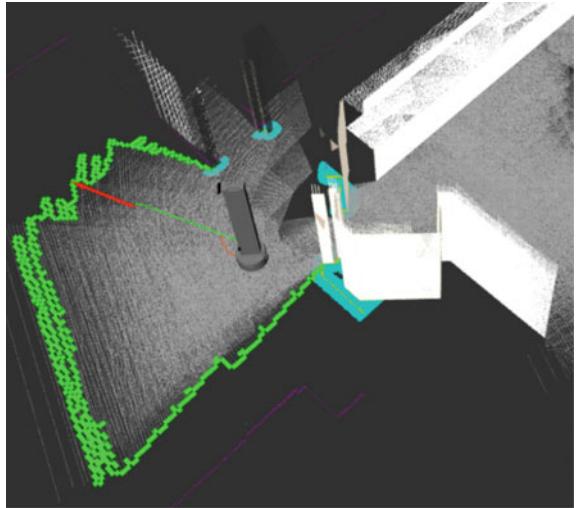
The suitability of the operational design and the usability of the human-robot interface is of utmost importance. For that, AdvanRobot was handed to store associates for its use for a month during the pilot's preparation after a training. On the operational side it is noteworthy the users flexibility requirements. Each user has its own specifications depending on the details of the use case. For instance, the use of zones has proven useful in some cases while it was not necessary in others. A convenient approach is using a modular design ready for a quick customization. Since AdvanRobot is designed operationally based on a divide and conquer strategy, adjustments on the field are easy and quick to apply. A more complex question is human-robot interaction since potential operators include non-skilled associates. For that, the design of a user-friendly mobile app for AdvanRobot's control and monitoring is crucial. We have noticed an initial steep learning curve mainly due to the lack of familiarity with advanced technology and a consequent low acceptance. Thus, while the interface has not presented remarkable issues, a good communication strategy to aid in the acceptance of a robotic solution on a shop floor is key for the success in its deployment.

6 Ongoing Developments

6.1 *Exploration for 3D Mapping*

Building a 3D model of an environment can be done with robots equipped with RGBD cameras. Combined with the ability to locate products, it opens the door to a range of interesting possibilities such as measuring the impact of the placement of products, furniture and their combination in the sales; building a virtual store with the aim to link the offline to the online world; or verifying the layout, planogram and signage of a store. For this reasons, the generation of 3D maps is a use case of interest to potential users.

Fig. 8 Gazebo simulation of an exploration using the exploration framework. The green cells represent the frontier between the known and the unknown space. The red arrow points at the exploration goal selected



Currently, 3D mapping is working by means of the ROS package *rtabmap_ros*²⁰ and the exploration for 3D mapping has been validated in simulation. Given the amount of factors that influence the output of an exploration, a ROS exploration framework has been developed in order to provide a fast and easy way of measuring the performance of different exploration approaches.

With the aim of generating a complete 3D map of the environment it is required sweeping all the space of interest. For that, a proper exploration strategy has to be applied. By using a 3 dimensional exploration, it is assured the completeness of observations of the space needed to generate the 3D map. Note that while the exploration considers the 3 dimensions of the space, the navigation is limited to 2 dimensions.

A number of mature techniques exist tackling the robot exploration problem. One family is the frontier-based exploration which has long been exploited since its introduction in [18]. Frontiers are regions on the boundary between open space and unexplored space. By selecting a certain frontier as exploration target, the complete environment exploration is ensured. The exploration framework is intended for the frontier-based exploration technique (Fig. 8).

The setup for the simulation using the developed framework includes two RGBD cameras. The second camera introduces an extra source of point clouds from a different perspective. Hence, objects are scanned from more than a single pose leveraging the 3D models. Furthermore, extra cameras are also beneficial for navigation purposes as the consequent increment of the field of view adds valuable redundancy to obstacle detection. The addition of more cameras is considered as they supply extra observation sources, beneficial for the completion and resolution of the 3D model.

²⁰http://wiki.ros.org/rtabmap_ros.

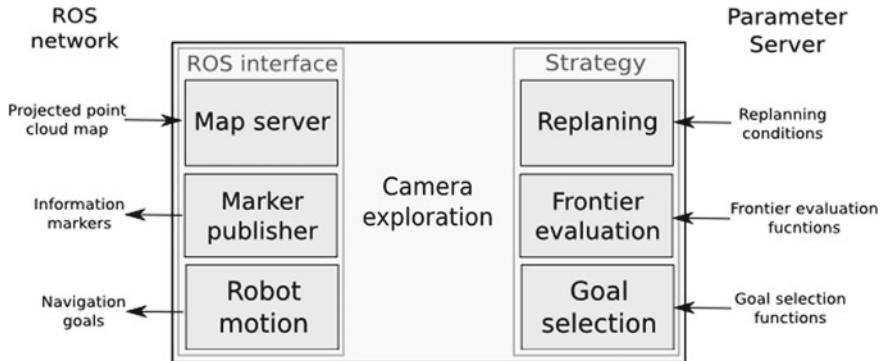


Fig. 9 Main *cam_exploration* structure with its main libraries

6.1.1 ROS Package: Exploration Framework

As exploration for 3D mapping is a novel paradigm, the needs for versatility in testing different strategies is a key requirement. Hence, to choose the appropriate sequence of 2D navigation goals, a frontier based navigation framework has been developed as a ROS package (*cam_exploration*), which is publicly available at UbiCALab github account, see fn. 4.

The main source of information used for the exploration is the projection of RGBD camera point clouds on the ground. This is achieved using *rtabmap_ros*, a package based on the work presented in [10]. Basically, the package provides a whole SLAM implementation for point cloud data.

The data flow starts with the RGBD readings from the sensors which are published as ROS point cloud messages. These messages are used by *rtabmap* node to build a 3D model and to compute its projection on the ground as a map. This allows the differentiation of unexplored regions and explored ones. At this point, the *cam_exploration* node uses the projection on the ground for exploration.

The *cam_exploration* code structure is shown in Fig. 9 with the developed libraries it contains. All the map related information is handled by the *map_server* library. The node also provides visual information of its state using markers, which are handled by the *marker_publisher* library. To keep track of the robot location and handle the interaction with the *move_base* node, the *robot_motion* library is used.

An important feature of this framework is its modularity. The main strategic decisions of an exploration that can be configured are:

- **Replanning.** To decide whether to send a new goal for exploration, a set of replanning conditions are used. Each condition represents a situation when it is desirable to send a new goal. Such conditions can be combined between them, so that their combination is what actually determines whether to send the new navigation goal. The combination method is an OR operation, so if any condition is met, the robot should send a new goal. This prevents the robot from getting stuck and takes profit

of the information received while heading to the goal.

The current implementation includes the following replanning conditions:

- *not_moving*: The robot is currently not heading to any goal.
- *too_much_time_near_goal*: Spending too much time near the goal. It votes for replanning if the robot has spent some time near its current goal and it is properly oriented with the goal.
- *isolated_goal*: The goal is not close to any frontier. It is activated when none of the cells in an arbitrary large neighborhood of the goal corresponds to a frontier.

- **Frontier evaluation.** The main policy to be studied in frontier-based exploration is the choice of the goal frontier among the complete set of frontiers. To achieve that, a cost function is usually defined taking into account some criteria. Frontier evaluation methods are defined which can be combined in a weighted sum to provide the final evaluation. The implemented frontier evaluation functions are:

- Maximum size (*max_size*). It favors larger frontiers over smaller ones.
- Minimum euclidean distance (*min_euclidian_distance*). It favors frontiers which are closer to the robot position, regardless of the obstacles in between.
- Minimum \mathcal{A}^* distance (*min_astar_distance*). The \mathcal{A}^* algorithm is intended to find a minimal cost path from a start point to a goal point in a grid. This optimal path is measured for each frontier from the robot's location and used as a distance measure. The function favors the frontiers with shorter distances in this sense.

- **Goal selection.** After a frontier is selected as the next exploration target, choosing a proper 2D navigation goal is not trivial. Specially, when working with projected point clouds. The actual 2D navigation goal is selected such that it is within the selected frontier and the robot cameras face the unexplored zone. The current implementation uses the frontier middle point (*mid_point*). In practice, any function can be used to select the frontier point. A likely to consider function is the closest frontier point to the robot location.

All the options can be configured in the parameter server, so a simple YAML can be used to describe the exploration strategy in use.

6.1.2 ROS API

Following, the node main subscriptions, publications and parameters are described. Note that the actual 3D map is published by *rtabmap* node.

Subscribed Topics.

- **/proj_map** (*nav_msgs/OccupancyGrid*)

Incoming map from *rtabmap* consisting in 3D camera point cloud projections.

Published Topics.

- **/goal_padding** (*visualization_msgs/Marker*)

Region considered as the goal neighbourhood for robot-goal proximity purposes.

- **/goal_frontier** (*visualization_msgs/Marker*)
Target frontier.
- **/goal_marker** (*visualization_msgs/Marker*)
Selected goal point.

Parameters.

- **/cam_exploration/frontier_value/functions** (*list(string)*, default: [])
List of frontier evaluation functions to be used. Possible values are *max_size*, *min_euclidian_distance* and *min_astar_distance*.
- **/cam_exploration/ < function > /weight** (*double*, default: 1.0)
Value used to weight the function < function >.
- **/cam_exploration/min_euclidian_distance/dispersion** (*double*, default: 1.0)
Degree of locality of the function *min_euclidian_distance*.
- **/cam_exploration/min_astar_distance/dispersion** (*double*, default: 1.0)
Degree of locality of the function *min_astar_distance*.
- **/cam_exploration/minimum_frontier_size** (*int*, default: 15)
Minimum number of cells of a frontier to be considered a target candidate.
- **/cam_exploration/goal_selector/type** (*string*, default: “*mid_point*”)
Way of choosing one of the target frontier points for target point. Only *mid_point* is implemented.
- **/cam_exploration/distance_to_goal** (*double*, default: 1.0)
Distance between the actual 2D navigation goal target frontier point. Should be close to the usual distance from the robot footprint to the nearest 3D camera point cloud projection point.
- **/cam_exploration/replaning/conditions** (*list(string)*, default: [])
List of replanning conditions to be applied. Possible options are *not_moving*, *too_much_time_near_goal* and *isolated_goal*.
- **/cam_exploration/too_much_time_near_goal/time_threshold** (*double*, default: 0.3)
Maximum time in seconds allowed for the robot to be near a goal in *too_much_time_near_goal* replanning condition.
- **/cam_exploration/too_much_time_near_goal/distance_threshold** (*double*, default: 0.5)
Minimum distance from the goal at which the robot is considered to be near it in *too_much_time_near_goal* replanning condition.
- **/cam_exploration/too_much_time_near_goal/orientation_threshold** (*double*, default: 0.5)
Maximum orientation difference between the one of the robot and the one of the goal, to allow replanning in *too_much_time_near_goal* replanning condition.
- **/cam_exploration/isolated_goal/depth** (*int*, default: 5)
Minimum rectangular distance from the goal to its nearest frontier allowed without replanning in *isolated_goal* replanning condition.

6.2 *Location of RFID Items*

Location of RFID labeled items is an actual topic of interest [12] and robotics contribution is paramount since it allows identification of items from multiple locations and knowing precisely the coordinates of that locations. Combining the latter with the detection model of the RFID sensor and applying proper probabilistic algorithms can output reasonable locations of the RFID labeled items. At the moment, an algorithm for the location of items is under validation. The estimated accuracy of the location algorithm is between 1 and 2 m, which is expected to be improved. The basic idea for the improvement of the accuracy is using extended detection instances and enhanced observations of the environment. For that, a precise location algorithm is being explored.

7 Future Work

In this section, a set of features in an exploratory or early development stage are discussed.

7.1 *Collaborative Inventorying*

The maximum area that the AdvanRobot can cover in a night shift is highly dependent on product density, and the complexity of the store layout. In sections with a lot of products per square meter AdvanRobot must slow down to allow enough time for the RFID system to read the thousands of tags that may be visible to the robot from a single pose. Another limiting factor may be sections with very narrow and/or irregular aisles, in which the effective speed of AdvanRobot is reduced.

As a result, each section of the store will require a minimum time for AdvanRobot to inventory. It may happen that a single robot is not able to inventory the entire store in a day. In this case there are two options: to complete the inventory in several days, or to employ a multi-robot network. Several robots may benefit from machine to machine communication to complete the inventory. This approach is not only more general and flexible, but also much more robust, as one robot may complete the job of another robot that might have malfunctioned or run out of battery.

7.2 *UAVs and AdvanRobot Collaborative System*

AdvanRobot achieves a 99.5% accuracy taking inventory in shop floors that are compliant with its navigation requirements. In order to extend the target scenarios,

Table 6 Impact of introducing UAVs in the system

	AdvanRobot	UAVs	Combined impact
Autonomy	High	Low	Mobile charging station
Maneuverability (DOF)	2.5	6	World observations enhanced. Reading height > 2.75 m
Passage width (cm)	>70	$\leqslant 70$	Target scenarios extended
Reading throughput	High	Low	Accuracy > 99.5%

for instance to warehouses and distribution centers, and aiming at higher accuracy rates the collaboration with UAVs is considered.

When the AdvanRobot and UAVs will be working together, it is foreseen that the AdvanRobot will read most of the RFID tags due to its very high reading throughput, which makes it very efficient at inventorying dense environments. Oppositely, while a drone can reach places that AdvanRobot cannot reach, its reading throughput is much lower since the RFID system it can load and supply cannot be as powerful as that of the robot.

On the other hand, the UAV can inventory areas that are not accessible to the AdvanRobot: aisles narrower than 70 cm and shelves higher than 250 cm. Also, by observing the environment from a higher point of view, it can provide additional information to the navigation system for planning and exploration. Accordingly, the overall mission efficiency can be increased.

In addition, the robot can act as a charging station to UAVs. Usually UAVs have a limited autonomy due their limited payload capacity which implies low-capacity batteries. Hence, using the AdvanRobot as a mobile charging station can improve the operational availability of UAVs.

Table 6 summarizes the benefits of a collaborative system combining robots and UAVs.

7.3 Applications Derived from Product Location

The location of items on a map of the store enables the development of valuable applications both for customers and retailers. First, by knowing the location of an item it is possible to detect if this is misplaced. Item misplacement is a source of frustration for customers and implies a cost for retailers. An unknown misplaced item can be considered as stolen. Second, the location of items can be used to guide customers and associates to find easily a product or to produce an optimal path to find a set of products. This is commonly known as wayfinding and its output can reduce greatly the time needed for picking the products of orders placed online. Last, given

the location of items, it is possible to analyze the profitability of products placements. For instance, a heat map of sales for a given product in different locations can be generated.

7.4 Simulation

Performing an end-to-end simulation of the system including the RFID propagation and detection model is a matter of interest. For the time-being, there is no Gazebo plugin for the faithful simulation of an RFID system due to its complexity. Thus, bringing forward RFID simulation in Gazebo is an interesting topic to work on in the future.

References

1. Bertolini, M., G. Ferretti, G. Vignali, and A. Volpi. 2013. Reducing out of stock, shrinkage and overstock through RFID in the fresh food supply chain: Evidence from an Italian retail pilot. *International Journal of RF Technologies* 4 (2): 107–125.
2. Durrant-Whyte, H., and T. Bailey. 2006. Simultaneous localization and mapping: Part I. *IEEE Robotics Automation Magazine* 13 (2): 99–110.
3. Engelson, S.P. 2000. Passive map learning and visual place recognition. Ph.D. thesis, Yale University.
4. EPCglobal: EPC Radio-Frequency Identity Protocols Generation-2 UHF RFID, Specification for RFID Air Interface, Protocol for Communications at 860 MHz 960 MHz, Version 2.0.1 Ratified. 2015. http://www.gs1.org/sites/default/files/docs/epc/Gen2_Protocol_Standard.pdf.
5. European Commission: Proposal for a directive of the European parliament and of the council on the approximation of the laws, regulations and administrative provisions of the member states as regards the accessibility requirements for products and services. 2015. <http://ec.europa.eu/social/BlobServlet?docId=14813&langId=en>.
6. GS1. 2014. Regulatory status for using RFID in the EPC Gen 2 band (860 to 960 MHz) of the UHF spectrum.
7. Hardgrave, B.C., J. Aloysius, and S. Goyal. 2009. Does RFID improve inventory accuracy? A preliminary analysis. *International Journal of RF Technologies: Research and Applications* 1 (1): 44–56.
8. Heese, H.S. 2007. Inventory record inaccuracy, double marginalization, and RFID adoption. *Production and Operations Management* 16 (5): 542–553.
9. House of Representatives of the United States of America. 1990. Americans with Disabilities Act of 1990. http://www.gs1.org/docs/epc/UHF_Regulations.pdf.
10. Labbe, M., and F. Michaud. 2014. Online global loop closure detection for large-scale multi-session graph-based SLAM. In *Proceedings of the IEEE/RSJ international conference on intelligent robots and systems*, 2661–2666.
11. Miesen, R., F. Kirsch, and M. Vossiek. 2013. UHF RFID Localization based on synthetic apertures. *IEEE Transactions on Automation Science and Engineering* 10 (3): 807–815.
12. NASA. 2016. RFID-enabled autonomous logistics management (realm) (RFID logistics awareness). http://www.nasa.gov/mission_pages/station/research/experiments/2137.html.
13. Nur, K., M. Morenza-Cinos, A. Carreras, and R. Pous. 2015. Projection of RFID-Obtained product information on a retail stores indoor panoramas. *IEEE Intelligent Systems* 30 (6): 30–37. Nov.

14. Rekik, Y., E. Sahin, and Y. Dallery. 2009. Inventory inaccuracy in retail stores due to theft: An analysis of the benefits of RFID. *International Journal of Production Economics* **118** (1), 189–198. <http://www.sciencedirect.com/science/article/pii/S0925527308002648>. (Special Section on Problems and models of inventories selected papers of the fourteenth International symposium on inventories).
15. RFID Journal. 2013. Tag-reading robot wins RFID journal's coolest demo contest. <http://www.rfidjournal.com/articles/view?10670>.
16. Sarma, S., D. Brock, and D. Engels. 2001. Radio frequency identification and the electronic product code. *IEEE Micro* **21** (6): 50–54. Nov.
17. Wang, J., E. Schluntz, B. Otis, and T. Deyle. 2015. A new vision for smart objects and the internet of things: Mobile robots and long-range UHF RFID sensor tags. *CorR*. [arXiv:abs/1507.02373](https://arxiv.org/abs/abs/1507.02373).
18. Yamauchi, B. 1997. A frontier-based approach for autonomous exploration. In *CIRA*, 146–151. New York: IEEE Computer Society.

Author Biographies

Marc Morenza-Cinos is a PhD candidate at Universitat Pompeu Fabra. His research interests include Robotics, Wireless Communication and Data Mining. Morenza-Cinos has an MSc in Information and Communication Technologies from Universitat Politcnica de Catalunya. Contact him at marc.morenza@upf.edu.

Victor Casamayor-Pujol is a PhD candidate at Universitat Pompeu Fabra. His research interests include Robotics, Artificial Intelligence and Aerospace. Casamayor-Pujol has a MSc in Intelligent Interactive Systems from Universitat Pompeu Fabra and a MS in Space Systems Engineering from Institut Supérieur de l’Aeronautique et l’Espace. Contact him at victor.casamayor@upf.edu

Jordi Soler-Busquets is a Robotics MsC student at Universitat Politcnica de Catalunya. His academic interests include Machine Learning and Robotics. Contact him at jordi.solerb@upf.edu

José Luis Sanz is Industrial Designer at Keonn Technologies. His career has been developed between Conceptual Design and Industrial Design and development for manufacturing. José Luis has a degree in Industrial Design Engineering from Jaume I University in Castelln, a degree in Design from Universitat Politcnica de Catalunya in Barcelona and a posgraduate in Composite Materials from Eurecat Technological Center.

Roberto Guzmán owns the degrees of Computer Science Engineer (Physical Systems Branch) and MSc in CAD/CAM, and has been Lecturer and Researcher in the Robotics area of the Department of Systems Engineering and Automation of the Polytechnic University of Valencia and the Department of Process Control and Regulation of the FernUniversitt Hagen (Germany). During the years 2000 and 2001 he has been R&D Director in Althea Productos Industriales. He runs Robotnik since 2002. Contact him at rguzman@robotnik.es.

Rafael Pous is an associate professor at Universitat Pompeu Fabra. His research interests include Ubiquitous Computing, Retail Technologies and Antenna Design. Pous has a PhD degree in Electrical Engineering from University of California at Berkeley. Contact him at rafael.pous@upf.edu.

Robotnik—Professional Service Robotics Applications with ROS (2)

**Roberto Guzmán, Román Navarro, Miquel Cantero
and Jorge Ariño**

Abstract This chapter summarizes new experiences in using ROS in the deployment of Real-World professional service robotics applications. These include climbing mobile robot for windmill inspection, a mobile manipulator for general purpose applications, a mobile autonomous guided car and a robot for the detection/measurement of surface defects and cracks in tunnels. It focuses on the application development of the ROS modules, tools, components applied, and on the lessons learned in the development process.

Keywords Professional service robotics with ROS · Robots for inspection · Service robotics · Autonomous robots · Mobile robots · Mobile manipulators · RB1 · RB1-Base

1 Contributions of the Book Chapter

ROS has become a standard for the development of advanced robot systems. According to the statistics presented in the ROS metrics report [1] that measures statistics related with awareness, membership, engagement, and code metrics. The community is also growing exponentially.

This chapter describes a number of professional service robotics applications developed in ROS. The number of robots using ROS in professional service robotics is continuously growing. However, even the ROS community and the number robotics startups are using ROS in their developments rise, the number of publicly documented Real-World applications and in particular in product development and commercialization is still relatively low.

R. Guzmán (✉) · R. Navarro · M. Cantero · J. Ariño
Robotnik Automation, SLL, Ciutat de Barcelona, 3A, P.I. Fte. del Jarro,
46988 Paterna, Valencia, Spain
e-mail: rguzman@robotnik.es
URL: <http://www.robotnik.eu>

This chapter presents as main contribution the description of four real products that use ROS, detailing the principal challenges found from the point of view of a ROS developer.

2 ELIOT: Climbing Robot for Windmill Inspection

Eliot is a climbing robot developed to address windmill maintenance. Maintenance of a windmill includes cleaning windmill shafts, painting blades, oil changes, tomography images shafts, and other tasks. Eliot was developed for Eliot Systems [2] a company that has patented several automated solutions for the cleaning and inspection of the windmill. The inspection robot makes use of a patented solution to climb metallic surfaces using magnetic tracks.

The robot climbs in semi-autonomous or teleoperated mode to the top of the windmill mast, takes detailed pictures with a high-res camera of cracks on the blades. Eliot uses thermal imaging to process the information from the cracks. The robot also able to mount several payloads to perform NDT measurement operations on the mast itself. Here is a brief summary of system, work done, different components, packages (used and developed), and the developed HMI with problems found (Fig. 1).

Due to the nature of the project and associated NDA (Non Distribution Agreement) with the end user, the packages of this robot are not available. Some of the packages used (non protected) are listed below.

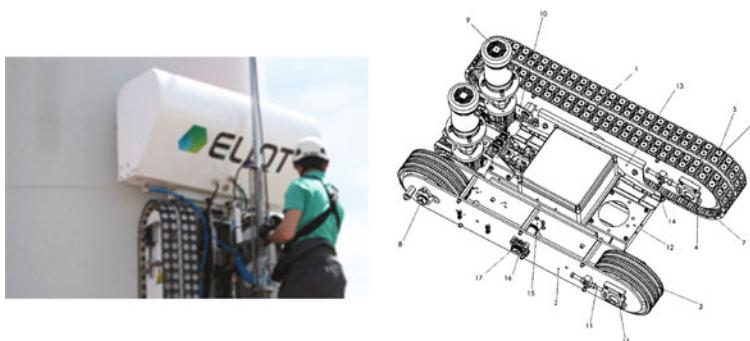


Fig. 1 ELIOT robot platform

multimaster_fkie:	https://github.com/fkie/multimaster_fkie
	a ROS meta-package that offers a complete solution for using ROS with multicores.
summit_xl_sim:	https://github.com/RobotnikAutomation/summit_xl_sim
	a simulation package for robots of the Summit family.
summit_xl_common:	https://github.com/RobotnikAutomation/summit_xl_common
	common packages (pad, navigation, localization, etc.) of the summit robot.
imu_tools:	https://github.com/cnny-ros-pkg/imu_tools
	a set of IMU-related filters and visualizers. The <code>imu_filter_madgwick</code> was used to filter the raw imu data from the arduimu. The meta-package includes also a plugin for the visualization of the imu in rviz.
robotnik_arduimu:	https://github.com/RobotnikAutomation/robotnik_arduimu
	ROS package for the Arduimu Board. Needs an adapted ArduIMU firmware that provides the raw data to be filtered externally.
gps_common:	http://wiki.ros.org/gps_common
	a package that provides common GPS-processing routines. The <code>gpsd_client</code> was used to read and process the gps data.
robotnik_gyro:	https://github.com/RobotnikAutomation/robotnik_gyro
	reads a gyroscope, a device used together with the internal IMU gyros.
axis_camera:	https://github.com/RobotnikAutomation/axis_camera
	contains Robotnik basic Python drivers for accessing an Axis camera's MJPG stream based on <code>axis_camera</code> ROS driver. Also provides control for PTZ cameras.

2.1 Brief Description of the System

This section focuses on Eliot *Preview*. Preview is the name of the smallest robot model developed by Eliot Systems with the purpose of inspection of towers and blades of windmills. The company has developed several robots with different sizes and functionality, in this case, they selected Robotnik for the development of this unit.

2.2 Robot Configuration

The robot mounts two reinforced tracks with a set of magnets that are able to stick to the metallic tower mast. The robot has an autonomy of 4 h and is able to climb at speeds of 200 mm/s. The weight of the robot is <50 kg. The robot traction was designed in a way that generates an electric brake in case of power loss. The standard operation permits this system to descend at a controlled low speed in this case (Fig. 2).



Fig. 2 ELIOT robot platform, different prototypes

2.3 *Robot Sensors*

Eliot Preview mounts a standard set of sensors for outdoor mobile robot platforms. These sensors include inertial measurement unit, wheel encoders and GPS. Other than the standard sensors it mounts specialty sensors which includes an inclinometer (provides a redundant measurement of the robot pose angles for climbing), and a high quality PTZ camera (for vision based inspection of the blades). The robot mounts a set of lights to illuminate operation areas and for visibility seen from a distance.

2.4 *Communications*

The robot is wirelessly operated from a distances of 100–200 m. As there is always LOS¹ between the base station and the robot, communication was established with standard WiFi devices with high gain antennas. A redundant channel was implemented in order to get control of the traction in case of failure of the CPU or standard communication channel.

2.5 *HMI*

The HMI was implemented in a laptop running a web server, the multimaster meta-package, the pad and joystick controller, to safely tele-operate the robot.

In short summary, the html interface has 3 tabs:

- main: cameras, sensors and robot state

¹Line Of Sight.

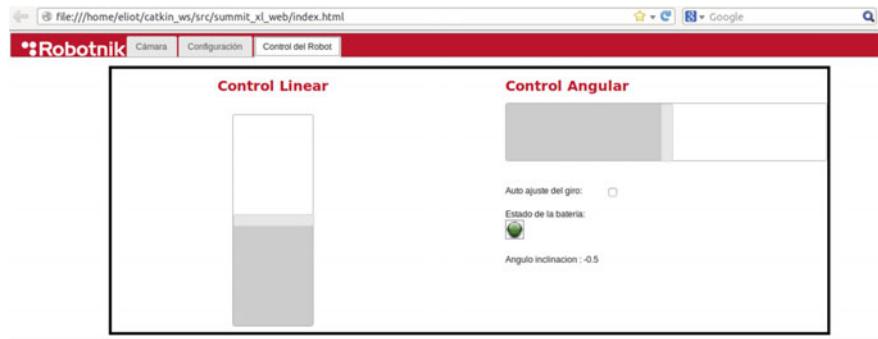


Fig. 3 ELIOT control tab of the web HMI

- configuration: parameters, max angles, ranges, autonomous control, etc.
- robot control: motion interface for the operator

The main screen allows the user to get detailed images from the areas of interest and permits the control of the pan-tilt-zoom. The interface allows the camera to predefined points or can be set to a programmable home position. The HMI has specific controls to store videos and pictures of the ongoing mission. It contains functions to download the files and manage subfolders. The main interface shows coloured markers to inform status of the IMU, battery and IMU temperature. These values are monitored and raise alarm in case they critical status flagged with red. The exact measured values are also available in the main screen (Fig. 3).

The configuration screen permits the activation/de-activation of the IMU/inclinometer based control (this can be useful in special circumstances, e.g. when the robot needs to perform pure horizontal motion). Other parameters are angle limits, tilting limits, horizontal offset and minimum rotation radius, which can be programmed also from this screen.

2.6 Challenges

The main challenges in the development of Eliot are related to safety of the robot and operators. The robot kinematic configuration allows it to adapt to different tower radiiuses and to climb different types of towers. A critical issue has been to keep the appropriate climbing angle for each condition. The magnet tracks allow the robot to climb the tower in almost any angle, but complete free motion is less safer than climbing at controlled angles. Without appropriate control, poor teleoperation of the robot may end in a fall from a dangerous height.

In order to guarantee a safe climb and a safe movement around the tower, the robot inclination was monitored with a redundant system using an IMU and an inclinometer. A state machine was defined which controlled the robot angle and

allowed to stop its motion if safety limits were exceeded. The internal control system is in charge of keeping the correct variable ranges which allows the robot to be commanded to climb up and down in autonomous mode.

There are wide range of applications on windmills that require height safety products, including vertical access to masts and access to the external elements of the wind turbine. Standard windmill infrastructure includes a set of height safety products that permit easy installation of a safety rope in the windmill mast. Eliot can therefore benefit from additional safety, a measure that was a common practice during testing and development phase of the robot.

Another important design decision was related with the robot reliability. The system was designed fault-tolerant in order to avoid the need to send a person to pick it up from the mast. The fault tolerance starts from the set of two tracks with double line of magnets and double motor traction. In case of a motor failure, the robot is still able to climb up and down.

Several options were considered to communicate the HMI with the robot. The selected configuration used one master in the HMI and one master in the robot, they communicate through the multimaster_fkie [3] package, that offers a set of nodes to establish and manage a multi-master network. The advantage of this configuration is that both nodes are independent, and can operate even in case of complete loss of the communication.

Deploying this kind of network is relatively easy using this package and requires only a minimal configuration (by default, all the topics are shared between the different masters, but it can be changed to reduce the bandwidth charge). Once the system is deployed, the communication between topics and services of the different networks are completely transparent. It seems like the system is running on a common network with the advantage a master could die without fatal consequences for the other masters (Fig. 4).

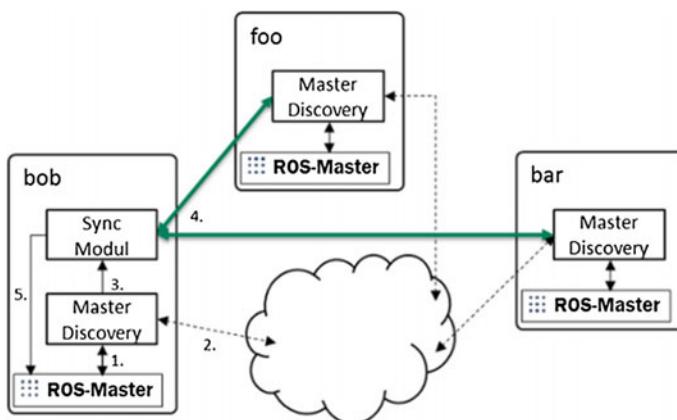


Fig. 4 Example diagram of the multi-master architecture

The communication system was made redundant. While the standard robot tele-operation is done via wifi, the robot permits a higher priority communication channel that permits the remote control of the robot even during CPU failure.

Similar to other professional service robotics applications, the use of ROS proved to be the right choice allowing easy development and reliable behaviour of the provided software components.

3 RB-1: A Mobile Manipulator for General Purpose Applications

The demand of indoor mobile robots and mobile manipulator applications has continuously been growing, from inspection to measurement and logistics. The IFR [4] predicts exponential rise in applications in upcoming years. The purpose is to address the widest application range with the objective of providing a robust and industrial grade solution. Under these objectives, Robotnik developed the RB-1 BASE and the RB-1 mobile manipulator.

RB-1 is a modular robot composed of a mobile robot base, robotic torso with a linear axis, pan-tilt head and a lightweight Kinova [5] robot arm with gripper.

This section presents a description of the robot, some real applications already implemented, and focuses on the different components and packages used/developed.

The main topics involved in this development are: URDF/Xacro [6], Gazebo [7], MoveIt [8], teb_local_planner [9].

The simulation packages for the RB1 can be found in:

https://github.com/RobotnikAutomation/rb1_common

https://github.com/RobotnikAutomation/rb1_sim

https://github.com/RobotnikAutomation/mico_common

https://github.com/RobotnikAutomation/mico_sim

The simulation packages for the RB1 BASE robot can be found in:

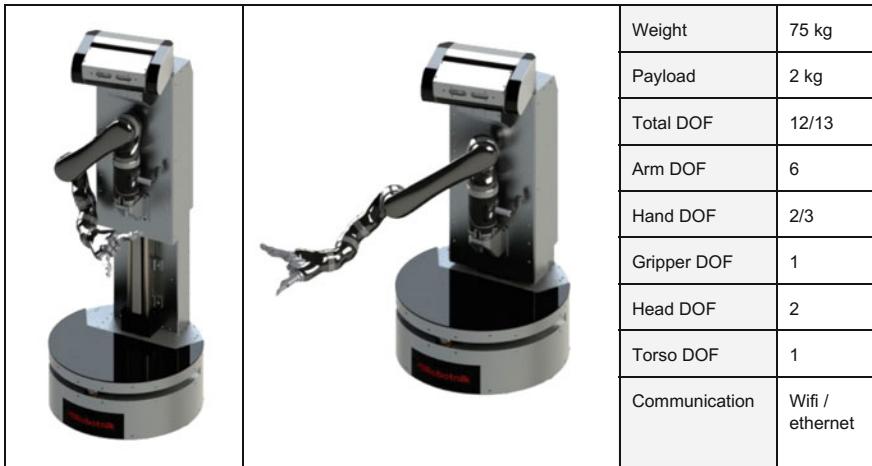
https://github.com/RobotnikAutomation/rb1_base_common

https://github.com/RobotnikAutomation/rb1_base_sim

The ROS version used is Indigo running on Ubuntu 14.04. The controller should have an i7 processor with at least 8GB RAM.

3.1 Brief Description of the System

RB-1 was designed to address the research and application development market of general purpose mobile manipulation.



Weight	75 kg
Payload	2 kg
Total DOF	12/13
Arm DOF	6
Hand DOF	2/3
Gripper DOF	1
Head DOF	2
Torso DOF	1
Communication	Wifi / ethernet

Fig. 5 RB-1 general purpose mobile manipulator

RB-1 can integrate the Kinova Mico or Jaco arms. Both are 6-DOF that can incorporate a 2 or 3 fingers gripper. These arms designed for assistive robotics and improve the quality of life of people with disabilities. They are extremely robust and provide very interesting functionalities for mobile manipulation which will be described next.

The software of the robot includes a control system, a tracking system (laser-based), a navigation system, as well as a HMI user interface for diagnostics and remote control.

The robot integrates a 2-DOF pan-tilt for the perception of the environment by an included RGBD (for Red, Green, Blue plus Depth) sensor and an additional RGBD sensor in the base. The RGBD sensors have various applications on the robot. The head sensor can be used to recognize and localize objects around it. It can also be used for navigation and location purposes by using benchmarks or using new RGBD Slam algorithms. The RGBD sensor in the mobile robot base is used for obstacle avoidance and to identify the robot charger location and autonomous docking (Fig. 5).

The mobile base is also sold separately for general purpose application development but principally for logistics. The RB-1 base platform is a differential robot developed for industrial grade applications indoors. The platform can carry up to 75 kg in its latest version, with a speed of 1.75 m/s, autonomy of 10 h and integrates a wide range of lasers, allowing a field of view of 270° (Fig. 6).

The RB-1 Base robot mounts two traction servomotors in a differential configuration. The base weight is distributed in the traction wheels via an independant suspension system and on omnidirectional castors, arranged for a stable footprint. The access to the main components, battery and controller has been designed in boxes that have front side (battery) or rear side (control box) access. This design simplifies manufacturing but also maintenance allowing fast and simple component



Fig. 6 RB-1 base platform

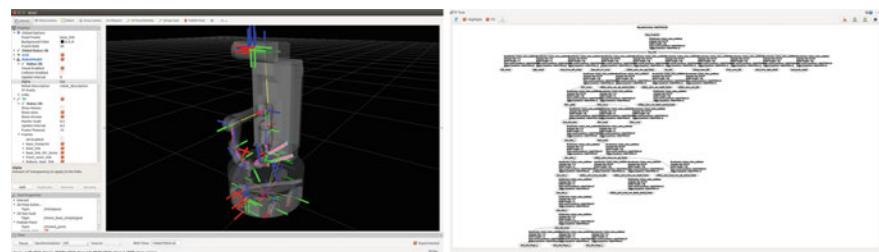


Fig. 7 RB-1 description

replacement. The battery is mounted in the robot bottom to reduce the height of the center of gravity. The battery includes contacts and electronics to dock to the robot self-recharging station.

3.2 Main Topics Covered

This section describes the different components and software packages used and developed.

3.2.1 URDF/Xacro

Sources: rb1_description

For robot simulation, the model description is required. Every robot part is distributed in different urdf files and assembled in the main robot file, e.g. rb1_robot_mico_3fg.urdf.xacro. This package integrates urdf/xacro files of the mico and jaco arms (Fig. 7).

```
$ roslaunch rb1_description rb1_mico_3fg_rviz.launch
```



Fig. 8 RB-1 Gazebo

3.2.2 Gazebo

Sources: rb1_sim/rb1_gazebo, rb1_sim/rb1_control (mico_sim/mico_arm_control mico_sim/mico_arm_gazebo)

Packages developed to simulate the robot in Gazebo (Fig. 8).

```
$ roslaunch rb1_gazebo rb1.launch
```

3.2.3 Moveit

Sources: rb1_common/rb1_mico_3fg_moveit_config, rb1_common/rb1_jaco_3fg_moveit_config, mico_common/mico_moveit_config

Moveit config packages of the robot with different arms and grippers. The rb1 moveit packages use the `robotnik_trajectory_suite/robotnik_trajectory_control` as a package to coordinate `FollowJointTrajectory` action between different sets of controllers that do not follow the ros control standard. On the contrary, the `mico_moveit_config` package makes use of the ROS controller type: “`effort_controllers/JointTrajectoryController`”. For simulation, both controllers can be used, however the `JointTrajectoryController`, results have smooth motions and simplifies the adjustments of axes PID gains (Fig. 9).

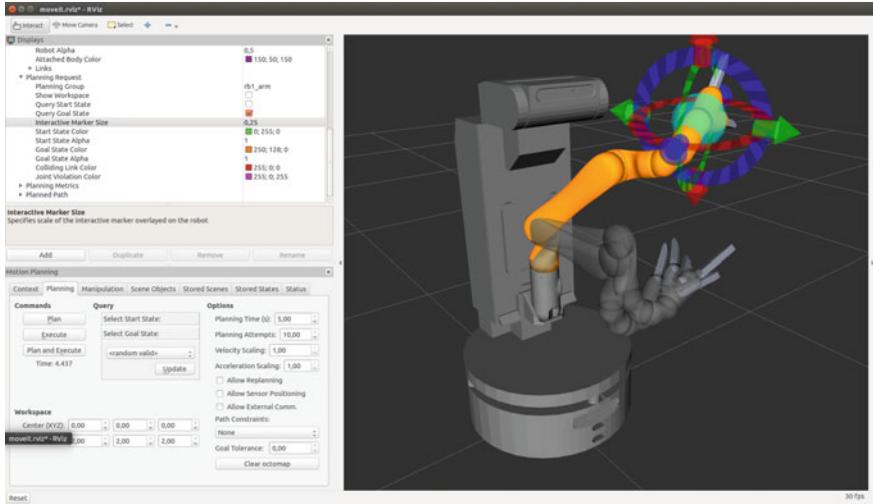


Fig. 9 RB-1 MoveIt!

```
$ roslaunch rb1_sim_bringup gazebo_moveit_trajectory.launch
```

3.2.4 Navigation

Sources: rb1_sim/rb1_gazebo, rb1_sim/rb1_control, rb1_base_sim/rb1_base_2dnav

The rb1_gazebo package includes simulation of the robot and environment in gazebo. The rb1_base_2dnav package implements several configurations of the move_base stack using different local planner plugins. It uses the default base_local_planner/TrajectoryPlannerROS (trajectory rollout) and allows also the new teb_local_planner [9]. This planner uses an underlying method called Time Elastic Band which produced optimized trajectories in the local costmap in as a difference with the default approach. The TEB planner optimizes the trajectory with respect to trajectory execution time, separation from obstacles and compliance with kinodynamic constraints during runtime.

The rb1_base_2dnav package includes pre-configured launch files to create maps and to localize using amcl [10] (Fig. 10).

```
$ roslaunch rb1_gazebo rb1_office.launch
```

```
$ roslaunch rb1_base_2dnav rb1_base_gmapping.launch
```

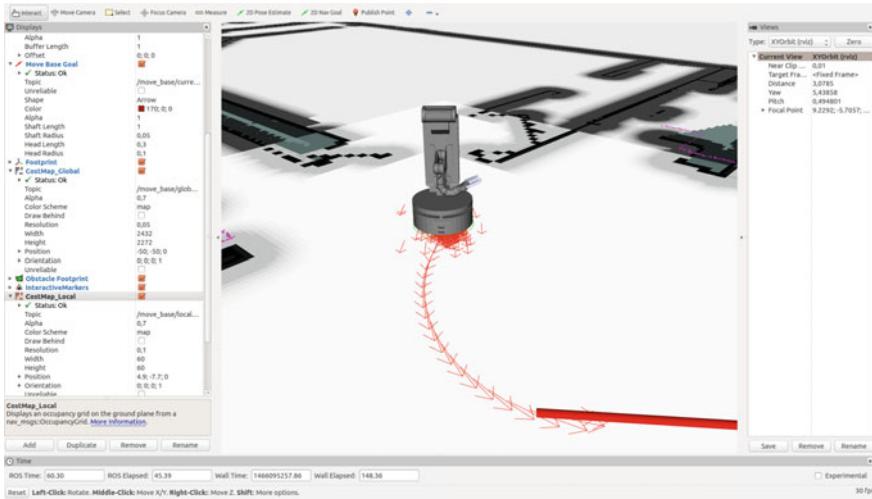


Fig. 10 RB-1 navigation

```
$ rosrun rb1_base_2dnav move_base_amcl_teb.launch
```

3.3 Challenges

The main challenge addressed in the robot design was related with the actuator selection. The first design used a robot arm developed with Dynamixel PRO actuators. This kept the system with as few references as possible, we also decided to use this kind of actuator in the mobile base (2×200 W), torso elevation and head pan-tilt unit. However, the Dynamixel PRO actuator electronics do not allow regenerative brake which makes the servomotor inadequate for industrial traction applications. The first units had special heaters and fans mounted on the traction motors to overcome the problem. The performance was not as expected and the cost of the hardware was high compared to brushless motors used in other Robotnik products. The performance of the heavier 7-DOF arm with cycloidal gearboxes, compared to the advanced lightweight Kinova arms with harmonic drive gearboxes, was the key factor to change the servo motors.

Another unexpected problem was detected when the first Kinova robot arm was mounted on the RB-1. The Kinova controller implements a homing sequence that is intended to be commanded from the device joystick via a software application. It is possible to change the homing sequence since the arm joints have absolute encoders and use an initialization algorithm. This sequence was not accessible via the robot api and could not be changed. The original homing sequence makes the arm collide with the robot head, so it was necessary to find a way to change it. Kinova provided a firmware update to overcome this limitation.

The final challenge to mention was the development of the docking station algorithm. The objective of this algorithm is to guide the robot towards the self-recharging station so that it can autonomously recharge the battery. The robot has to follow the landmarks (in this case QR-codes) on the docking station until the battery charging plates are in contact with the charger pins. The mechanism has some compliance to adapt to positioning inaccuracies but the algorithm needs to be correctly tuned to dock accurately 100% of the time. The current algorithm makes use of visual servoing to perform the correct tracking in a two stage approximation and its development took an unexpected effort.

4 RBCAR: A Mobile Autonomous Guided Car

In the last years a number of autonomous person transport applications have emerged. From early adopters to the latest developments of Google, Tesla and Toyota, the progressive introduction still requires the overcoming of technical and regulatory barriers.

The mobile robot RBCAR was designed with the purpose of providing a low-cost autonomous car driving platform for application development and R&D. The robot provides the modular automation of an electric car with a full set of sensors, traction control, direction and brakes. RBCAR uses Ackerman kinematics. The traction is controlled by an AC motor with incremental encoder and the direction through a power steering system with an absolute encoder (Fig. 11).

With the configuration of suitable sensors, the robot can navigate autonomously, teleoperated with a joystick or a steering wheel, as an electric vehicle.

This part describes the software developed, how ROS has been used in the robot software implementation (simulation, control and navigation) and how a new 3D sensor from Hokuyo has been used and integrated in the robot navigation.



Fig. 11 RBCAR robot

The main topics covered in this section are: URDF/Xacro [6], Gazebo [7]. The simulation packages for the RBCAR robot can be found in:

- https://github.com/RobotnikAutomation/rbcar_common
- https://github.com/RobotnikAutomation/rbcar_sim
- https://github.com/RobotnikAutomation/robotnik_purepursuit_planner

The rbcar_common repository contains all the common packages needed by the simulated and real robot.

- **rbcar_description**

- It contains the urdf, *meshes*, and other elements needed in the description are contained here.

- **rbcar_pad**

- This package allows controlling the robot using a joystick or game-pad, by sending the messages received through the joystick input, correctly adapted, to the correct command topic.

The rbcar_sim is composed of the following packages:

- **rbcar_control**

- This package contains all the configuration files needed to simulate the motor controllers in Gazebo (using skid_steering plugin).

- **rbcar_gazebo**

- This contains the launch and config files to launch Gazebo with the robot.

- **rbcar_robot_control**

- This is the robot's Gazebo plug-in controller. It implements the control of the ackerman kinematics of the robot, controlling the traction and steering motors. This component publishes the robot's odometry.

- **rbcar_sim_bringup**

- This contains several launch files in order to launch some or all the components of the robot.

The robotnik_purepursuit_planner is composed by the following packages:

- **robotnik_purepursuit_planner**

- ROS meta-package implements the pure pursuit algorithm for mobile robots.

- **robotnik_pp_msgs**

- This contains the messages and actions definition (GoTo action).

- **robotnik_pp_planner**

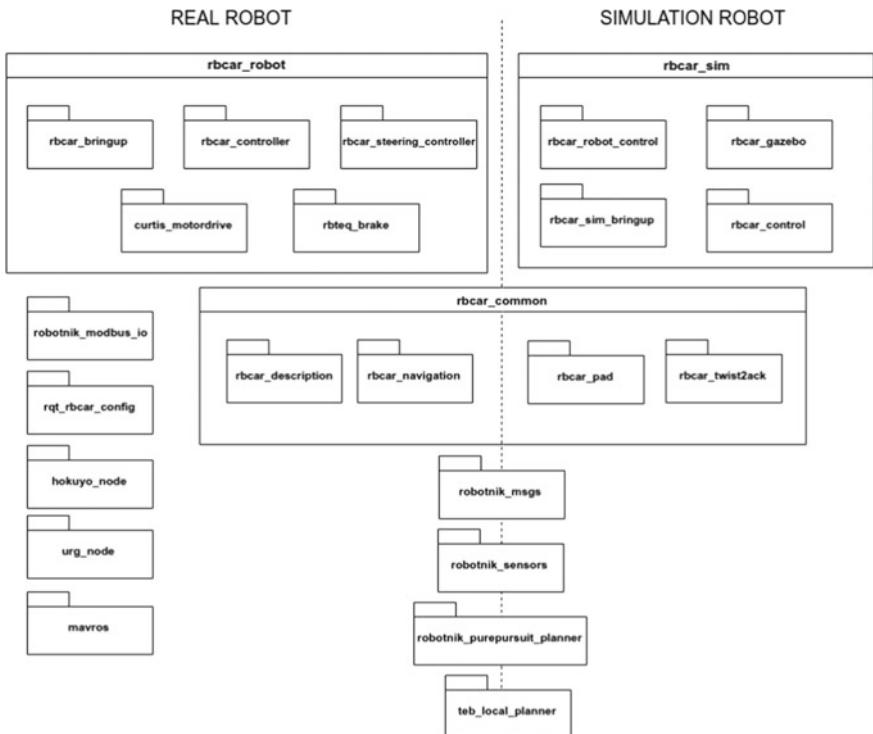


Fig. 12 RBCAR robot software architecture

- This is the component that performs the navigation. It contains functions to add/delete waypoints, to preprocess a path and the pure pursuit algorithm and state machine to follow a path (Fig. 12).

4.1 Brief Description of the System

The mobile robot RBCAR is based on a common design of electric vehicle with Ackermann steering. The robot can be controlled automatically or in manual mode. In automatic mode, the robot can receive ackermann_msgs/AckermannDriveStamped either from an external joystick or from an autonomous robot planner as the one provided with the robotnik_purepursuit_planner metapackage. In manual mode the robot is operated exactly as an electric car. In addition to the traction and direction controllers, the RBCAR basic equipment includes distributed modbus input/outputs, safety electronics to handle the e-stop, 2D/3D hokuyo laser range finder option and a wide range of GPS options (Fig. 13).



Weight	700 kg (with batteries)
Dimensions	2660 mm x 1230 mm x 1720 mm
Payload	2 persons + 150 Kg on the box
Speed	32 Km/h
Motor	3.3 kW AC 48V
Autonomy	70 Km
Brakes	Hydraulic
Body	ABS thermoformed
Frame	Galvanized Welded Steel
Max. climbing angle	25%
Controller	ROS PC with Linux
Communication	Wifi / ethernet

Fig. 13 RBCAR autonomous robot main technical specifications

4.2 Main Topics Covered

4.2.1 URDF/Xacro

The rbcar_description package contains the urdf and xacro files of the robot model. The main robot file: robots/rbcar.urdf.xacro, integrates the robot chassis from urdf/bases/rbcar_base.urdf.xacro and the wheels from urdf/wheels/suspension_wheel.urdf.xacro (Fig. 14).

The modelling of the suspension system is based on the ackermann_vehicle model by Jean-Baptiste Passot [11]. In the kinematic chain of each wheel a linear joint type is added with specific parameters of damping and friction of the shock absorber. Even these axes are passive, it is possible to set the JointEffortController with no references (zero position reference), thus allowing the PID controller parameters to model the system response (Fig. 15).

```
$ rosrun rbcar_description rbcar_rviz.launch
```

4.2.2 Gazebo

Sources: rbcar_sim/rbcar_gazebo, rbcar_sim/rbcar_control, rbcar_sim/rbcar_robot_control

The above packages were developed to simulate the robot in Gazebo [7]. These packages set the motor controllers and configuration to run the platform in the sim-

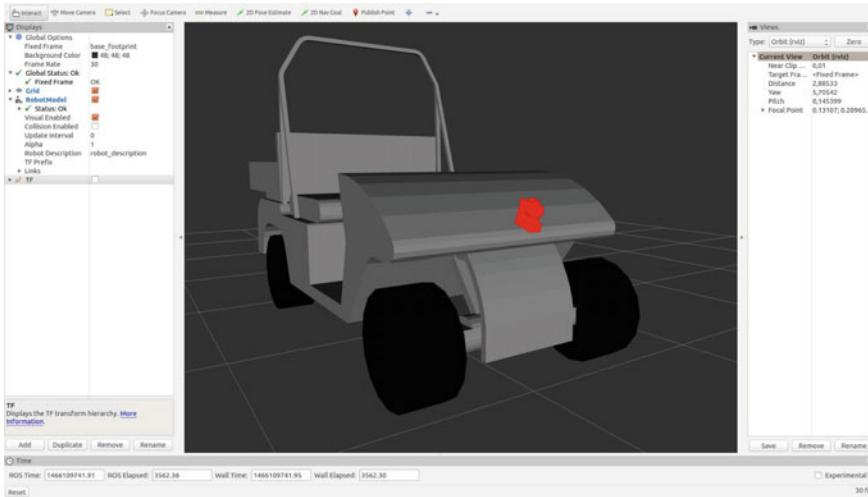


Fig. 14 RBCAR description



Fig. 15 RBCAR tf tree

```
$ roslaunch rbcar_sim Bringup rbcar_complete.launch
```

ulator. It is an example of how the Ackermann kinematics can be set in Gazebo (Fig. 16).

4.2.3 Navigation

The package “`robotnik_purepursuit_planner`” implements the *PurePursuit* algorithm either for Ackermann or Differential robots. The robot package is now also compatible with the `tew_local_planner`, that is able to plan correctly for Ackermann steering.

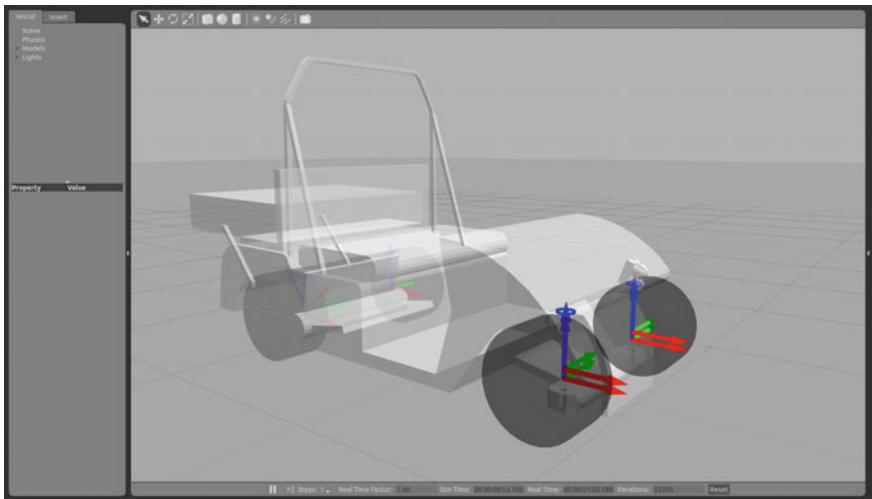


Fig. 16 RBCAR Gazebo

In order to test the following path algorithm with the purepursuit algorithm, the purepursuit node needs to be launched.

```
$ roslaunch rbcar_sim_bringup purepursuit.launch
```

Then the path marker node from the robotnik_pp_planner needs to be launched to allow the user to define waypoints interactively and send path following commands in the ROS visualization tool.

```
$ roslaunch rbcar_sim_bringup purepursuit_marker.launch
```

Launch the RVIZ to visualize and send the trajectories to the robot.

```
$ rosrun rviz rviz
```

In the Displays menu, add the InteractiveMarkers with the topic /path_marker/update. After that you will see a red marker in front of the robot. By clicking with the right button, a context menu appears offering the options to add waypoints at different speeds, to delete waypoints, to save the path or to make the robot follow the path, interrupt it, or make it execute backwards (Fig. 17).

This will test the robot navigation with the teb planner (Fig. 18):

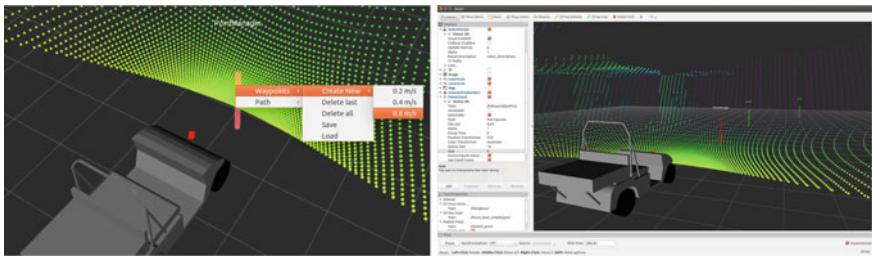


Fig. 17 RBCAR Interactive marker path definition in RVIZ

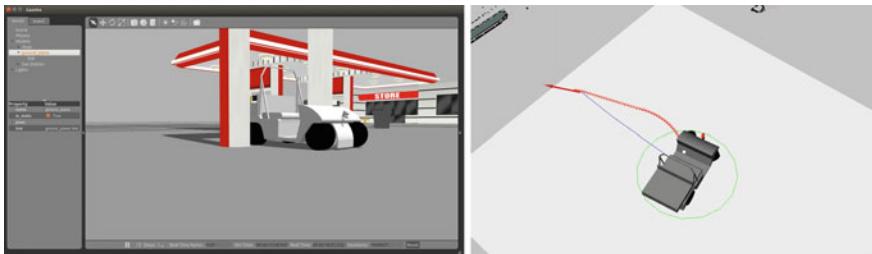


Fig. 18 RBCAR Path following in the simulated environment

```
$ rosrun rbcar_sim bringup rbcar_complete_gs.launch
```

```
$ rosrun rbcar_navigation move_base_amcl_teb.launch
```

4.3 Challenges

One challenge to mention during development of the RBCAR was related with the automation of the steering axis. A servomotor is installed to act as the electric power steering in a vehicle that has a pure mechanical steering mechanism. The steering axis is controlled in position using the internal PID of the servo amplifier, that needs to be correctly tuned to adapt to different ground frictions and robot payload. The steering system mounts two encoders, an absolute encoder to define the orientation of the wheels and an incremental encoder in the actuation motor.

The braking distance was the next challenge. The electric vehicle mounts a pedal brake that actuates the four wheel hydraulic brakes. Without actuation of the hydraulic brake, the traction servo amplifier has braking distances that can reach 15 m at the max speed of 32 km/h. For some applications this is not good enough, so an automatic brake was developed that is now sold as an option for the RBCAR. The design, manufacturing, adjustment and programming of this extra brake was the most time consuming task of the whole project.

5 ROBO-SPECT: Robot for the Detection and Measurement of Surface Defects and Cracks in Tunnels

The assessment of structural integrity of existing civil structures is of great importance to identify and determine its reliability levels on the ability to carry existing and future loads and fulfil its task having in mind human life, financial, maintenance and operational risks [12].

The objective of ROBO-SPECT is to provide an automated, faster and reliable tunnel inspection and assessment solution that can combine in one pass both inspection and detailed structural assessment that does not interfere with tunnel traffic. ROBO-SPECT is co-funded by the European Commission under FP7-ICT-Robotics [13].

Driven by the tunnel inspection industry, ROBO-SPECT proposes an advanced mobile manipulator with navigation capabilities in tunnels in order to automatically scan the intrados for detection of potential defects on the surface, measurement of radial deformations in the cross-section, measuring distances between parallel cracks, measure cracks and open joints that may impact the tunnel stability. The proposed solution allows in one pass, both the inspection and structural assessment of tunnels. Intelligent control and robotics tools are interwoven to set an automatic robotic arm manipulation and an autonomous vehicle navigation to minimize human interaction. This way, the structural condition and safety of a tunnel is assessed automatically, reliably and quickly.

The ROBO-SPECT project integrates a range of technologies including vision systems, NDT measurement of cracks, structural assessment, HMI, etc. The work carried on by Robotnik is related with the mobile robot control, navigation and communications (Fig. 19).

This part describes the software developed for this robot, how ROS has been used in the robot software implementation (simulation, navigation and control).

Most of the packages developed for the project can be found at <https://github.com/RobospectEU>.

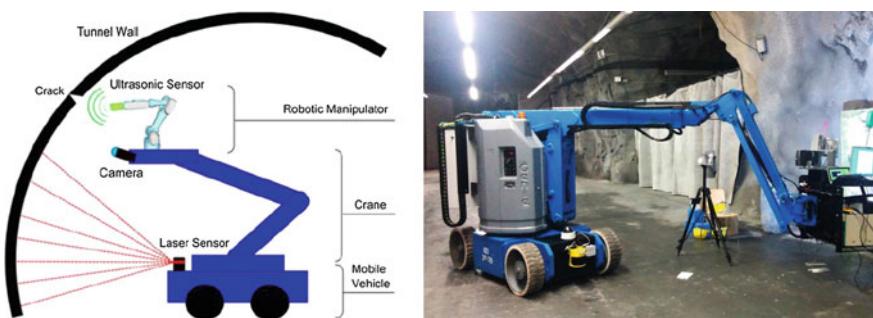


Fig. 19 ROBO-SPECT robot initial concept and the robot prototype during first trial in a tunnel



Fig. 20 ROBO-SPECT sensors for control and navigation

Common packages: https://github.com/RobospectEU/robospect_common
 Simulation packages: https://github.com/RobospectEU/robospect_sim

5.1 Brief Description of the System

The goal in ROBO-SPECT is to offer an autonomous and automated system for concrete transportation tunnels that will provide efficient structural inspection and an accurate structural assessment in one pass.

The mobile robotic platform (vehicle and crane) consists on an industrial wheeled robotic system able to extend an automated crane to the dimensions of most common tunnels. The vehicle chosen is the Genie Z30/20N [14], a 6.5 m height articulated crane vehicle, able to move on roads and flat surfaces.

This platform was adapted and modified in order to be automated and controlled through ROS (Fig. 20).

For control purposes the following sensors and devices were installed into the vehicle:

- Two incremental encoders for traction control and one absolute multi-turn encoder for the steering control to control the mobile platform.

- Two absolute multi-turn encoder, 4 internal (embedded in the pistons) linear encoders and 1 external linear encoder to control the crane.
- One Programmable Logic Controller for the low-level control of the vehicle.

For navigation purposes the vehicle was equipped with several navigation sensors installed in the vehicle in different positions due to their different use. The following sensors were introduced:

- Two SICK S3000 laser systems for obstacle avoidance and safety of the vehicle, one in the front and the other in the rear of the robot. The working distance is ~ 50 m.
- One pan-tilt-zoom camera on the front for teleoperation of the vehicle. Its image is transmitted to the HMI interface in the control station.
- One SICK NAV200 laser to detect the artificial landmarks inside the tunnel. The NAV200 calculates its own position and orientation on the basis of fixed reflectors positioned in the environment. Its maximum range is 30 m, but in operational conditions where a minimum of 3 landmarks are needed for detection, the range is about 20 m.
- One gyroscope CRG20 system to improve the odometry of the robot.

5.2 Main Topics Covered

5.2.1 URDF/Xacro

Sources: robospect_common/robospect_description

The Robospect URDF description contains all joints and links according to the vehicle specification and other elements like a robotic arm and a pan-tilt camera. All the data is in the package robospect_description.

The robot model tries to follow a modular structure. Components like the base, crane, arm and cameras are defined in the folder urdf, while the whole robot model is defined in the folder robots. This structure allows building different configurations easily and with a package that is more understandable (Fig. 21).

```
$ roslaunch robospect_description robospect_rviz.launch
```

For a more detailed description it is recommendable to use the rqt_tf_tree tool in order to visualize the tree of links and how they are interconnected (Fig. 22).

5.2.2 Gazebo

Sources: robospect_sim

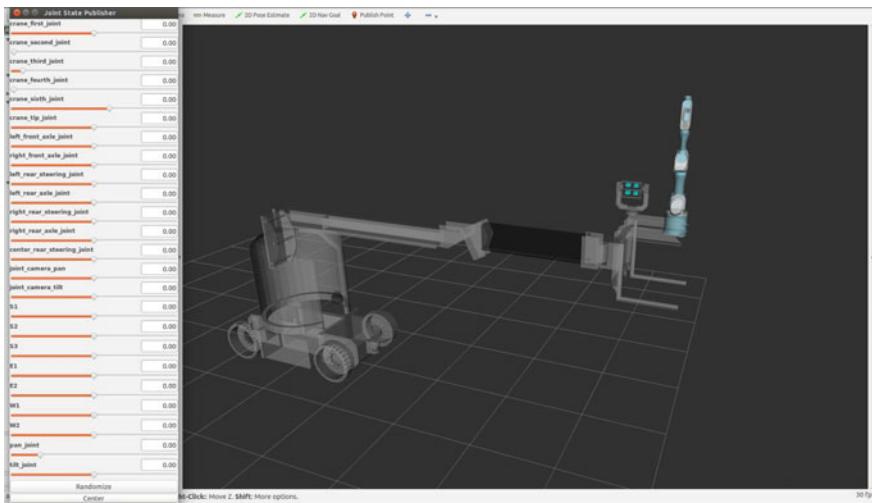


Fig. 21 Robot model visualized with RVIZ and the node joint_state_publisher to test all the joints

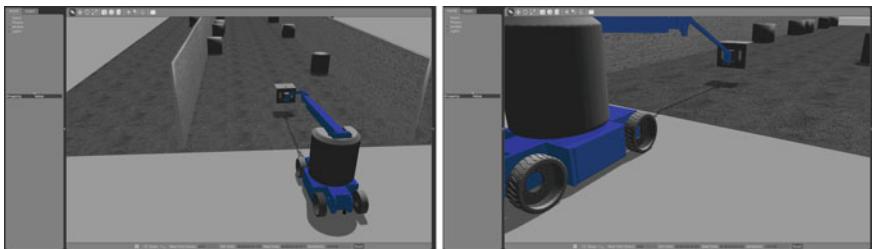


Fig. 22 Robospect simulation in Gazebo in a test environment

```
$ roslaunch robospect_sim_bringup robospect_complete.launch
```

5.2.3 Localization and Navigation

The localization of the vehicle is provided by the NAV200 and the installation of reflective beacons in the tunnel area (Fig. 23).

This localization system was chosen for the following reasons:

- The symmetry of the tunnels does not allow standard SLAM algorithms to work properly.
- Good accuracy. The theoretical position accuracy of the sensor is between ± 4 and ± 25 mm, depending on the beacons distribution and the number of them that sensor is detecting.
- Out-of-the-box system, robust and easy to configure.

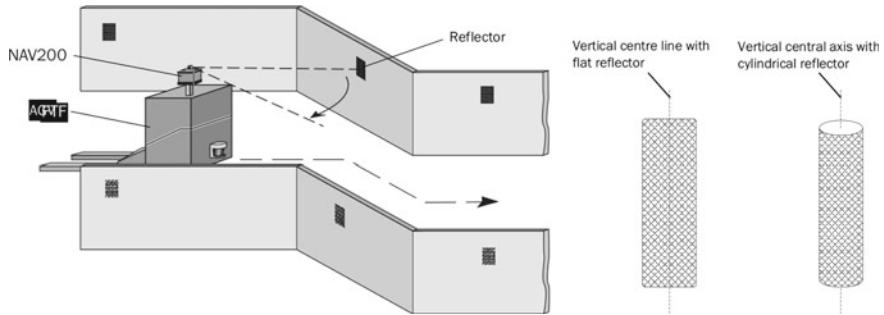


Fig. 23 Robot localization system used in the tunnels

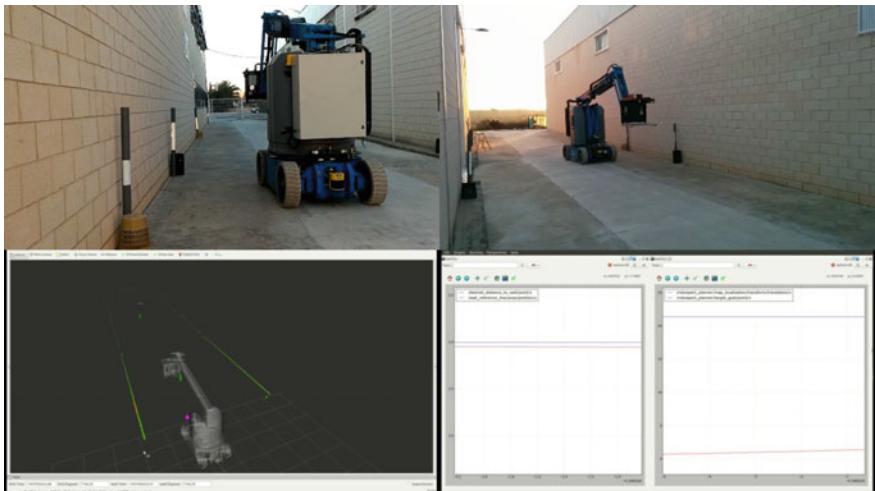


Fig. 24 Robot localization and navigation tests

Using the on-board navigation sensors and installed landmarks (beacons) in the tunnels, the navigation will focus on tracking the tunnel wall at a defined distance, commonly between 1 and 2 m. The location of the landmarks will be in the range of 15 m between, being commonly 3–4 marks visible from the vehicle in any time.

A ROS driver for the NAV200 [15] was developed to communicate with the sensor and provide the transform between map and odom.

For the navigation through the tunnels, an algorithm was implemented to follow the tunnel wall at a desired distance from the wall. This approach was necessary since the vision system needs to be at the same approximated distance from the wall in order to make its algorithms work while detecting cracks and defects successfully.

The package developed [16] uses the scan data from the lasers to calculate the target position and command the vehicle controller (Fig. 24).

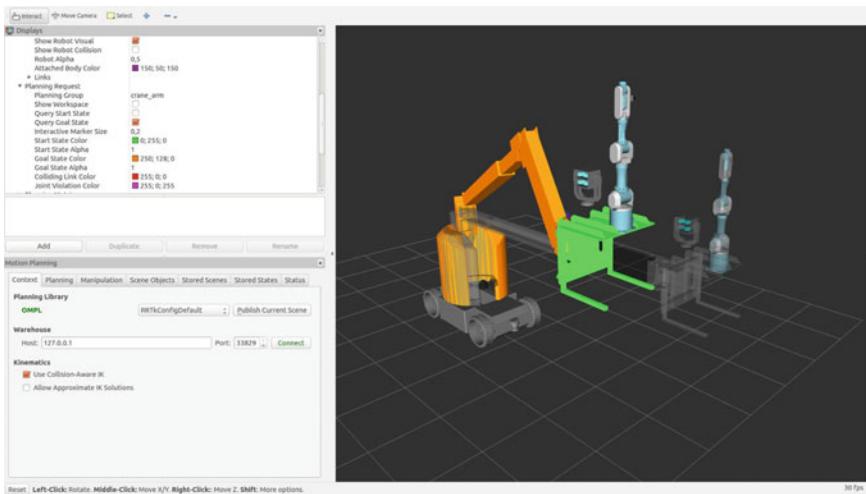


Fig. 25 Robot MoveIt! configuration.

5.2.4 Crane Control with MoveIt!

The selected crane has a configuration which allows to cover most of the tunnel types of the ROBO-SPECT project. It is a 6 DOF system with independent motion of every joint (axis). Due to the limitation of the hydraulic on-board system (main pump), the joints move one-by-one. It means that the crane path will be the combination of independent joint motion.

A package [17] based on MoveIt was developed to solve the forward and inverse kinematics of the crane (Fig. 25).

```
$ roslaunch robospect_moveit_config demo.launch
```

5.3 Challenges

5.3.1 Vehicle Automation

The automation of an hydraulic-actuated machine turned out to be a real challenge. These kind of machines are not intended for automated control due to the lack of a smooth actuation. Despite the installation of accurate sensors to measure the position of every joint, the control of these joints with on/off actuators makes it really difficult to reach the target position with minimum deviation.

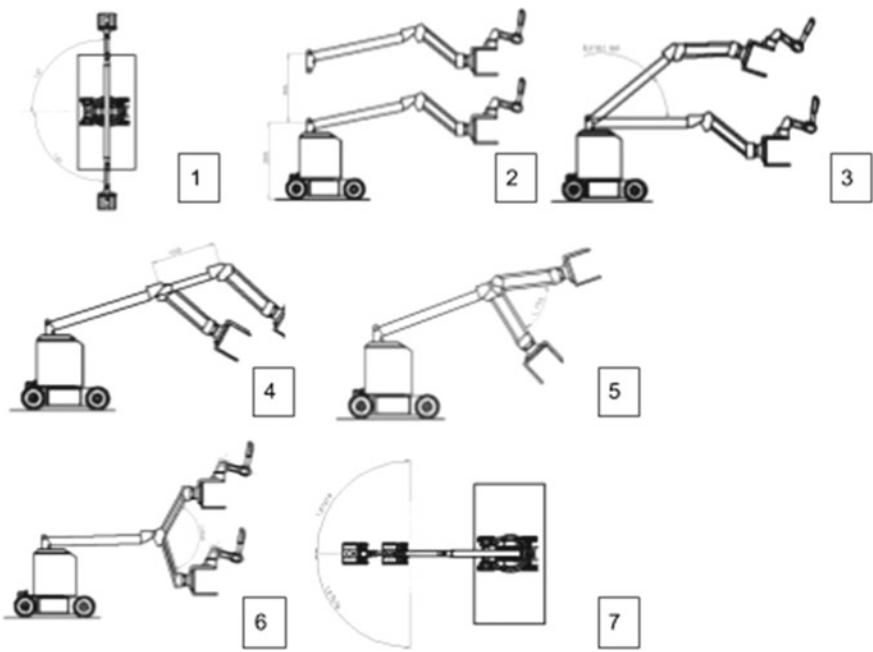


Fig. 26 Robospect crane joints configuration

A huge effort with the electronic design was also needed in order to make possible this adaptation of the machine functionality. Some reverse engineering becomes essential for this purpose.

Special mention is required with regard to the low level coding of the machine's behaviour. Debugging and tuning the program has been proved to be one of the most time consuming tasks.

5.3.2 Localization

Providing an accurate, reliable and repetitive position estimation in this type of environment was an important point of the project.

The adopted technology has been proved satisfactory, but there are some limitations we had to deal with:

- Pre-installation of the reflective beacons. The installation of many beacons was needed. For a very accurate position estimation of the beacons, a total station is required.
- Field of view limitation. The Nav200 is a 360° sensor that uses all the measurements to improve the localization. Due to the vehicle configuration, the location of the device blocks the detection of the backwards beacons. This restriction implies

- the installation of more reflectors (increasing the cost) and the reduction of the accuracy.
- The slope of the terrain. If the road is not flat, the farthest reflectors (inside the detection area) are not detected unless we increase the height of the reflective area (Fig. 26).

6 Summary

This chapter describes four projects where ROS is an integral part of a professional, commercial robotics solution: a climbing mobile robot for windmill inspection, a mobile manipulator for general purpose applications, a mobile autonomous guided car and a robot for the detection/measurement of surface defects and cracks in tunnels. ROS has been used as the main architecture in all the developed systems, and the developed packages and software architecture has been described in detail in previous sections.

A number of challenges have been found and described, many of them not directly related with the software development, but with robot and operator safety issues, design of fault tolerant and redundant systems, actuators or firmware functionalities, process control/automation related problems or lack of process or component information.

Some challenges were related with the ROS architecture or the development itself and a number of lessons have been learnt that will be summarized next. A lot of details of the packages have been discovered, but this summary is about general characteristics.

The first lesson is about the system robustness. ROS has proven to be reliable and fault tolerant in all the mentioned applications. In most cases it is used in a daily bases and once the system is working, the number of failures is really low.

A second lesson is about the utility of the system simulation. By simulating the robot and environment (process) it is possible to speed up the development by the parallelization of the hardware and software development. However, creating realistic simulation environments in Gazebo is not an easy task. It always implies a high cost in manpower and in many cases it is not possible to get the desired result. The limitations are both from a wrong configuration (and lack of documentation) or due to bugs or model limitations of Gazebo. Even so, it is usually possible to model the system up to a certain level, thus making possible the validation of some functionalities, while many others have to be validated with the real robot.

The communication middleware is in current ROS version² has turned out to be a limitation. In some cases, configuring a multi-master network allowed the system operation in case of communication loss, but even in this case, communication problems arise, specially related with the service connections and action services.

²Replaced by DDS in ROS 2.

As ROS is a distributed system, it requires a correct design in the distribution of nodes in the architecture. This design allows a development that can be easily distributed among a community of developers and at the same time a much easier future system maintenance.

ROS still has some limitations at the user interface level. rqt is a useful tool for engineers but it does not ease the development of professional user interfaces (at least in a documented and simple way). The interfaces developed have in common a basic appearance, specially compared with other systems where there is a wide range of libraries that facilitate the development.

Finally, despite the drawbacks, our experience has been extremely positive in the four examples given, but also in all products we have developed in the past. We therefore encourage all prospective ROS users to give ROS a try as the advantages are enormous compared with just a few drawbacks.

References

1. Foote, T. Community Metrics Report - Reporting on July 2015 - Comparisons are to August 2014 Report.
2. Eliot Systems. <http://www.eliotsystems.com/en/index.php>.
3. multimaster_fkie. 2016. Retrieved from http://wiki.ros.org/multimaster_fkie.
4. IFR. 2016. Retrieved from <http://www.ifr.org/service-robots/>.
5. Kinova. <http://www.kinovarobotics.com/>.
6. URDF Unified Robot Description Format. 2015. Retrieved from <http://wiki.ros.org/urdf>.
7. Gazebo. 2016. Retrieved from <http://gazebosim.org/>.
8. MoveIt! (2016). Retrieved from <http://moveit.ros.org/>.
9. teb local planner. 2016. Retrieved from http://wiki.ros.org/teb_local_planner.
10. amcl Adaptive Monte Carlo Localization. 2016. Retrieved from <http://wiki.ros.org/amcl>.
11. ackermann_vehicle model. 2016. Retrieved from https://github.com/jbpassot/ackermann_vehicle.
12. Rücker et al. 2006. Federal Institute of Materials Research and Testing (BAM), Division VII.2 Buildings and Structures, Unter den Eichen 87, 12205 Berlin, Germany - SAMCO Final Report 2006, A Guideline for the Assessment of Existing Structures.
13. Robo-spect EU. 2016. Retrieved from <http://www.robo-spect.eu/>.
14. Genie Z30/20N. <http://www.genielift.com/en/products/boom-lifts/articulating-booms-electric/z3020n/index.htm>.
15. Xacro. <http://wiki.ros.org/xacro>.
16. NAV200. https://github.com/RobotnikAutomation/nav200_laser.
17. robospect_planner. https://github.com/RobospectEU/robospect_planner.
18. robospect_moveit_config. https://github.com/RobospectEU/robospect_common/tree/master/robospect_moveit_config.

Author Biographies

Mr. Roberto Guzmán (rguzman@robotnik.es) earned the degrees of Computer Science Engineer (Physical Systems Branch) and MSc in CAD/CAM, and has been Lecturer and Researcher in the Robotics area of the Department of Systems Engineering and Automation of the Polytechnic

University of Valencia and the Department of Process Control and Regulation of the FernUniversität Hagen (Germany). During the years 2000 and 2001 he has been R&D Director in “Althea Productos Industriales”. He runs Robotnik since 2002.

Mr. Román Navarro (rnavarro@robotnik.es) earned the degree of Computer Science Engineer (Industrial branch) at the Polytechnic University of Valencia. He has worked at Robotnik since 2006 as software engineer and in the R&D department.

Mr. Miquel Cantero (mcantero@robotnik.es) earned the degree of Industrial Engineering (specialized in automation and electronics) by the Polytechnic University of Valencia. He has worked in Robotnik’s engineering department since 2014, also collaborating with R&D and european projects FP7 and H2020.

Mr. Jorge Ariño (jarino@robotnik.es) earned the degree of Computer Science Engineer (Industrial branch) at the Polytechnic University of Valencia. He has worked at Robotnik since 2014 as software engineer and in the R&D department.

Using ROS in Multi-robot Systems: Experiences and Lessons Learned from Real-World Field Tests

**Mario Garzón, João Valente, Juan Jesús Roldán, David Garzón-Ramos,
Jorge de León, Antonio Barrientos and Jaime del Cerro**

Abstract This chapter presents a series of experiences and lessons learned during several implementations and real-world tests of ROS-based Multi-Robot Systems. It also describes, analyses and compares several ROS components relevant for these applications, taking into account the scenarios where they can be used. Also, some general issues of importance of Multi-Robot Systems on real-world, such as software and communications architectures, types of information shared are described in detail. Finally, the difficulties and specific challenges that arose when using a Multi-Robot Systems for any application will be discussed.

Keywords Multi-robot systems · Field robotics · ROS for multi-robot systems · Robot cooperation

M. Garzón (✉) · J. Valente · J.J. Roldán · D. Garzón-Ramos · J. de León ·

A. Barrientos · J. del Cerro

Centro De Automática y Robótica, UPM-CSIC, Calle José Gutiérrez Abascal, 2,
28006 Madrid, Spain

e-mail: ma.garzon@upm.es

J. Valente

e-mail: joao.valente@upm.es

J.J. Roldán

e-mail: jj.roldan@upm.es

D. Garzón-Ramos

e-mail: dgarzon@etsii.upm.es

J. de León

e-mail: jorge.deleon@upm.es

A. Barrientos

e-mail: antonio.barrientos@upm.es

J. del Cerro

e-mail: j.cerro@upm.es

1 Introduction

Nowadays, one of the main challenges for the robotics community is to take robots out of the laboratories and use them in real-life applications. Although a large amount of resources have been directed in this direction, the challenge remains. Furthermore, when working with multi-robot systems (MRS), even more complications arise. ROS has been a very useful tool in this regard. It has allowed the increase of maturity levels, and has placed itself in the centre of many robotics developments.

This chapter does not intend to enhance the benefits of ROS in field robotics since its advantages are well known. Instead, its objective is to provide a survey of experiences and lessons learned during several implementation and real-world tests of ROS-based MRS, in a variety of applications, such as: surveillance/security [14], search and rescue [15], environmental measurements [31] and agriculture [2]. The chapter also will describe, analyse and compare several other relevant components for these applications, some examples are: ROS multi-master packages, approaches to multi-robot integration, choice in network communication technologies, wireless sensor networks, etc. The analysis will also take into account the different types of scenarios where they can be used.

Researchers interested in multi-robot systems represent an inherently diverse community¹ because several competences are needed in this field, ranging from modeling MRS [22], design and use of the heterogeneity [27], bio-inspiration and swarm intelligence [8], optimal control and optimization methods [12], control architectures and scalability [35], motion planning, coordination [13] and cooperative decision making [19], just to list a few.

As the number of different robotic applications increases, it becomes more and more common to involve different types of robots simultaneously. This results in several advantages for perception systems compared to a setting that involves only one single robot because richer information may become available. In order to benefit from these advantages, several challenges need to be addressed. First, when using more than one robot, it becomes necessary to fuse information obtained from individual robots. Second, use of different types of robotic platforms often requires consideration of a broader variety of sensor types with different sensing modalities, and thus, different types of measurements. In this chapter, we will discuss how these challenges can be addressed in practical applications, particularly for estimation and mapping scenarios. This discussion will cover simultaneous consideration of different levels of uncertainties, a sound representation of underlying domains, and cooperation of multiple heterogeneous agents in large-scale map building. The presented methodology will involve directional and robust statistics, and use of optimization based estimation approaches

In order to summarize the contents of this chapter, a brief description of the topics covered is presented next:

¹<http://multirobotsystems.org/>.

- First, a background section on multi-robot systems is presented. It describes some of characteristics as well classifications that can be made according to different aspects or the MRS.
- Then, some general aspects of any multi-robot software architecture will be described and immediately after that, some considerations about the different technologies for communication in a MRS.
- The next section is focused on a comparative description of the different multi-master packages available on the ROS repositories.
- After that, a few applications of MRS using ROS that have been developed by the authors are presented, in all cases the proposed solution and the software architecture used is described.
- Finally, the chapter concludes with a series of lessons learned as well as some of the issues that need to be solved in order to facilitate the use of ROS based MRS in any application.

2 Background

Multi-robot systems can be defined as a group of robotic agents coordinated in order to perform a particular task, which may not be achievable by a single robot or can be optimized if performed by a group [16]. This means that an analysis of the task is required in order to determine whether or not to use a MRS. This analysis should be focused on finding a possible increase in efficiency obtained when using a MRS instead of a single robot system for a given task [6]. This background section presents a summary of classifications for the MRS according to characteristics or typologies, taking into account the size of the team, level of cooperation, morphology and the type of task performed. Taxonomy and definitions named in this section were taken from studies of several authors [1, 9, 17].

2.1 Classification by Size of the Team

One of the simplest classifications that can be made in a MRS is based on the number of agents composing the system. In this case four groups can be identified:

- **Single Unit** Although technically they are not MRS, it can be said that the minimal number of agents is one.
- **Two Units Systems** is the minimum number of robots required to form a team. Two unit teams are usually focused on tasks that a single robot can not achieve by itself or to obtain complementary capabilities.
- **Multi-unit Systems** refers to teams with several members but where the number of units is relatively small with respect to the area/task required. It is perhaps the

most common type of team, and therefore with the most varying types of robots, strategies and applications.

- **Swarms** refers to a team of robots in which the number of members is very large, even unbounded. Usually robots in these numbers have limited capabilities and they achieve their goals by emerging behaviours. Moreover, these type of teams should be highly robust to changes in the number of members.

Also, when regarding the size of the team, a different classification can be made. Not taking into account the number of team members but the capacity of the team to re-organize itself when this number changes. In this case, there are three different categories.

- **Static Arrangement** The configuration is fixed and the task can not be achieved if one of the agents is lost. It is the most simple case and is usually the configuration for Two Units Systems.
- **Coordinated Re-Arrangement** In this case, it is possible to modify the number of robots in the team. But any change of this type will require an intermediate step, such as re-negotiation or reconfiguration of the mission. Therefore each robot is assigned with a new task or set of tasks.
- **Dynamic Arrangement** This is the most complex configuration. The number of robots can change and the mission should continue without any re-assignments of tasks. This means that MRS should autonomously detect that one of its members has left and dynamically adapt the behaviour of the remaining members to achieve the main goal. This is the most common case for robot swarms. It is also used by some Multi-Unit Systems but it may result in a very complex additional arrangement sub-task.

2.2 *Classification by Morphology and Capabilities of the Robots*

The morphology of a robot refers to different aspects: The mechanical description, locomotion type and kinematics of the robot. Each one of those aspects, in combination with the computing and processing power of the robot, define the capabilities of the robot. This means that a classification of robots or MRS according to their morphology will intrinsically involve their capabilities. Knowing this, it is possible to define three different categories.

- **Identical** The members of the MRS are homogeneous in both their mechanics and their capabilities. This does not mean that all robots perform the same task, but rather that all tasks in the mission can be assigned to any of the team members, and that they can be replaced by any other team member.
- **Homogeneous** The members of the MRS have similar morphology or capabilities but not exactly the same. This means that the members of the team can perform the same task up to some extent. Whereas some of the capabilities are exclusive to

a limited sub-group of the team. For instance a team of wheeled robots that have some sensors in common but not all the same. To summarize, this means that if one of the team members is removed, the mission can be only partially completed.

- **Heterogeneous** Members of the MRS differ in both morphology, capabilities or locomotion techniques. Typical examples are MRS consisting on aerial and ground robots. In contrast with the previous categories, the mission can not be completed if one of the members is removed.

2.3 Classification by Level of Coordination

This classification defines different levels of coordination and interaction between the robots in a MRS. This type of classification has been widely studied and the categories presented here have been used in [11, 20].

A summary of the classification by level of coordination is presented on Fig. 1. In this schema, the first level of separation is based on whether or not the robots are aware that they are cooperating within a MRS. Even if robots in a team are unaware of each other, it is possible to obtain cooperative, or at least coordinated, tasks. On the other hand, when the robots have knowledge of each other, three different sub-groups are defined according to their level of coordination. In this case, coordination refers to the ability of a robot to take into account the actions executed by the other robots. When there is coordination, it can be of two types, strong or weak depending on whether or not the coordination is implicit, meaning that it follows a set of pre-defined rules for coordination (strongly coordinated) or they obtain an emerging or reactive coordination (weakly coordinated). Finally, the organization of the MRS should also be taken into account. This differentiation is based on the capacity of each member to take autonomous decisions on which actions to perform. Again, three

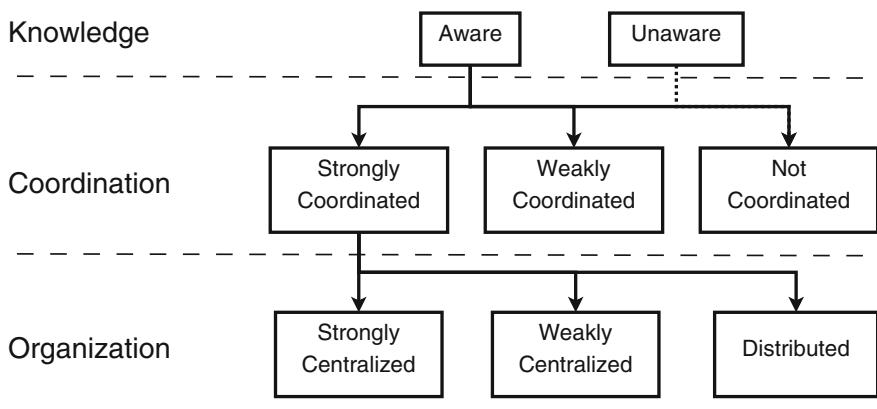


Fig. 1 MRS Classification by level of coordination

different categories are found [7]: distributed coordination, where each robot decides their task autonomously; strongly centralized systems, where the decisions on the actions of all robots in the team are taken by a single leader; and weakly centralized, where MRS's have hierarchical architectures which are locally centralized or there is only one agent taking the decision, but the role of leader can change from one robot to another at different moments or situations.

3 Multi-robot Software Architecture

The software architecture, which can be defined as a depiction of the system that aids in the understanding of how the system will behave,² is one of the most important components of any complex robotic system. This is even more evident when using a team of robots.

The design of a software architecture is a complete area of study itself [29], therefore it is out of the scope of this chapter. Rather, the objective of this section is to describe some of the characteristics needed when working with multi robot systems under ROS. The ROS core is one of the main components of the computational graph used by ROS, its definition can be taken from the ROS Wiki, at <http://wiki.ros.org/Master>: “The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.”

As it is well known, most of the applications using ROS use a centralized ROS Master, either because they are intended to be used on a single robot, or because they need to have a common point for processing the information. Moreover, namespaces are usually the main tool used for differentiation between different sub-systems of a large application. Nevertheless, in MRS set-up, the naming of the nodes, topics and parameters becomes more complex and may result in duplicities, high computing costs, large demand for communications, delay in the processes and other problems related to the system handling by an overloaded single ROS master.

Different software architectures have been designed for each one of the applications described in this chapter. However, they have many things in common. Mainly the objective of these designs was to enhance the ROS framework. Moreover, they are based on the communication between several ROS masters. For most cases: one for the base-station, and another of each of the robots on the system.

Figure 2 shows an example of multi-master software architecture, in this case developed for a signal search multi-robot system [14]. It uses a single base station, which is mainly used for visualization and control purposes. The main modules, such as target detection, autonomous navigation or other high level tasks are executed on board each robot. Moreover, in order to control the communications between the

²<http://www.sei.cmu.edu/architecture/>.

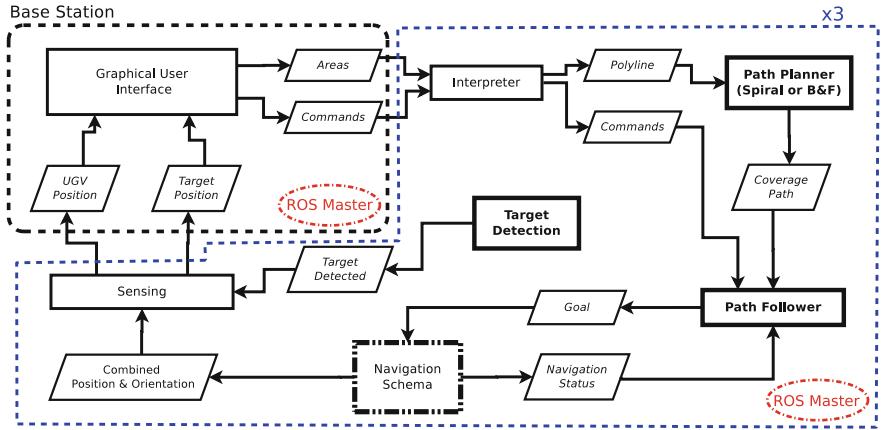


Fig. 2 Example of a software architecture for a cooperative signal search application

robots and the base station, two gateways, *sensing* and *interpreter* are used on each robot.

Another important characteristic for this architecture is the modularity. The multi-master schema facilitates the execution of the mission and reduces the required bandwidth for communications. Moreover, as shown in Fig. 2, a main-master runs on the base station and there is also one secondary master on each robot. This configuration also increases the robustness of the solution, because all the processes or ROS nodes of each robot are controlled by a ROS-master running on the same computer, or with a wired Ethernet connection. As result of the distributed management, new connections can be established and the parameter server is available although other ROS masters have not started up, shuts down, or they are no longer available due to communication issues. Furthermore, this facilitates supervision and re-spawning in case of failure.

The communication between the base station and the robots is improved also by using a multi-master schema. Even though communication between ROS nodes is point-to-point, and therefore high volume data is kept inside the robot (e.g. odometry, IMU or GPS, point clouds, laser scans, navigation data). The housekeeping data, required to control such a high number of topics and nodes, when controlled by a single master, may introduce delays or possible data loss in high value data (e.g. robot positions, target detection, mission state).

4 Communications for MRS

Communication is a key issue when dealing with multi-robot systems. Independently of considering centralized or decentralized solutions, there is always a strong need of communicating with and among the robots and the control station.

Table 1 Communication technologies comparison

Requirements	Wi-Fi	Wimax	LTE	ZigBee
Bandwidth	$\leq 7 \text{ Gbps}^a$	$\leq 140 \text{ Mbps}$	$\leq 300 \text{ Mbps}^b$	250 Kbps
Range over 1 Km	No	Yes (up to 50 Km)	Yes*	No
Mobility	Yes	Yes	Yes	Yes
Low latency	Yes	Yes	Yes	No
Unlicensed operating	Yes	Yes	No	Yes

*LTE can cover unlimited ranges

^aTheoretical maximum. Due to channel contention, real max. is between 1.7 and 2.5 Gbps

^bDepends on network provider (See <https://opensignal.com/reports/2016/11/state-of-lte>)

Most MRS use wireless communications in order to avoid restrictions on the movements of the agents. Therefore, the robustness of the communications in terms of bandwidth, range or latency turns out to be a crucial aspect when evaluating the overall system performance. The most simple form of communication is a point-to-point scheme, where the MRS agents send information directly to its receiver. However, this approach is only suitable when the number of components of the MRS is low, because when working with a considerable number of robots, channel saturation can be easily reached and delays or other problems arise. More complex approaches, such as one-to-many or many-to-many schemas are also widely used. Furthermore, a very common schema is based on exchanging information between robots that are near to each other, although they require an extra process to find which other robots are near, and sometimes combine them with simple rules.

The application of the MRS defines the two main communication requirements, which are range and bandwidth and consequently the possible technical solution to use. Currently, several existing wireless technologies are available in the market with an affordable cost. Some examples are Wi-Fi (IEEE 802.11a/b/g/n/ac standards), Worldwide interoperability for Microwave Access network known as WiMAX (IEEE 802.16), 3G or 4G LTE, ZigBee. Table 1 summarizes the main features of the mentioned technologies considering their possible use in MRS.

Considering the applications presented in this chapter, which require large areas to be covered (near to 1 Km) Wimax and LTE technologies are the most adequate a-priori. Nevertheless, from an economical point of view, the cost of equipment, licenses or the services provider make them unsuitable. Wi-Fi solutions, on the other hand, are much more interesting in this regard. Moreover, most computers and devices use Wi-Fi, therefore there is no need of additional *bridge* devices. Furthermore, the problem of distance range can be solved by adding repeaters or range extenders.

Usually, the communication systems for MRS have most of their power at a fixed or semi-mobile location near the base-station, because of power consumption and the size of the antenna at the control station this is rarely a problem, whereas they may be severely restricted in the robots. This means that the size of the on board antennas depends on the size of the robots and the environment where they are going

to operate. Moreover, the weight could be a problem, mainly when dealing with aerial vehicles with a very restricted payload.

Bandwidth may be a very important issue when working with MRS, the number of nodes, topics and the information exchanges can grow exponentially when the size of the MRS increases. Nevertheless, in most applications each robot relies on an on board computer. This computer usually hosts the nodes publishing most of the large data, which is usually raw sensor data such as: laser, cameras, odometers, inertial measurements, etc. Moreover, the same computer also hosts most of the nodes that use that information (different control levels, navigation, etc.). This means that higher bandwidth is required inside the robot whereas among the robots the communications may be less intensive, only sending high level commands and reduced telemetry or mission status reports. This leads to the necessity of designing a good software architecture so as to keep the large data exchange inside the robot, reduce the amount of communications between the robots and therefore require less bandwidth from the communication systems.

Considering the previous example, each robot is endowed with an on board wireless router. An additional router is used in the control station. This solution allows isolating the internal and external communication of each robot. In order to communicate the routers among them, a solution based on wireless distribution system (WDS) has been chosen. WDS allows a wireless network to be expanded using multiple access points without the traditional requirement for a wired backbone to link them. Another notable advantage of WDS over other solutions is that it preserves the MAC addresses of clients across linked access points.

This solution not only enables the communication among the robots and control station but also extends the range of the communications using the routers on the robots as relays. Nevertheless, it introduces some restrictions. Thus, the maximum wireless effective throughput may be halved after the first retransmission (hop) was made, although dual band/radio APs may avoid this problem, by connecting to clients on one band/radio, and making a WDS network link with the other. Moreover, dynamically assigned and rotated encryption keys are usually not supported in a WDS connection. This means that dynamic Wi-Fi Protected Access (WPA) and other dynamic key assignment technology in most cases cannot be used, though WPA using pre-shared keys is possible. This lack of standardization lead to a strong recommendation of using the same, or very similar, routers models in every robot.

4.1 Relationship with Multi-robot Classification

It is not an easy task to correlate the feasible communication technologies used and the type of multi-robot in which it will be applied. In principle, any type of communication could be used for any type of multi-robot system. However, some considerations can be made. It should be remarked that, the choice of which communication technology to use is highly dependent on the application, moreover, possible selections according to the scenario or application were described above.

Clearly Wi-Fi is the most flexible solution, due to its characteristics and cost, it can be used for any size or configuration of MRS. Wimax, due to its high cost, does not seem as a good selection for swarms, but can be used in both two and multi-unit systems, mainly in large scenarios where there is a line of sight between all the components of the MRS. Furthermore, since this technology allows high bandwidth data transmission, it can be used for any organization of MRS. LTE on the other hand, because of its point-multipoint, network based nature, is recommended for very large scenarios, where there is no line of sight between the elements or when other technologies can not be used. Depending on the configuration LTE can also be used for two-unit and also for multi-unit systems, but in no case is it a good option for swarms. Finally ZigBee, can be used for two or multi-unit systems only if the amount of data transmitted is very low, however, due to its low cost, it may be a good choice for swarms, and distributed systems when they are used in indoors scenarios, and where the amount of data exchanged by each robot is usually low.

5 A Brief Review of ROS Multi-master Packages

In a first approach, using multi-master architectures in order to implement MRS in ROS is not completely necessary. A single Master node can store and manage the information of active nodes, topics, and services as well as the configuration parameters among several interconnected robots. Correctly defining, naming and including name-spaces for all components of the ROS computing graph can be enough to maintain the structure required in several simulated and real MRS applications. This may be the case for applications where the MRS is composed by robots that do not have high complexity, or on board computing power, and therefore each one of the robots have a small number of nodes and the communications are ensured at all times with a good bandwidth. When these conditions are not met, some issues could lead to the malfunction of the MRS; accordingly, the probability of failure grows exponentially when the MRS has an increasing number of robots.

MRS applied to field robotics have usually heterogeneous and complex robots. Moreover, they are used in large scenarios where communications can not be completely guaranteed. In addition to this, due to the conditions of environment and the task assigned to each robot, there may be a situation where one of the members is out of range or even has a failure. This situation, in combination with all those previously mentioned, makes it necessary to outline more robust and efficient approach for handling MRS.

The main issue will be how to preserve the capabilities and services offered by the ROS master and at the same time avoid the weaknesses of a centralized management. It is in this situation where the multi-master approach arises as a natural solution. Each robot in the MRS can keep its own ROS master and at the same time it can also exchange information with other components of the MRS.

This section presents a brief overview about some tools that can be used for implementing MRS using the multi-master approach in ROS. Since early ROS dis-

tributions, the multi-master problem has been addressed by the ROS community. Nevertheless it is a not well explored area due to the large infrastructure required, the amount of equipment needed and the incoming difficulties related to set-up a big number of robots for MRS testing.

Many of the proposed multi-master approaches have taken advantage of the feedback provided by the community as well as the concepts, technologies and functionalities provided by ROS as they evolve with each release of a new distribution. For instance, *multimaster_experimental*³ is a set of tools that enable the communication among different masters through a “foreign_relay” node. This node subscribes to a topic in the foreign master and publishes it in the local one, where the data can be locally distributed among multiple subscribers without increasing bandwidth consumption across the MRS. The main drawback of this package comes from its static configuration and the strong restriction of only one topic to be relayed.

The *multimaster*⁴ package goes a step further allowing to register multiple local topics and services into a foreign master and vice-versa. Its main difficulty is that the remote master needs to be manually addressed before launching the package, and it may cause a high load network communications due to the constant communication among foreign nodes. The *socrob_multicast*⁵ provides a multicast library with series of functionalities that uses the Adaptive Time Division Multiple Access (Adaptive-TDMA) scheme to enable the master communication between multiple masters within the robot network. In *wifi_comm*⁶ a MRS communication library is developed. This library is based on the link state routing protocol OLSR and the foreign_relay node from the multimaster_experimental package. Unlike its predecessor, it offers a discovery method to get information related to available neighbours and easy functionalities to open and closed “foreign_relays” among different masters. However, those packages are no longer maintained nor supported and therefore they are not recommended for new ROS users. Its usage should be restricted to systems that are not able to migrate to newer ROS distributions.

Nowadays, there are three packages that have taken the best of previous solutions and developed efficient multi-master architectures for MRS which continues to be active and available: *adhoc_communication*,⁷ *multimaster_fkie*⁸ and *rocon_multimaster*.⁹ Each one of three currently available packages will be described in the remainder of this section.

Table 2 summarizes the multi-master packages available for each ROS distribution and Table 3 shows the status for each one of them. It should be noted that the status of each package was determined taking into account the activity in their repositories as well as the information included in their Ros Wiki at the time of writing this chapter.

³http://wiki.ros.org/multimaster_experimental.

⁴<http://wiki.ros.org/multimaster>.

⁵http://wiki.ros.org/socrob_multicast.

⁶http://wiki.ros.org/wifi_comm.

⁷http://wiki.ros.org/adhoc_communication.

⁸http://wiki.ros.org/multimaster_fkie.

⁹http://wiki.ros.org/rocon_multimaster.

Table 2 Multi-master packages in ROS distributions

Package/distribution	Kinect	Jade	Indigo	Hydro	Groovy	Fuerte	Electric
multimaster_experimental					X	X	X
multimaster						X	X
socrob_multicast						X	
wifi_comm						X	
adhoc_communication			X	X			
multimaster_fkic		X	X	X	X	X	X
rocon_multimaster			X	X	X	X	

Table 3 Status of multi-master packages: (N) None, (P) Poor, (O) Normal, (H) High

Package	Active	Documented	Tutorials
multimaster_experimental	N	N	N
multimaster	N	P	N
socrob_multicast	N	O	N
wifi_comm	N	O	O
adhoc_communication	P	O	N
multimaster_fkic	H	H	H
rocon_multimaster	O	H	H

5.1 adhoc_communication

This package is based on the idea of using ad hoc networks as well as ad hoc routing protocols in order to establish communications among different ROS masters. This configuration allows to achieve communication with agents located outside the immediate neighbourhood (local network). This topology provides a good flexibility when working with MRS networks that are constantly changing their topology. Moreover, this solution offers three suitable levels of information exchange: unicast, multicast and broadcast. In unicast level (one-to-one), the data is transmitted directly and only to a destination robot. In multicast level (one-to-many), every robot has its own multicast group and other robots can join it, then the data is sent to all members of the group. Finally, in broadcast level the data is spread throughout the whole network.

Nevertheless, this package has critical drawbacks related to its implementation in the ROS architecture. First, the communication among masters is acquired by stand-alone interfacing through raw sockets implementations. For this reason, the node needs to be executed with super user privileges, which is not desirable, even less for new ROS users. Furthermore, the solution is implemented at user level instead of kernel level. Thus, the bandwidth is limited and it is not as fast as TCP/IP approaches. Finally, the package has no full compatibility with all ROS messages.

If the type of message that is required is not implemented yet, the user must adapt the communication protocol and then build the message by himself.

5.2 *multimaster_fkie*

The *multimaster_fkie* is a fully compatible multi-master implementation for topic and services transactions. The most remarkable feature of this package relies in its simplicity. This means that all ROS architecture run unmodified, moreover, no special API or libraries need to be linked in order to start the MRS communications. The robots can be configured as independent agents and the multi-master extension can be turned on or off without any advertising to the rest of the system. Furthermore, for interested users, the package provides a GUI for topic and service management among all masters in the network. Therefore, this solution is an easy and fast way to configure a multi-master, and it includes the capability of selecting which topics and services should be shared and to whom they will be shared with.

The architecture of this package is based on the implementation of two additional nodes: *master discovery* and *master synchronization*. They can be configured in unicast, multicast and zeroconf models (i.e. protocol for automatic service discovering in a network). Multicast is the most recommended due to the possibility of distributing the MRS in smaller groups of robots that only share the information required by the sub-group. First, the *master discovery* node is continuously scanning and advertising to the group about its local master state while it is receiving the same information from other reachable masters in the network. When a state change is advertised by the *master discovery*, the *master synchronization* node is triggered and the local master requests the information to the foreign node and synchronizes it into its own topic or service.

This implementation has also some drawbacks caused by the continuous master state scanning and the delay between changes in advertising as well as information exchange. High rates for fast topic synchronization could lead to a significant load for the local master and increasing the synchronization issue derived from the delay. Furthermore, the package does not involve any namespace handling. This means that if there are nodes, topics or services with the same name, even if they are in different robots, some problems or malfunctioning of the multi-master system can appear. For that reason this package is useful for developments that require an easy plug-and-play solution although it may not offer a high level of robustness.

5.3 *rocon_multimaster*

The Robotics in Concert project could be one of the most ambitious and complete solutions for MRS in ROS. The *rocon* stack has a great number of tools that addresses problems related to high level orchestration (i.e., management of several kind of

heterogeneous services) in MRS. It is conceived as a multi-robot framework running on the top of the interactions aiming to establish a centralized workspace to coordinate a robotised solution with features as wireless connectivity, multi-service handling, robot scheduling, software sharing and, lately, human interactions too.

Due to the high abstraction level in the *rocon* project, its multi-master approach named *rocon_multimaster* is not as simple as the *multimaster_fkie* package. The *rocon* solution is based on a gateway model for information exchange and hubs to coordinate MRS. Instead of communicating all the masters, they are organized in groups with a central hub where the gateway is located. Hence, a more organized and secure architecture is achieved keeping large information trespassing inside the hub, but with the possibility of communicating to other instances inside other hubs or in the user level. Also, this package has implemented the automatic service discovery through zeroconf. The main drawback behind this project is caused by its complexity. Since this solution is intended to be used in large scale implementations, they require several steps of parametrization and configuration before fully activating the multi-master system. Hence, this package is more useful for large scale projects with time to set-up a more robust multi-master system.

6 Example Applications for MRS and ROS

This section presents a series of ROS real world applications of MRS. In each case a small description of the problem addressed is presented. Also, the proposed solution, including the type of MRS used and a simplified schema of the software architecture used in each case are also presented. Three different applications are presented, the first one is based only on ground robots, and is used for search and rescue missions. The second one is an air-ground heterogeneous system used for monitoring environmental variables in green houses and the last one is composed only of air robots performing aerial surveys, photography and mosaicking which is intended for crop monitoring and weeding tasks.

6.1 MRS for Search and Rescue

The use of robots for search and rescue tasks has acquired importance after the attack on the World Trade Centre in 2001. Since then, there have been a total of 34 interventions that have performed this type of robots in three different domains: land, sea and air [24]. Using robots in this scenarios can provide a remote presence for the rescue teams, they can reach much further than the 3 to 4 meters a camera wand or telescopic mono-pod can offer. Moreover, they can offer important information about the situation in places where there is not enough space or where it is not safe enough for humans or dogs.

There are thirteen types of missions for search and rescue robots: search, reconnaissance and mapping, rubble removal, structure inspection, in situ medical assessment and intervention, medically sensitive extrication and evacuation of casualties, acting as a mobile beacon or repeater, serving as a surrogate for a team member, adaptive shoring, providing logistic support, victim recovery, estimation of debris volume and types and direct intervention. Until the year 2014, the robots deployed in search and rescue tasks did not have the capabilities for mapping the scenario or applying computer vision. In fact, only four of these robots had enough autonomy for performing waypoint navigation.

Rescue robots must be small enough and robust to work in extreme terrains and under hard conditions. UGVs (Unmanned Ground Vehicles) must move through irregular voids, work in small openings and sometimes operate in extreme heat or explosive atmospheres. UMVs (Unmanned Marine Vehicles) may have to work in marine currents avoiding floating debris. UAVs (Unmanned Air Vehicles) may have to overcome unpredictable buffeting and wind shears.

Also, rescue robots typically work in GPS and wireless denied environments. The material density of the buildings interferes in GPS and other signals used by the robots that work inside them. The power of wireless communications can be increased, but this leads to making the robots larger to carry the power devices, and they can become too large to be used in these scenarios.

Problem Description. The system presented in this section results from the experience of the SARRUS Team in the euRathlon 2015 challenge. Some of the authors were part of SARRUS Team and are specialized in search and rescue tasks.

The main goal of euRathlon 2015 challenge was to encourage the development of robotic solutions for disaster support in the sea, air and land domains. Consequently, the organizers proposed a series of challenges in realistic scenarios, in order to evaluate the performance of the unmanned vehicles for every domain. SARRUS team took part in the two single domain land trials defined by the organization:

- Land Trial (L1): Reconnaissance in urban structure.
- Land Trial (L2): Mobile manipulation.

Both of them were designed to evaluate four significant capabilities for carrying out rescue operations with UGVs:

- 2D Mapping: Ability to generate a digital representation of the environment that can be used in other tasks.
- Object Recognition: Perception, classification and localization of OPIs (Objects of Potential Interest).
- Object Manipulation: Ability to manipulate objects.
- Obstacle Avoidance: Ability of the UGV to perform a task while avoiding collisions with static and dynamic obstacles.

The trials should be done with the highest autonomy possible and reducing human interventions to only exceptional cases. However, strong safety constraints should be maintained and the time slot is only 45 min. Furthermore, live information about



Fig. 3 Graphic description of trial L2

state, position and images should be transmitted to the control station. In the first trial, the robot must enter the building and find a safe path to the machine room. Also, the robot should create a map of the inside of the building and detect OPIs to determine whether or not the entrance is blocked. For the second trial, the robot should detect some pipes and inspect if they are closed or not. Then, open valves must be closed by the robot. Those pipes could be found both inside and outside the building.

Figure 3 shows a diagram of Land Trial L2.

Solution Description. As mentioned before, GPS signals and wireless communication may fail inside the buildings. Additionally, the humidity and salinity of environment can attenuate the wireless signals. Therefore, the communications during the challenge were far from optimal.

To solve the wireless communication issues, the SARRUS Team used a MRS composed of two homogeneous ground robots in both trials. This solution provides several advantages, different tasks can be allocated to each robot, but they can be exchanged if necessary, also one of the robots can provide external imagery of the situation of the other one while performing complex tasks. Moreover, the communications range can be extended by using one of the robots as a communication relay. Also, the information of sensors on both robots makes it possible to build better maps of the environment.

The robots chosen for the search and rescue tasks are based on the Summit XL® robotic platform by Robotnik®. They have skid-steering kinematics based on four high efficiency motors. The robots can move autonomously or can be teleoperated



Fig. 4 The Summit XL®robotic platform with the sensors

by using video feed from an on-board camera. Furthermore, they are equipped with a small embedded PC, which runs the data processing and navigation algorithms autonomously.

The robots have multiple sensors. The odometry is provided by the encoders of the wheels and a high precision angular sensor assembled inside the chassis. A Hokuyo UTM-30LX-EW laser range finder is installed on the platform, it can scan a 270° semicircular field, with a guaranteed range from 0.1 to 30 m and a maximum output frequency of 40 Hz. It is placed in the central part of the robot at 60 cm over the ground. A Novatel OEM-4 GPS engine is also used; it offers a position accuracy of centimeters and a measurement frequency of 2 Hz. RS232 serial communication is used to read the incoming data and send correction commands. The engine is complemented with an ANT-A72GOLA-TW GPS antenna. A MicroStrain 3DM-GX3 25 high-performance miniature Attitude Heading Reference System (AHRS) is mounted inside the robot. It combines accelerometers, gyroscopes and magnetometers in the three axes, with temperature sensors and an embedded processor to provide orientation and angular velocity, as well as inertial measurements. Additionally, a Pan-Tilt-Zoom camera (PTZ) is placed in the front of the robot and provides video in real-time.

An image of the one of the robots with all the aforementioned equipment mounted is shown in Fig. 4.

Figure 5 shows the multi-master software architecture developed for this scenario. The base station is used for visualization and control purposes, whereas the main modules, are executed on the robots. Moreover, autonomous navigation or other high level tasks are executed on board each robot.

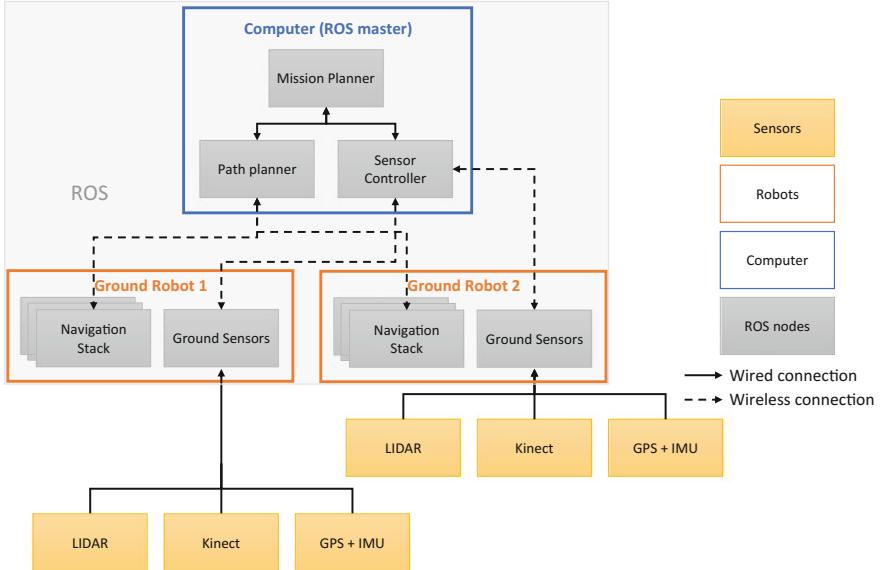


Fig. 5 Software architecture of the MRS for search and rescue

The modularity of the architecture, allows changing any module. Moreover, the multi-master schema facilitates the execution of the mission and reduces the required bandwidth for communications. Furthermore, a main-master runs on the base station and there is also one secondary master on each robot.

This configuration also increases the robustness of the solution, because all the processes or ROS nodes of each robot are controlled by a ROS-master running on the same computer, and can be more easily supervised or re-spawned in case of failure.

The central computer allows for the sending of commands to the robots (move, change between master and slave mode, cancel mission and take picture). Additionally, it is used to supervise the state of the mission (each robot's position, detections, etc.) and it can obtain the image feed from any of the robots. The additional computers are used to launch all the processes that run on board the UGV (i.e. drivers, localization, navigation, etc.). Moreover, those commands and the feedback are sent and received to/from each robot independently, thus enhancing the robustness of the complete system.

Communications. A base station was designed and implemented in order to control and monitor the execution of the mission. Its main component is the operator GUI, and it also includes three additional components, used to remotely monitor each robot in the fleet, and if needed, take control or stop it. The monitor stations can run on the same computer as the main GUI or they can be executed on different machines. Also, an omnidirectional high gain antenna (17 dB) was placed at the control station in order to increase range and bandwidth of the communications.

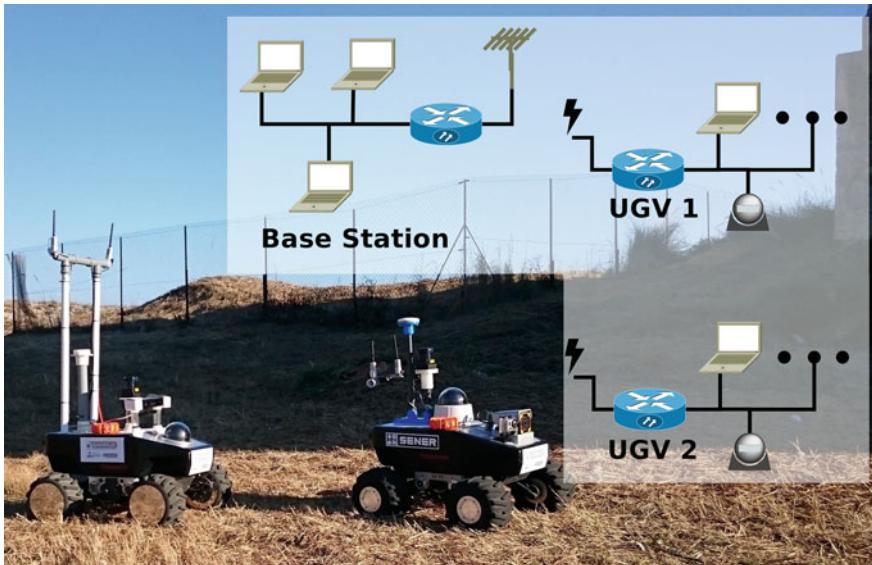


Fig. 6 Communications set-up. There is an internal network on each robot, and another one on the base station. Also, each sub-network has its corresponding ROS Master. The connection between the base station and the UGVs is done using *Wi-Fi*. The background image shows the robots during a test on euRathlon 2015

An internal Local Area Network (LAN) runs on each robot in order to speed-up the communications. The on-board computer and all the other Ethernet-based devices (e.g. Laser scanner, Ethernet cameras, etc.) are connected to this LAN. In order to communicate with the base station, a wireless router is installed into each robot. Those routers are configured, using WDS protocols, as wireless clients of a higher level WAN in which the base station is found.

Additionally the routers on board the robots can serve as signal relays. By doing so, the robots that are further away from the base station can still be reached. Furthermore, a ROS multi-master architecture has been used in order to minimize the bandwidth. This configuration helps to reduce the effects of delay or package losses usually present in *Wi-Fi* networks. In addition, two amplifiers of 9 dB were installed in each router with an unitary gain antenna.

On the base station side, all the computers are connected using Ethernet cables to a wireless router, that is enhanced with a directional antenna in order to provide coverage to the whole field. The antenna has a range of 200 m and an aperture of 120° and it is allocated next to the base station in one of the corners of the field. An image showing the architecture and its implementation is shown in Fig. 6.

For safety, a take over system was configured in each robot, just in case that all the networks fall down, the operator could control the robot manually and in a different bandwidth: FM 72 MHz.

Classification and Multi-master Configuration. The main multi-master characteristics required by the MRS for search and rescue can be listed as:

- An easy set-up procedure in order to be able for replacing some of the UGV's in a failure situation during the competition.
- A low computational load with the aim of avoiding to use resources needed by vision and Guidance, Navigation and Control (GNC) algorithms.
- Robustness against the entrance and leave of agents due to intermittent communications.

As it was detailed in the previous section, the most suitable package for this requirements was *multimaster_fkie*, so it was used to establish the information exchange between agents. Moreover, it has been successfully implemented in field robotics [4] for ROS-based MRS communication. Finally, the classification of this multi-robot system is as follows:

- **By size:** The system is a Two-unit system, and they have a static arrangement, because both robots are needed in order to complete the full mission. Moreover, in case of failure of one of them, the other robot was able to keep operating, but without the communication relay it will not be capable of entering the building and completing the mission.
- **By Morphology:** In this case the multi-robot system is Homogeneous because although both robots are from the same type, they are not equipped with the same sensors and actuators. Furthermore, as seen on Fig. 6, the configuration of the communication, and placement of the antennas, was different, having one repeater and one *end-point*.
- **By Level of coordination:** This system is Unaware, because most of the tasks and high level decisions were controlled by an operator. Moreover, as shown in Fig. 5, the base station computer receives and process the information from all robots.

6.2 MRS for Environmental Measurements

Another important application for MRS is environmental monitoring: not only for ground, marine or aerial vehicles, but also for heterogeneous fleets. The literature contains proposals about the acquisition of meteorological information [30], the control and monitoring of greenhouse gases [3] and contaminant clouds [34] among others.

The application described in this section is related to the use of an heterogeneous MRS in greenhouses. This is a promising application for MRS for various reasons: full availability all day and night, work under hazardous conditions and acquisition of complete and valuable information. The literature addresses some proposals for automating tasks related to climate control, crop surveillance, infestation and disease detection, planting and harvesting. The works described in this section, on the other hand, are focused on environmental monitoring as they address the measurement

of several variables of greenhouses: e.g. air temperature, air humidity, luminosity, ground temperature, ground humidity and carbon dioxide concentration. This task is very useful for controlling the crop conditions and the traceability of products.

The autonomous environmental monitoring systems for greenhouses can be divided in two categories: on the one hand, those based on Wireless Sensor Networks (WSNs) and, on the other hand, those based on mobile robots. The first group covers the majority of cases, but the second one is rising in recent years. The WSNs are efficient, modular and fault tolerant, but the motes are fixed and this fact limits their spatial resolution and may increase their costs. Meanwhile, the robotic systems can move in the greenhouse, place the sensors in the required locations and perform more “complex” tasks. However, the robots are not capable of taking measures simultaneously at various points, therefore the time rate of the measurements is limited. Moreover, the difficulties inherent to any mobile and complex system should also be considered.

Description of the Solution. The MRS application presented is an heterogeneous Two-unit MRS that results from the combination of two systems developed separately for this task. The first one is based on an air robot [26] whereas the second uses a ground robot [28]. It is clear that the ground robot is the best alternative to measure the ground properties, because it can access easier points of interest (e.g. it can dig holes, deploy sensors, take measures and collect samples). Additionally, this robot is more robust and has much more battery endurance allowing it to perform a continuous operation. The aerial robot, on the other hand, is a better choice for measuring the air variables, the reason for this is that it can reach practically any point of three-dimensional space. Moreover, it provides an ability of moving and taking measures over and between the plants. The air-ground system was developed by means of incremental steps: i.e. the development of new components independently and their integration in the whole system. In the referred multi-robot sensory system, the aerial platform was developed first, the ground later and the integration of them was the final step.

The ROS architecture of this multi-robot sensory system is shown in Fig. 7. As shown, a WLAN is required to connect all the robots and devices, while a central computer is used to collect, process and store the environmental data. During the experiments, a single ROS master was executed in this computer, where the mission is monitored and controlled, because the aerial robot used does not have the possibility of running its own ROS master, therefore a multi-master architecture will not be a good choice in this case.

The UAV employed as aerial sensor platform is a Parrot AR.Drone 2.0 (see Fig. 8a). This quad-rotor has a size of $525 \times 515 \times 120$ mm and a weight between 0.38 and 0.42 kg depending on the hull. The battery endurance changes from 12 to 24 min according to the chosen battery. It has an embedded computer with an autopilot controller and a router to generate or connect to Wi-Fi networks.

This UAV is connected to the WLAN and controlled by the central computer. In order to control and monitor the UAV, the *ardrone autonomy*¹⁰ package was used. It

¹⁰http://wiki.ros.org/ardrone_autonomy.

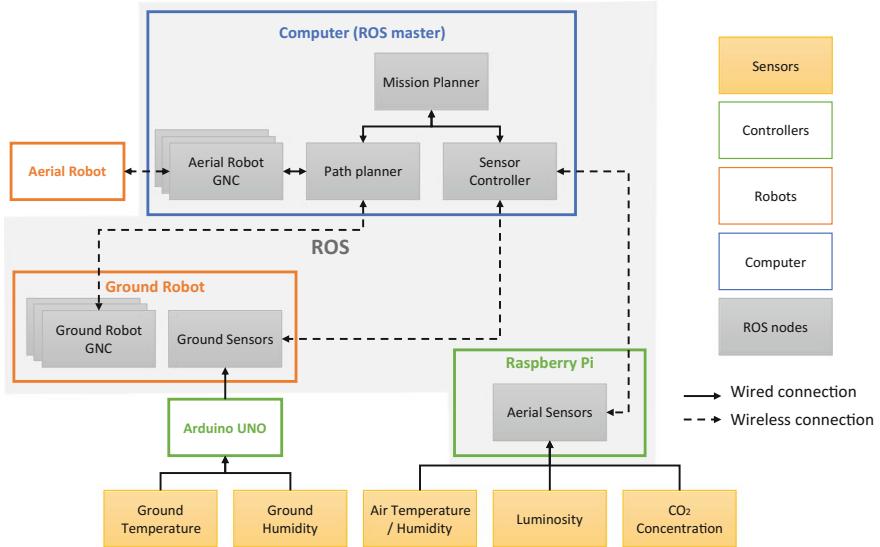


Fig. 7 Software architecture of the MRS for environmental measurements

is a widely used ROS driver for AR.Drone 1.0 and 2.0 quad-copters. This package allows for the control of the movements by means of speed commands and provides full telemetry: e.g. position, orientation, speed, acceleration, altitude, battery level, motor power as well as the image feedback from the cameras on board the UAV and tum ardrone [10].

The following sensors are carried by the UAV:

- Air temperature and humidity sensor: RHT03.
- Luminosity sensor: TSL2561.
- Carbon dioxide concentration: MG811.

The integration of these sensors in the system was carried out by using a Raspberry Pi model B+. This computer works with Raspbian Wheezy and ROS Hydro. A ROS node written in Python was designed in order to read the data of sensors. It should be pointed out that in order to manage physical pins of Raspberry Pi it is necessary either to assign super user privileges to the program or modify the port's permissions to allow a normal users to access them. The operation of the node is quite simple and works as follows: First of all, it configures the controller pins, defines the parameters of sensors and creates the publishers for the readings. Then, the code starts a loop with frequency of 1 Hz, where the measures of the sensors are collected and published.

The UGV employed as ground sensor platform is a Robotnik Summit XL (see Fig. 8b). The manufacturer provides low level control and teleoperate ROS packages for this robot. The autonomous navigation is based on the *move_base* package which is the standard ROS navigation schema described in previous sections.

The following sensors are carried by the UGV:



(a) Air Robot

(b) Ground Robot

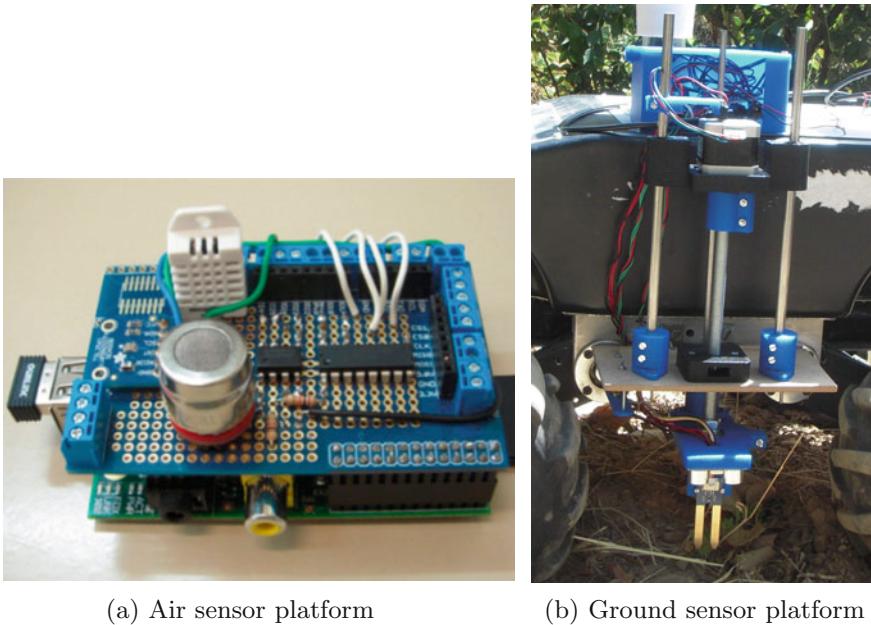
Fig. 8 Parrot AR.Drone 2.0 (Fig. 8a) and Robotnik Summit XL (Fig. 8b) with the sensors in agricultural environments

- Ground temperature (measured at distance): MLX90614.
- Ground humidity (measured in contact): SEN92355P.

The integration of these sensors in the system is performed through an Arduino UNO. This micro-controller is connected via USB to the computer of Summit XL and integrated in the ROS architecture by using rosserial package. The MLX90614 sensor is connected to the controller via I2C and uses 5 V, ground, SDA and SCL pins. Meanwhile, the SEN92355P sensor is connected to 5 V, ground, and analogue input pins.

A ROS node was written in Arduino environment and C language. In order to get the readings of the sensors and publish them as ROS topics. This program is also straight forward, it initializes the pins, the serial port and the ROS publishers in the set-up function. Then, in the loop function, the program controls the deployment of the moisture sensor in the soil, collects and publishes measures of ground temperature and humidity, and controls the collecting of the moisture sensor. Each loop of the program spends around 40 s to obtain a measure, mainly because of the time needed for deploying and collecting the sensors.

As mentioned before and shown in Fig. 9, two alternatives for integrating the sensors in the multi-robot sensory system have been studied and implemented: use an Arduino UNO in the ground robot and a Raspberry Pi in the aerial robot. This election is not arbitrary, but is based on our experience working with micro-controllers, sensors and actuators under the ROS environment. In the ground robot, the Arduino UNO is a powerful choice, because the main works can be performed by the robot computer reducing the costs of the system. Additionally, the integration of Arduino UNO in ROS architectures as a node is easy. On the other hand, in the aerial robot, the Raspberry Pi is the a suitable choice, because it is able to perform all the works without the support of robot computer. It should be considered that the robot controller has a limited performance and does not work under ROS, which advises against the direct integration of sensors.



(a) Air sensor platform

(b) Ground sensor platform

Fig. 9 Air sensor platform with Raspberry Pi, RHT03 temperature and humidity sensor, TSL2561 luminosity sensor and MG811 carbon dioxide sensor (Fig. 9a), and ground sensor platform with Arduino UNO, MLX90614 distance temperature sensor, SEN92355P contact moisture sensor and the mechanism for deploying and collecting them (Fig. 9b)

In general, Raspberry Pi presents some advantages over Arduino controllers, such as its connectivity and direct integration with ROS. Regarding the connectivity, Raspberry Pi can connect to WLANs by means of a simple Wi-Fi/USB module (in fact, the recent Raspberry Pi 3 has internal modules for Wi-Fi and Bluetooth), while Arduino UNO requires external Wi-Fi, Bluetooth or ZigBee shields. Furthermore, considering the integration with ROS, the Raspberry is able to run a Linux operating system (e.g. Raspbian and, in recent models, Ubuntu) with a ROS installation, but the Arduino UNO is only able to behave as a ROS node by using the serial port and a host node. However, Arduino may satisfy the requirements with more efficiency than Raspberry Pi in certain scenarios, mainly when direct handling of motors or other active sensors is involved, because power handling may be an issue and may cause unexpected reboots or failures. Moreover, timing and/or synchronization issues may appear when using high level operative systems.

These works show the potential of ROS in MRS for environmental monitoring applications. The main contribution of ROS in these scenarios is the ability to integrate and coordinate different robots to accomplish a common mission. Additionally, a wide range of sensors can be integrated in the architecture by means of controllers such as Raspberry Pi and Arduino. Finally, the measures of different sensors on board

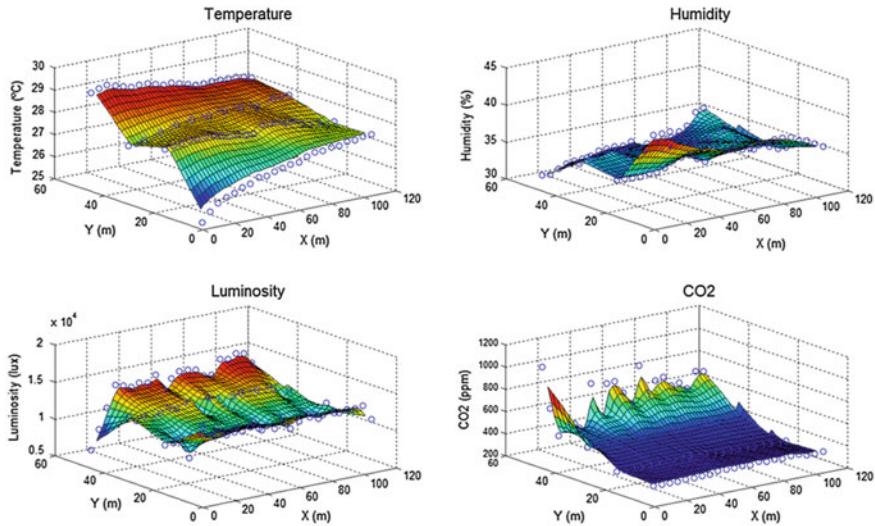


Fig. 10 The maps of temperature, humidity, luminosity and carbon dioxide concentration obtained with the aerial sensory system in our previous work [26]

different robots can be referred to certain locations and times, what is useful to build maps of environmental variables as the shown in Fig. 10.

Classification and Multi-master Configuration. For this use-case, as aforementioned, there is a single master ROS configuration, because the aerial robot does not have the capacity of running its master on board. The classification for this MRS is as follows:

- **By size:** As with the previous use-case, this MRS is a Two-unit system, and they have an static arrangement.
- **By Morphology:** This multi-robot system is Heterogeneous because it is composed by an aerial and a ground robot.
- **By Level of coordination:** This system is aware and weakly coordinated, because the aerial platform should react to the movements of the terrestrial robot.

6.3 MRS for Aerial Surveys

Nowadays small Unmanned Aerial Vehicles (UAVs) are employed in Precision Agriculture (PA) for crop observation, map generation through aerial surveys and some other related tasks. The traditional imagery services such as satellites, manned aircrafts or ground sensors were found unsuitable according to the ongoing requirements: Satellites images have limited resolution; the manned aircrafts are expensive; and the ground sensors are inefficient.

The small UAVs are a great promise due to the high availability and low cost. The maps are usually built by stitching a set of geo-referenced images through mosaicking procedures. These high-resolution images can detail out the information about biophysical and several other parameters of the crop field. Nevertheless, the aerial surveys are currently applied to other applications, for instance, archaeology [5], rangeland monitoring [25], among others.

Description of the Problem. One of the most researched tasks in agriculture is the localization and removal of undesired plants that may grow and spread out in crop fields. This weeding task is usually accomplished with local chemical inputs delivered by ground machines. The precise geo-referenced location of the undesired weed is obtained from high resolution images generated through mosaicking procedures after the aerial surveys.

The problem addressed here is how to survey a wide area by using small team of UAVs with limited battery endurance. Therefore, the mission must be accomplished in the shortest amount of time. The aerial survey is usually subject to a set of workspace restrictions, such as the take-off and landing positions as well as a safety distance between elements of the fleet. Moreover, it has to avoid no-fly zones.

Description of the Solution. The solution is based on the use of a multi-unit identical MRS, composed of several small UAVs in combination with an optimization algorithm for multi-robot coverage path planning (CPP) [32].

The system integrates different technologies in order to provide users with an embedded and fully capable tool. It has been made up of in a set of elements interconnected among them: air robots, payload and communication resources, as well as other avionics.

The type of aerial vehicle chosen was a quad-rotor. Aerial outdoor missions set up several requirements for the drone. For example, the camera (and lenses) should fit with the vehicle's plate-holder, considering also the security of the assembly. It implies not only an adequate mechanical conception but also the redistribution of the engine power, that should be able to hold up the additional weight. At the same time, this power should be managed in such a way that maintains the vehicle as much stable as possible. Besides, the adequate image acquisition clearly depends on the frame's steadiness and vibration reduction, that should be also provided by the correct engine distribution, control and election.

The solution proposed is based on the use of Ascending technologies (ASCTEC) quad-rotors because it was found that their UAVs cover most of the previously defined requisites. Moreover, the ROS packages of drivers and controllers for those robots have a very good state of maturity. The ASCTEC drivers available online¹¹ concern to the low-level system of those systems. A complete and comprehensive aerial system can be designed from the scratch based on those packages taking advantage of the base navigation functionalities. The aerial platforms from the aerial survey fleet are depicted in Fig. 11. For further information about aerial surveys carry out, please refer to the work of Valente *et al.* (2013) [33].

¹¹http://wiki.ros.org/asctec_drivers.



Fig. 11 The ASCTEC UAVs used for the aerial surveys: The hummingbird and the pelican models

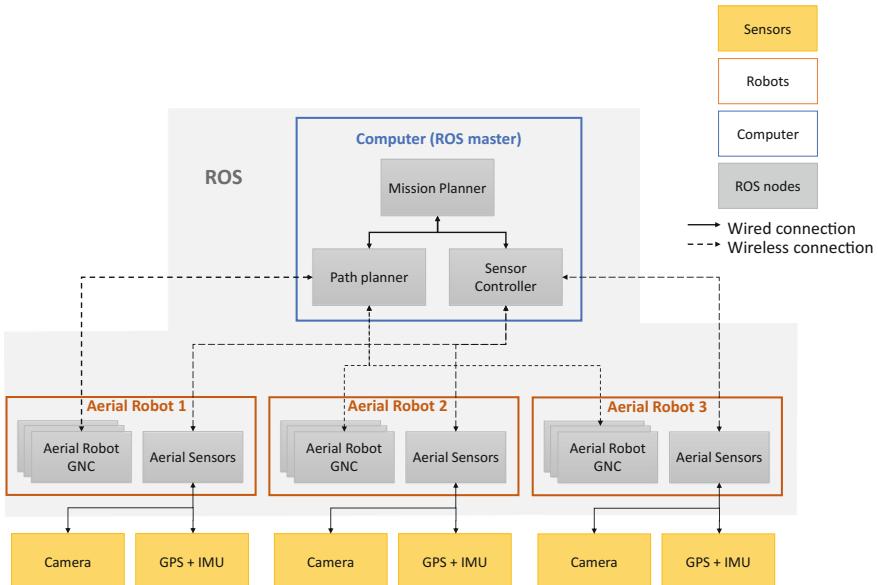


Fig. 12 Software architecture of the MRS for aerial surveys

As was done with the other applications presented in this section, a ROS-based node architecture was designed in order to achieve a fully functional MRS aerial remote sensing system. The node architecture design is shown in Fig. 12.

There is a central computer (in the base station) that works as master node. Each aerial robot connects to the central computer via a ZigBee channel allocated on the 2.4 GHz wireless frequency band. This dual-band channel enables data from/to the aerial robots, e.g., receives navigation data, sends way-points. The images can be acquired and stored on board (high resolution) or remotely (low resolution). Onboard storing of the images means that the images are saved to an internal SD card, whereas remotely stored means that they are forwarded to the central computer using a Wi-Fi connection.

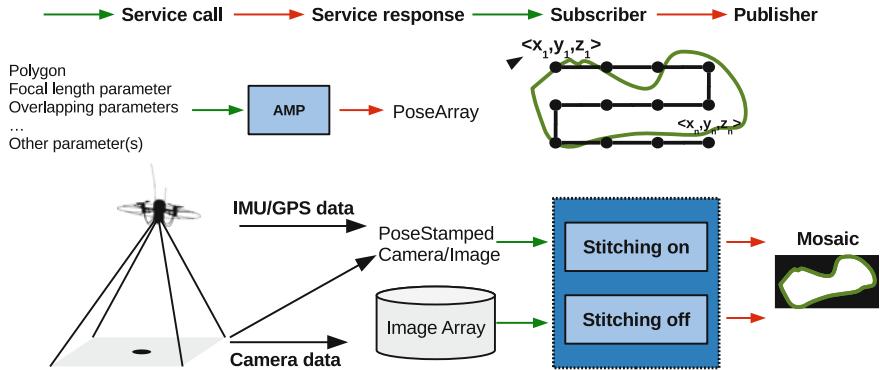


Fig. 13 The High-level ROS nodes from the aerial system are indicated by a light blue rectangle. The nodes are AMP (Aerial Mission Planner), Stitching on (image mosaicking in real time), Stitching off (image mosaicking pos-flight)

The designed solution is reliable enough to carry out the aerial survey with a team of several aerial robots. Nevertheless, if it is possible to increase the coordination level of the system by using a multi-master ROS system. This will allow the system to continue working even if there is a sudden peer-to-peer loss in communication, because the aerial robot behaving as master node will be able to replace another robot in the mission. If there is a communication failure, the aerial robot involved will fly back to its starting position and land.

Finally, the ROS-based node architecture is completed with two high-level nodes: An Aerial Mission Planner (AMP) or Mission planner in Fig. 12 and the Mosaicking nodes, i.e., Stitching on-line and off-line. (See Fig. 13). It can be noticed that the communication between the AMP, the Stitching off-line nodes and the rest of the nodes is obtained through a ROS service. Since the aerial mission planner computes the aerial robots trajectories off-line there is no need to have a synchronous communication channel. On the other hand, a ROS publisher and subscriber communication strategy is used if real time stitching and visualization is required. Using this system, a high-resolution map is created through the following procedure: 1. The user introduces the field and mission parameters, and enables one of the two the Stitching modes; 2. AMP computes the coverage path and publishes a list of way-points. That list will be divided by the number of quadrotors; 3. Aerial survey: The image and the navigation data is published; 4. Image and navigation data subscription and a high-resolusion image is published in real-time or after successfully completing the mission.

Classification and Multi-master Configuration. As with the previous use-case, the aerial robots cannot run ROS nodes or execute a master ROS on board, therefore the solution uses a single ROS master configuration. The classification for this MRS is as follows:

- **By size:** This MRS is multi-unit system, composed by two or three UAVs and they have a coordinated re-arrangement, because in order to change the number of robots it is necessary to re-define the coverage trajectory for all of them.
- **By Morphology:** This multi-robot system is Identical, because all robots have the same capabilities, and they can be exchanged or replaced without limiting the full operability of the system.
- **By Level of coordination:** This system is aware and, strongly coordinated, and strongly centralized. Because all the planning is done in the central computer, and the robots only execute their given missions.

7 Lesson Learned and Issues to Overcome

As aforementioned, MRS can be used in a wide variety of applications. Therefore, there are several issues or difficulties specific for each application. Those may arise from the required task, the complexity of the systems used or the scenario where it should be performed. The issues regarding very specific tasks or scenarios have been left out of this discussion because they may differ a lot from each other and they are outside the scope of this chapter.

Nonetheless, there are some challenges that are common for most MRS applications. Basically, they are divided into several groups that will be described next.

Task Allocation and Coordination. One of the main characteristics of a MRS is that it should operate, and be considered as a single entity. This means that the autonomy of the system does not depend only on each robot but also on the group itself. There are two main task related to the group as a whole, sub-task allocation or distribution and group coordination or supervision. The first one is usually executed at the beginning of the mission while coordination and supervision are executed through all the mission.

One of the main difficulties of task allocation is that it should seek for an efficient use of all the resources in the fleet. Although some MRS require a human intervention in order to assign or modify the sub-task that each agent needs to accomplish. When this process is automated, the MRS increases its capabilities and therefore it can use their resources in a more efficient manner. However, the autonomous task allocation can be very complex and computationally expensive. Moreover, coordination and supervision are highly dependent on the task allocation, because when it is correctly executed it will result in less adjustments and therefore it facilitates the work of the supervisor, even if it is a human being and not an autonomous agent. Several different techniques have been developed to achieve this goal and they will not be described in this chapter. The reader can refer to the work of *Khamis et al. (2015)* [21] where the most important techniques to solve the task allocation issue are described.

Communications. Communications between all members of a MRS, including the non-mobile agents (e.g. base station) are a main requirement for the correct operation of any MRS. Some of the technologies used for this task are explained in detail in Sect. 4.

The difficulty regarding the communications is not only given by the schema or technology used, but on how to guarantee that the information will reach its destination or how the system will react when some losses to communications are present. To solve this, it is necessary to use techniques that provide robustness against possible packages losses or delays. An example of this is found in the work of *Hernandez et al. (2014)* [18] where each robot in a patrolling system updates its information when it reaches a given point. But, it can continue operating with its current state if it does not receive any new data, and updates its state as soon as it receives the incoming information. Moreover, other works have proposed package routing techniques that take into account range or bandwidth limitations in order to ensure that the data arrives at its destination [23].

Contrary to what might be thought, increasing the height of the antenna by using a mast is not the solution, since it introduces vibrations as well as losses on the cable that connects the antenna to router or amplifier. Furthermore, using high gain antennas on the mobile robots can also lead to lost of communication because those usually provide a narrow lobe angles which may be unable to reach the area of influence of the main antenna. Therefore, a low-gain omnidirectional antenna (but with a wide lobe angle in their radiation pattern) on board the robots turned out to provide better performance while working together with a directional high power antenna in the base station. If the base station is allocated in the centre of the scenario instead of the border, an omnidirectional one should be used but it will considerably affect the effective range.

Type of information exchanged. Another important factor when working with a MRS is in regard to the information that needs to be shared between members of the team. It is very important to correctly define whether raw or post-processed data may be shared. Also, communications can be unicast, broadcast or multicast. In general terms, it is always preferable to share the least amount of raw data possible. The reason for this is that, transmitting raw data not only increases the bandwidth requirements, but it will also introduces delays or other timing issues which can lead to instabilities in the control loops. This is even more evident in Wi-Fi networks, that are characterized by having relatively high latencies. However, the payload and computing capacities of the robots also need to be taken into account, because If processing the data on board will take longer than transmitting it, or if this computational load may cause failures in other tasks of the controller, then it obviously should be avoided.

Time Synchronization. One of the most common issues when working with different onboard computers and ROS masters is the time differences that may appear. Since usually during field tests the robots are not connected to the internet, GPS time synchronization is a very good choice when available, and it can be used in combination with a local network time protocol server installed in one of the computers of the MRS, this is much more evident when working with battery operated robots, and even more if multi-master approaches are used.

Standardizing Names and Namespaces. When working with MRS in real world test, the number of nodes being executed is usually very high. This means that

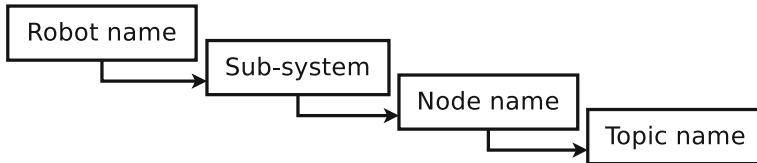


Fig. 14 Namespaces hierarchical structure

correctly naming the nodes and their respective parameters, services and topics may be critical. By standardizing the naming of the nodes and correctly using namespaces many launching errors can be avoided, and detecting failures and re-spawning nodes becomes a much more easy task. Even more, creating scripts for setting parameters as well checklists of the steps required to launch the complete system is highly recommended, as it will largely facilitate debugging tasks. For the different MRS presented on Sect. 6, the same namespace practice was used and it is depicted in Fig. 14.

The namespacing proposed has a hierarchical structure, with the robot name at the higher level, then the different sub-systems, such as localization or object recognition. The next level contains the names of the nodes and the lower level corresponds to the name of the topics. Furthermore, it is a good practice to try keeping lower elements of the structure (i.e. nodes and topics names) as common as possible, changing only the high level namespaces.

Autonomous Supervision. The autonomous supervision and alerting of the nodes and or topics that may have failed can be a very useful tool in order to improve the operator's awareness about the state of the fleet. This is a challenge that the ROS community is trying to solve, for example with the/statistics features introduced in ROS Indigo and the ARNI package¹² that uses and extends this information, comparing it against a set of reference values and allowing to run optional countermeasures when a deviation from the reference is detected. This functionalities can help working in large projects such as those presented in this chapter, where for instance, during many of the experiments it was found that one of the sensors stopped sending measurements, or that a node has failed and it took the operators a large amount of time to realize this situation.

8 Conclusions

The field experiments pointed some facts that are relevant for MRS. For instance, the number of ROS nodes running is usually very high. Therefore, it is necessary to define a software architecture. Moreover, the communication system is also the key issue for supporting any MRS application. Taking into account the communications when

¹²<http://wiki.ros.org/arni>.

defining the software architecture and vice-versa will render in a better performance of both of them.

Communications was perhaps the most recurrent issue in all the systems presented. Even though this has been widely studied, when working on real scenarios, there will be always some conditions that will slow down or cause failures in communications. To solve this, the MRS should be able to support short-term communication losses. If working on critical scenarios, backup and recovery systems should be included so as to prevent any damage to the equipment or the scenario where the task is being carried out.

Finally, it should be said that, although it is clear that by using MRS it is possible to speed up a given task or tackle more complicated ones, the difficulties and additional challenges that arose when using a MRS should be analysed. As shown in this chapter, the use of a fleet of robots considerably increases the required complexity of the communications and software architecture, and this additional issues should be taken into account when deciding whether or not use a MRS for any given mission.

References

1. Arai, T., E. Pagello, and L.E. Parker. 2002. Guest editorial advances in multirobot systems. *IEEE Transactions on Robotics and Automation* 18 (5): 655–661.
2. Barrientos, A., J. Colorado, J.d. Cerro, A. Martinez, C. Rossi, Sanz D, and Valente J. 2011. Aerial remote sensing in agriculture: A practical approach to area coverage and path planning for fleets of mini aerial robots. *Journal of Field Robotics* 28 (5): 667–689. doi:[10.1002/rob.20403](https://doi.org/10.1002/rob.20403).
3. Berman, E.S., M. Fladeland, J. Liem, R. Kolyer, and M. Gupta. 2012. Greenhouse gas analyzer for measurements of carbon dioxide, methane, and water vapor aboard an unmanned aerial vehicle. *Sensors and Actuators B: Chemical* 169: 128–135.
4. Brüggemann, B., D. Wildermuth, and F.E. Schneider. 2016. *Search and Retrieval of Human Casualties in Outdoor Environments with Unmanned Ground Systems—System Overview and Lessons Learned from ELROB 2014*, 533–546. Cham: Springer International Publishing. doi:[10.1007/978-3-319-27702-8_35](https://doi.org/10.1007/978-3-319-27702-8_35).
5. Brumana, R., D. Oreni, L. Van Hecke, L. Barazzetti, M. Previtali, F. Roncoroni, and R. Valente. 2013. Combined geometric and thermal analysis from uav platforms for archaeological heritage documentation. ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences II-5/W1, 49–54. <http://www.isprs-ann-photogramm-remote-sens-spatial-inf-sci.net/II-5-W1/49/2013/>.
6. Cai, Y., and S.X. Yang. 2012. A survey on multi-robot systems. *World Automation Congress (WAC)* 2012: 1–6.
7. Cao, Y.U., A.S. Fukunaga, and A. Kahng. 1997. Cooperative mobile robotics: Antecedents and directions. *Autonomous Robots* 4 (1): 7–27. doi:[10.1023/A:1008855018923](https://doi.org/10.1023/A:1008855018923).
8. Dorigo, M., D. Floreano, L.M. Gambardella, F. Mondada, S. Nolfi, T. Baaboura, M. Birattari, M. Bonani, M. Brambilla, A. Brutschy, et al. 2013. Swarmanoid: a novel concept for the study of heterogeneous robotic swarms. *IEEE Robotics & Automation Magazine* 20 (4): 60–71.
9. Dudek, G., M.R.M. Jenkin, E. Milios, and D. Wilkes. 1996. A taxonomy for multi-agent robotics. *Autonomous Robots* 3 (4): 375–397. doi:[10.1007/BF00240651](https://doi.org/10.1007/BF00240651).
10. Engel, J., J. Sturm, and D. Cremers. 2014. Scale-aware navigation of a low-cost quadrocopter with a monocular camera. *Robotics and Autonomous Systems* 62 (11): 1646–1656.

11. Farinelli, A., L. Iocchi, and D. Nardi. 2004. Multirobot systems: a classification focused on coordination. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)* 34 (5): 2015–2028.
12. Fu, Y., X. Wang, H. Zhu, and Y. Yu. 2016. Parameters optimization of multi-uav formation control method based on artificial physics. In *2016 35th Chinese Control Conference (CCC)*, 2614–2619.
13. Garzón-Ramos, D., M.G. Oviedo, and A. Barrientos. 2016. Protección multi-robot de infraestructuras: Un enfoque cooperativo para entornos con información limitada. In *XXXVII Jornadas de Automática*.
14. Garzón, M., J.a. Valente, J.J. Roldán, L. Cancar, A. Barrientos, and J. Del Cerro. 2015. A multirobot system for distributed area coverage and signal searching in large outdoor scenarios. *Journal of Field Robotics*, n/a-n/a. doi:[10.1002/rob.21636](https://doi.org/10.1002/rob.21636).
15. Garzón, M., J. Valente, D. Zapata, and A. Barrientos. 2013. An aerial-ground robotic system for navigation and obstacle mapping in large outdoor areas. *Sensors* 13 (1): 1247–1267. <http://www.mdpi.com/1424-8220/13/1/1247>.
16. Gautam, A., and S. Mohan. 2012. A review of research in multi-robot systems. In *2012 IEEE 7th International Conference on Industrial and Information Systems (ICIIS)*, 1–5.
17. Gerkey, B.P., and M.J. Matarić. 2004. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research* 23 (9): 939–954. <http://ijr.sagepub.com/content/23/9/939.abstract>.
18. Hernández, E., A. Barrientos, and J. del Cerro. 2014. Selective smooth fictitious play: An approach based on game theory for patrolling infrastructures with a multi-robot system. *Expert Systems with Applications* 41 (6): 2897–2913. <http://www.sciencedirect.com/science/article/pii/S0957417413008403>.
19. Hernández, E., J. del Cerro, and A. Barrientos. 2013. Game theory models for multi-robot patrolling of infrastructures. *International Journal of Advanced Robotic Systems* 10.
20. Iocchi, L., D. Nardi, and M. Salerno. 2001. Reactivity and Deliberation: A Survey on Multi-Robot Systems. *Balancing Reactivity and Social Deliberation in Multi-Agent Systems: From RoboCup to Real-World Applications*, 9–32. Berlin: Springer. doi:[10.1007/3-540-44568-4_2](https://doi.org/10.1007/3-540-44568-4_2).
21. Khamis, A., A. Hussein, and A. Elmogy. 2015. Multi-robot Task Allocation: A Review of the State-of-the-Art. *Cooperative Robots and Sensor Networks*, 31–51. Cham: Springer International Publishing. doi:[10.1007/978-3-319-18299-5_2](https://doi.org/10.1007/978-3-319-18299-5_2).
22. Mohammed, A., F. Stolzenburg, and U. Furbach. 2010. *Multi-robot systems: Modeling, specification, and model checking*. INTECH Open Access Publisher.
23. Mosteo, A.R., L. Montano, and M.G. Lagoudakis. 2008. Multi-robot routing under limited communication range. *IEEE International Conference on Robotics and Automation, ICRA 2008*, 1531–1536.
24. Murphy, R.R. 2014. *Disaster Robotics*. Cambridge: The MIT Press.
25. Rango, A., A. Laliberte, J.E. Herrick, C. Winters, K. Havstad, C. Steele, and D. Browning. 2009. Unmanned aerial vehicle-based remote sensing for rangeland assessment, monitoring, and management. *Journal of Applied Remote Sensing* 3: 033542.
26. Roldán, J.J., G. Joossen, D. Sanz, J. del Cerro, and A. Barrientos. 2015. Mini-uav based sensory system for measuring environmental variables in greenhouses. *Sensors* 15 (2): 3334–3350.
27. Roldán, J.J., P. García-Aunon, M. Garzón, J. de León, J. del Cerro, and A. Barrientos. 2016. Heterogeneous multi-robot system for mapping environmental variables of greenhouses. *Sensors* 16 (7): 1018. <http://www.mdpi.com/1424-8220/16/7/1018>.
28. Ruiz-Larrea, A., J.J. Roldán, M. Garzón, J. del Cerro, and A. Barrientos. 2016. A ugv approach to measure the ground properties of greenhouses. *Robot 2015: Second Iberian Robotics Conference*, 3–13. Springer.
29. Shaw, M., and D. Garlan. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Englewood Cliffs: Prentice Hall. Prentice Hall Ordering Information.
30. Spiess, T., J. Bange, M. Buschmann, and P. Vörsmann. 2007. First application of the meteorological mini-uav'm2av'. *Meteorologische Zeitschrift* 16 (2): 159–169.

31. Valente, J.a., D. Sanz, A. Barrientos, J.d. Cerro, A. Ribeiro, and C. Rossi. 2011. An air-ground wireless sensor network for crop monitoring. *Sensors* 11 (6): 6088. <http://www.mdpi.com/1424-8220/11/6/6088>.
32. Valente, J., J. Del Cerro, A. Barrientos, and D. Sanz. 2013. Aerial coverage optimization in precision agriculture management: A musical harmony inspired approach. *Computers and Electronics in Agriculture* 99: 153–159.
33. Valente, J., D. Sanz, J. Del Cerro, A. Barrientos, and M.Á. de Frutos. 2013. Near-optimal coverage trajectories for image mosaicing using a mini quad-rotor over irregular-shaped fields. *Precision Agriculture* 14 (1): 115–132.
34. White, B.A., A. Tsourdos, I. Ashokaraj, S. Subchan, and R. Źbikowski. 2008. Contaminant cloud boundary monitoring using network of uav sensors. *Sensors Journal, IEEE* 8 (10): 1681–1692.
35. Yim, M., W.M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G.S. Chirikjian. 2007. Modular self-reconfigurable robot systems [grand challenges of robotics]. *IEEE Robotics Automation Magazine* 14 (1): 43–52.

Author Biographies

Mario Garzón was born in Pasto, Colombia, in 1983. He is a researcher at the Robotics and Cybernetics research group at Centro de Automática y Robótica (CAR UPM-CSIC). He received the Electronics Engineer degree from Escuela Colombiana de Ingeniería in 2005 and both Msc and PhD degrees in Automation and Robotics from Universidad Politécnica de Madrid (UPM) in 2011 and 2016 respectively. He was also visiting researcher at INRIA (e-Motion team project) in Grenoble, France in the years 2013 and 2014.

His main research interest is field robotics, specifically on navigation and cooperation between heterogeneous mobile robots as well as the detection and prediction of pedestrian trajectories. He has also worked on Bio-Inspired robotics, computer vision, machine learning and data mining.

João Valente (Lisbon, Portugal, 1982) received is M.Sc. in Electrical and Computer Engineering from the New University of Lisbon in 2008, and both M.Sc and PhD. in Automation and Robotics from the Polytechnic University of Madrid, in 2011 and 2014, respectively. Moreover, from 2007 to 2008 he was a junior researcher in the Intelligent Systems for Emergencies and Civil defence Laboratory, from the University of Rome “La Sapienza”. He was a researcher in the Robotics and Cybernetics group from the CAR UPM-CSIC from 2008 to 2014. He was also visiting researcher in USA (2012), at the robotics and intelligent systems lab from the City College of New York, and in Belgium (2014), at IRIDIA from Universit'e Libre de Bruxelles. His research interests are field robotics, aerial robots, remote sensing, path and mission planning. Currently, he is visiting professor and researcher at the Universidad Carlos III de Madrid.

Juan Jesús Roldán (1988, Almería-Spain) studied BSc+MSc on Industrial Engineering (2006-2012) and MSc on Automation and Robotics (2013-2014) in Technical University of Madrid (UPM). He has researched about emergency detection and management in multirotors, surveillance of large fields with multiple robots and ground and aerial robots applied to environmental monitoring in greenhouses. Currently, he is a Ph.D. candidate of Centre for Automation and Robotics (CAR, UPM-CSIC) and Airbus Defence and Space, whose research is focused on the multi-UAV coordination and control interfaces.

David Garzón-Ramos was born in Pasto, Colombia in 1990. He received his Electronic Engineer degree from the Universidad Nacional de Colombia (UN) in 2014. Then, he carried out his master's studies on Automation and Robotics in the Universidad Politécnica de Madrid (UPM) and

Engineering - Industrial Automation in the Universidad Nacional de Colombia (UN) until 2016. His research activities started in 2011 as undergraduate research assistant in the Optical Properties of Materials (UN) research group. From 2014 to 2016 he did an internship in the Aerospace and Control Systems Section from SENER Ingeniería y Sistemas, S.A. In 2015 he joined to the Robotics and Cybernetics (UPM) group as post-graduate research assistant. Currently, he is a PhD candidate of Centre for Automation and Robotics (CAR, UPM-CSIC) and his main research interests (non-exhaustive) are aimed to Unmanned Ground Vehicles (UGVs), Multi-Robot systems (MRS) and Guidance, Navigation, and Control (GNC).

Jorge de León was born in Barcelona in 1988. He 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 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.

Antonio Barrientos received the MSc Engineer degree in Automatic and Electronic from the Polytechnic University of Madrid in 1982, and the PhD in Robotics by 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 on 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.

His main interests are in air and ground field robotics. He is author of several textbooks in Robotics and Manufacturing automation, and also is co-author of more than 100 scientific papers in journals and conferences. Ha has been director o co-director of more than 20 PhD thesis in the area of robotics. Currently he is the head of the Robotics and Cybernetics research group of the Centre for Automatic and Robotics in the Technical University of Madrid - Spanish National Research Council.

Jaime del Cerro received his Ph.D. degree in Robotics and Computer vision at Polytechnic University of Madrid at 2007. He currently teaches Robotics, Guidance, Navigation and Control of autonomous robots and system programming in this University and also collaborates at UNIR (Universidad International de la Rioja) in the master of Management of Technological projects. He has participated in several European Framework projects and projects funded by ESA (European Space Agency) and EDA (European Defense Agency) as well as commercial agreements with relevant national companies.

Part V

Perception and Sensing

Autonomous Navigation in a Warehouse with a Cognitive Micro Aerial Vehicle

Marius Beul, Nicola Krombach, Matthias Nieuwenhuisen,
David Droschel and Sven Behnke

Abstract Micro aerial vehicles (MAVs), such as multirotors, are envisioned for autonomous inventory-taking in large warehouses. Fully autonomous operation of MAVs in such complex 3D environments requires real-time state estimation, obstacle detection, mapping, and navigation planning. To this end, we employ a cognitive MAV equipped with multiple sensors including a dual 3D laser scanner, three stereo camera pairs, an IMU, an RFID reader, and a powerful onboard computer running the ROS middleware. Tasks with hard real-time requirements such as attitude control and state estimation are processed on a Pixhawk Autopilot, which communicates with the main computer via the MAVLink protocol. In this chapter, we describe our integrated system for autonomous MAV-based inventory in warehouses. We detail the involved components and evaluate our system with the real autonomous MAV in a realistic scenario. We also report lessons learned during field testing.

Keywords MAV · Multimodal sensor setup · 3D laser scanner · Sensor fusion · Autonomy

1 Introduction

Micro aerial vehicles (MAVs) are enjoying increasing popularity, both in research and in applications such as aerial photography, inspection, surveillance, and search and rescue missions. Most MAVs are remotely controlled by a human operator or follow global navigation satellite system (GNSS) waypoints in obstacle-free heights. For autonomous navigation in complex 3D environments, sufficient onboard sensors and computing power are needed in order to perceive and avoid obstacles, build 3D maps of the environment, and plan flight trajectories.

M. Beul (✉) · N. Krombach · M. Nieuwenhuisen · D. Droschel · S. Behnke
Autonomous Intelligent Systems Group, University of Bonn,
Friedrich-Ebert-Allee 144, 53113 Bonn, Germany
e-mail: mbeul@ais.uni-bonn.de
URL: <http://www.ais.uni-bonn.de>

In this chapter, we present a use case for indoor MAV operation employing the ROS infrastructure: autonomous warehouse inventory. For this purpose, we built an MAV with a multimodal omnidirectional sensor setup, a fast onboard computer, and a robust data link. The sensors include a lightweight dual 3D laser scanner, three stereo cameras, and a radio-frequency identification (RFID) reader module. All components are lightweight and hence well suited for MAVs. Our MAV can localize itself in indoor environments fusing visual odometry and 3D laser scan registration to a 3D map. It avoids static and dynamic obstacles perceived with the onboard sensors reliably.

On the MAV, we employ ROS Indigo Igloo as middleware on top of Ubuntu 14.04 to facilitate fast development through a modular software design. This allows us to transfer technology between multiple MAVs, e.g., built for outdoor mapping [6], and ground robots, e.g., in a space exploration scenario [30]. Furthermore, ROS allows for an easy and flexible connection with several ground control stations.

After a discussion of related work in the next section, we will briefly describe our MAV. Our perception pipeline is explained in Sect. 4, putting special emphasis on the cameras (Sect. 4.2) and laser scanner (Sect. 4.3). We then outline our mapping approach in Sect. 5 and describe the localization and state estimation capabilities in Sect. 6. The navigation pipeline is detailed in Sect. 7. Section 8 describes the user interfaces. The MAV system is experimentally evaluated in Sect. 9. We conclude the chapter with a discussion of lessons learned in Sect. 10.

2 Related Work

In recent years, many MAVs with onboard environment sensing and navigation planning have been developed. Due to the limited payload of MAVs, most approaches to obstacle avoidance are camera-based [8, 17, 21, 24, 26, 28, 29, 33]. Approaches using monocular cameras to detect obstacles require translational movement in order to perceive the same surface points from different perspectives. In order to estimate depth of object points instantaneously, stereo cameras are used on MAVs, e.g., in the works of Schmid et al. [29] and Park and Kim [24]. Tripathi et al. [33] use stereo cameras for reactive collision avoidance. The limited field of view (FoV) of cameras poses a problem when flying in constrained spaces where close obstacles can surround the MAV.

To overcome these limitations, some MAVs are equipped with multiple (stereo) cameras. Schauwecker and Zell [28] use two stereo cameras, one oriented forward, the other backward. Moore et al. [20] use a ring of small cameras to achieve an omnidirectional view in the horizontal plane, but rely on optical flow for velocity control, centering, and heading stabilization only.

Grzonka et al. [11] use a 2D laser scanner to localize the MAV in environments with structures in flight altitude and to avoid obstacles. This limits obstacle avoidance to the measurement plane of the laser scanner. Other groups combine laser scanners and visual obstacle detection [13, 14, 32]. Still, their perceptual field is limited to

the apex angle of the stereo camera (facing forward), and the mostly horizontal 2D measurement plane of the scanner. They do not perceive obstacles above or below this region or behind the vehicle. We allow omnidirectional 4D movements (3D position and yaw) of our MAV, thus we have to take obstacles in all directions into account. The proposed MAV extends our own previous work [6], an MAV with a 3D laser scanner and two wide-angle stereo camera pairs. Another MAV with a sensor setup that allows omnidirectional obstacle perception is described by Chambers et al. [3]. We significantly increase field of view and bandwidth of the onboard cameras, add a second laser scanner to measure simultaneously in orthogonal directions, and use a faster onboard computer.

In combination with accurate pose estimation, laser scanners are used to build 3D maps. Fossel et al. [9], for example, use Hector SLAM [15] for registering horizontal 2D laser scans and OctoMap [12] to build a three-dimensional occupancy model of the environment at the measured heights. Morris et al. [22] follow a similar approach and in addition use visual features to aid state estimation. Still, perceived information about environmental structures is constrained to lie on the 2D measurement planes of the moved scanner. In contrast, we use a continuously rotating laser scanner that does not only allow for capturing 3D measurements without moving, but also provides omnidirectional obstacle sensing at comparably high frame rates (4 Hz in our setup).

To the knowledge of the authors, there exist no scientific works regarding MAV-based stocktaking. However, it is worth mentioning that the proposed system was developed for the German BMWi funded Autonomics for Industry 4.0 project InventAIRy [7]. Furthermore, the company DroneScan [25] recently demonstrated first results of MAV-based inventory.

3 System Overview

Our MAV design is a hexarotor with a 1.24 m diameter frame surrounding the rotor plane. The total weight is 5.0 kg. The thrust-to-weight ratio is approximately 1.5. Figure 1 shows our MAV in an indoor environment. While fragile equipment like computer and laser scanner lies in the core of the MAV, the frame protects the rotors and is used for mounting multiple sensors. For sensor data processing and navigation planning, we use an unboxed Gigabyte GB-BXi7-4770R as the onboard processing system. The small board is equipped with an Intel Core i7-4770R quadcore CPU, 16 GB DDR3-memory, and a 480 GB SSD.

For state estimation, obstacle detection, localization, and mapping, our MAV is equipped with a multimodal sensor setup. Figure 2 gives an overview of the installed sensors. The vision system of our MAV features a ring of six Ximea MQ013MG-E2 1.3 M Pixel USB 3.0 cameras, yielding an omnidirectional FoV. The cameras are used for visual odometry and for the detection of visual features like AprilTags [23].

We use two rotating Hokuyo UST-20LX laser scanners with orthogonal measurement planes to achieve a comprehensive perception of the MAV surroundings. Each laser scanner provides distance measurements of up to 20 m with 270° apex angle.



Fig. 1 Our MAV has been designed for inventory and short-range inspection tasks in indoor environments. Reliable perception of obstacles in the surrounding is key for safe operation

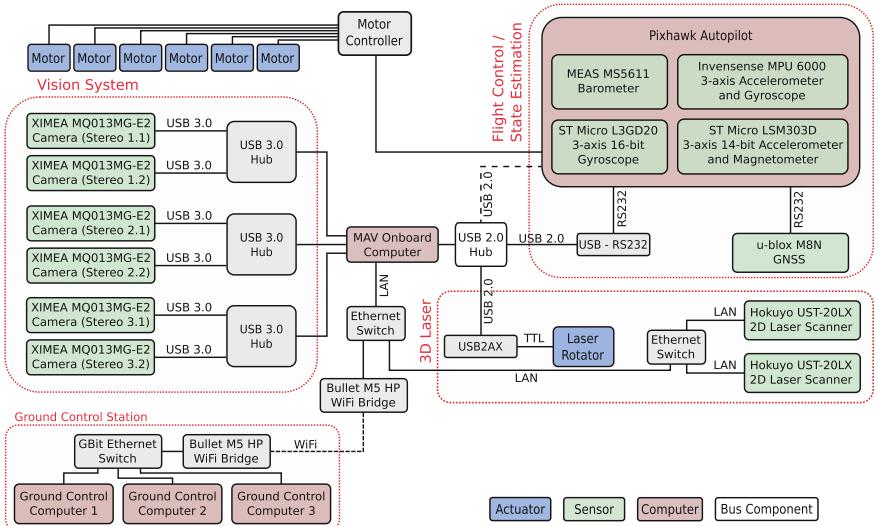


Fig. 2 Scheme of the sensors, actuators, computers, and bus systems on our MAV. We use high-bandwidth USB 3.0 connections for the cameras due to the high data rates, and lower-bandwidth buses for flight control and RFID reader. The *dashed line* indicates a wireless connection

The 3D laser is used for obstacle perception and 6D self-localization in a 3D map. An RFID reader module allows for the fast detection of passive RFID tags that identify storage places and warehouse stock.

For high-level navigation tasks, we employ ROS as middleware on the onboard computer and on ground control stations. For low-level velocity and attitude control, the MAV is equipped with a Pixhawk Autopilot flight control unit [19] that also contains gyroscopes, accelerometers, a compass, a barometer, and an optional GNSS receiver. We modified its firmware to meet our requirements. In contrast to the original implementation, we control the MAV by egocentric¹ velocity commands calculated by the onboard PC. Hence, we need a reliable egocentric velocity estimate, independent from allocentric² measurements, i.e., compass orientation. Our state estimation filter, which estimates 3D positions, 3D velocities, and 3D accelerations, integrates—in addition to the measurements already considered in the original implementation—external sources provided by the onboard PC. These include visual odometry velocities and laser-based localization.

To achieve high camera frame rates at full resolution, we connect the cameras via three hubs, one per stereo pair, to a dedicated USB 3.0 bus of the onboard computer. Onboard components with lower bandwidth requirements, i.e. the flight control unit and the laser scanner rotator, are connected to a second USB 2.0 bus. The Pixhawk Autopilot is connected twice. The first connection via an USB-to-serial converter provides the telemetry and control connection according to the MAVLink protocol [18]. The second connection is inactive during flight and is only used for debugging and firmware updates of the Pixhawk Autopilot on the ground. Figure 2 illustrates our onboard USB setup.

While our MAV frame also supports the use of 15" propellers, we use six MK3644/24 motors (111 g each) with 14" propellers to generate thrust. Turnigy 5S, 10 Ah, 35C batteries power the MAV, including all periphery. The batteries weight 1.28 kg each and are hot-swappable. Thus, it is not necessary to shut down the onboard computer while changing batteries.

Real-time debugging and control of onboard functions with our ground control stations is crucial for the efficient development of algorithms. To ensure seamless operation, we use two Ubiquity Networks Bullet BM5HP WiFi adapters. They are configured to work in Wireless Distribution System (WDS) Transparent Bridge Mode to behave as if a wired connection would be present. Our network setup is also shown in Fig. 2.

4 Perception

We use the ROS packages `tf`, `robot_state_publisher`, and `urdf` to incorporate the physical sensors, mounted on the MAV, into our software. The transformations for the robot model are first estimated from a coarse CAD model

¹The egocentric frame lies in the center of the MAV.

²The allocentric “world” frame is a globally fixed frame in the warehouse.

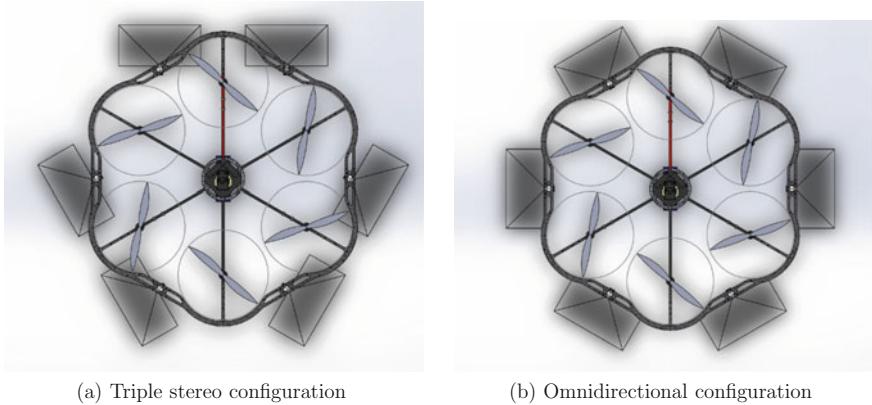


Fig. 3 Mounting of the cameras in **a** triple stereo and **b** omnidirectional configuration. Configuration **a** facilitates the usage of available standard stereo methods. In configuration **b** cameras have partial image overlap with both neighboring cameras. This allows the development of truly omnidirectional vision methods

(Figs. 3 and 6), and later calibrated by sensor-specific methods which are described in the respective subsections. In the following, we detail the sensors used on the MAV, and describe how they are incorporated into our ROS infrastructure.

4.1 Accelerometers, Gyros, Compass, and Barometer

Low-level sensors like accelerometers, gyros, compass, and barometer are part of the Pixhawk Autopilot to ensure real-time processing of these—in relation to, i.e., USB interface latency—comparatively fast sensors. For a fast transient response, state estimation—detailed in Sect. 6—runs directly on the Pixhawk Autopilot. Hence, raw data of accelerometers, gyros, compass and barometer is only fed to the main computer for logging purposes (e.g., `sensor_msgs/Imu` ROS message), but processed directly on the flight control unit. The filtered results are also transferred to the onboard PC with the MAVLink protocol and published on ROS topics by our ROS-MAVLink communication node which is based on `mavlink_ros`.³

4.2 Cameras

We use six Ximea MQ013MG-E2 global-shutter monochrome USB 3.0 cameras (22.5 g each) for visual perception. The camera configuration can be easily switched

³http://github.com/mavlink/mavlink_ros.

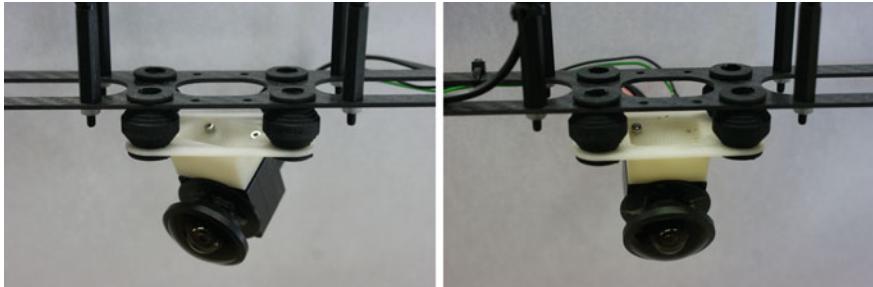


Fig. 4 To cope with unavoidable vibrations during flights, our camera mounts are equipped with vibration dampers. The *left* mount is for stereo configuration, the *right* mount for omnidirectional camera configuration

from three stereo-pairs (Fig. 3a) to an omnidirectional configuration including all six cameras (Fig. 3b). While the stereo configuration facilitates the usage of available standard stereo methods, the latter allows the deployment of truly omnidirectional vision methods that have not been addressed by many researchers yet. The mountings are detailed in Fig. 4. We use vibration dampers to isolate the cameras from high frequency oscillations caused by the imbalance of the propellers. Each mounting—including dampers—weights 11.5 g.

In combination with Lensagon BF2M2020S23 fisheye lenses with 195° apex angle (25 g each), an omnidirectional FoV can be obtained. The use of multiple camera pairs not only facilitates omnidirectional obstacle perception, but also provides redundancy. So if, e.g., the MAV points one camera-pair towards featureless surfaces or the sky, the others are still able to perceive the environment.

Communication and Control

With every pair of stereo cameras sharing a passive USB 3.0 HUB, we achieve frame rates of up to 55 fps, depending on exposure timings. All cameras are synchronized by hardware triggering. When data from all cameras has been received, the next frame is triggered. This enables us to achieve adaptive high frame rates which results in data rates of up to 200 MB/s. Figure 5 shows an image obtained during flight.

The communication with all six cameras is performed by a single driver node that processes all images. The node acts as a wrapper for the Ximea xiAPI⁴ API. Missing on-camera functionalities, e.g., gamma correction and rectification, are performed directly in this node and the enhanced fisheye and rectified images are published on ROS topics, accordingly. We use the dynamic_reconfigure ROS package to set the camera and acquisition parameters, e.g. exposure time, gain, frequency, and auto-exposure, dynamically during runtime. Additionally, the user has the choice to employ different image enhancement techniques, as listed in Table 1. The image correction and rectification is performed on the original 10-bit images during readout, using a pre-generated look-up table. By using the

⁴<https://www.ximea.com/support/wiki/apis/XiAPI>.



Fig. 5 Fisheye camera image of one of the onboard cameras. Due to the large FoV, the frame is slightly visible in the stereo configuration. We address this issue by either masking or by rectification which removes these artifacts

Table 1 Available pixel-wise operations for image I

Image transformation	Operation
Gamma	$I_{out} = c \cdot I_{in}^{\frac{1}{\gamma}}$
Logarithmic	$I_{out} = c \cdot \log(1 + I_{in})$
Contrast-stretching	$I_{out} = \frac{1}{1 + \left(\frac{m}{I_{in}}\right)^E}$
10-to-8-bit conversion	$I_{out} = I_{in} \gg 2$
None	$I_{out} = I_{in}$

The parameters, including gamma γ , scaling constant c , mid-line m , and slope control E , can be changed during runtime using, e.g., `rqt_reconfigure`

`dynamic_reconfigure` server, the look-up table can be recalculated if the correction parameters are changed by the user. The processed images are published as `sensor_msgs/Image` messages, together with downsampled rectified images and corresponding `sensor_msgs/CameraInfo` info messages. For efficiency reasons, the driver can write camera Bag files directly to the disk, bypassing the ROS communication infrastructure.

Calibration and Rectification

Each stereo camera pair is calibrated intrinsically and extrinsically, using the epipolar equidistant model [1] especially developed for fisheye camera calibration. It uses a 3D calibration target with point markers on three orthogonal planes (see Fig. 6b), which is observed from different distances and angles during calibration. The approximate positions of the 3D point markers have to be known. In an offline calibration run, the intrinsic and extrinsic calibration parameters are estimated together with the 3D coordinates of the calibration target by bundle adjustment, formulated as least squares problem. The user can select different applicable projection and distortion models in the calibration toolbox. For modeling the projection of our fisheye cameras, we employ the epipolar equidistant model, that describes the projection of a spherical image onto a plane as shown in Fig. 6a. The lens distortion is described using a third-order Chebyshev polynomial. For image rectification with horizontal epipolar lines, we pregenerate look-up tables to allow for fast online processing during flight. Furthermore, to improve computational efficiency, the images are downsampled to half the resolution during rectification. The overall time needed for the rectification of one image is approximately 1 ms. Timings for different resolutions are listed in Table 2. Overall, we obtain a reprojection error of 0.75 pixels and estimate a baseline of 53.362 cm.

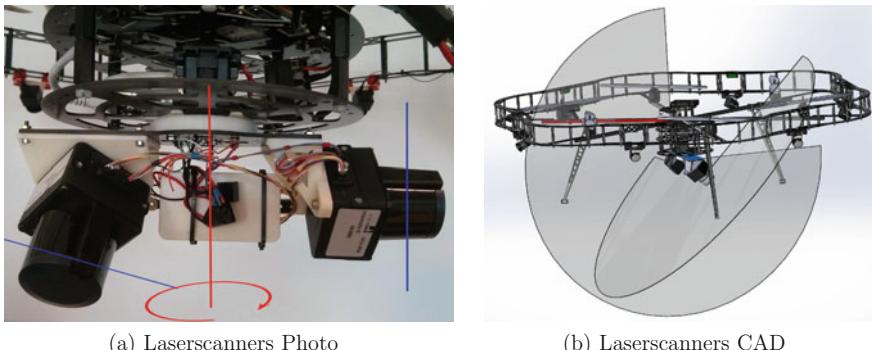


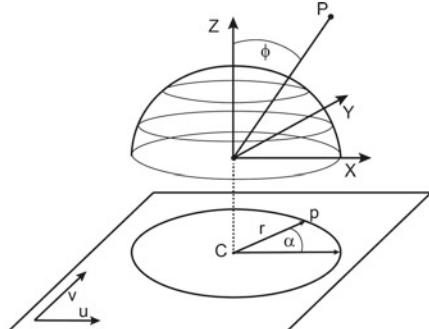
Fig. 6 Photo and CAD drawing of our 3D laser scanner with the FoV of the individual 2D laser scanners (blue). **a** Scanner 1 (right), and scanner 2 (left) have different FoVs. The Hokuyo 2D laser scanners are mounted on a bearing and rotated around the red axis. **b** Scanner 1 is rotated to the back of the image plane to show the 270° opening angle of the scanner. Scanner 2 is in the front, showing the twisted scan plane

Table 2 Rectification runtime

Target image resolution	Time per image (ms)
1280 × 1024	4
640 × 512	1
320 × 256	0.7

4.3 Laser Scanner

Our custom-built 3D laser perceives the environment around the MAV at a frequency of 2Hz. The sensor combines two Hokuyo UST-20LX laser range finders mounted on a link. A Robotis Dynamixel MX-28 servo actuator rotates the link around the vertical axis with one revolution per second, yielding an spherical FoV. The servo actuator measures the angular position of the laser range finders with 0.088° resolution. Figure 6 depicts the scanner arrangement, showing that one scanning plane is parallel to the axis of rotation while the other is twisted by 45° to obtain denser measurements in a $\pm 45^\circ$ vertical \times 360° horizontal FoV. This arrangement results in a small upward pointing cylindrical blind spot of the first scanner and conical, upward and downward pointing blinds spot for the twisted scanner. Hence, this setup maximizes the FoV and obtains many measurements in flight height. Since the blind spot is closed by copter attitude changes, it does not degrade mapping or obstacle detection in our scenario. Furthermore, due to the large FoV of 270° within the scan planes, a half rotation of the link produces a 3D scan in almost all directions.



(a) Fisheye camera model



(b) 3D point target



(c) Raw image



(d) Rectified image

Each 2D laser range finder has a scanning frequency of 40 Hz with 1,080 measurements per scan plane resulting in 43,200 measurements per second. Figure 7 shows resulting point clouds of the environment perceived by each laser and the combined point cloud. Each scanner weights 143 g (without cables). The whole sensor assembly weights 420 g including motor, a network switch, and a slip ring allowing for

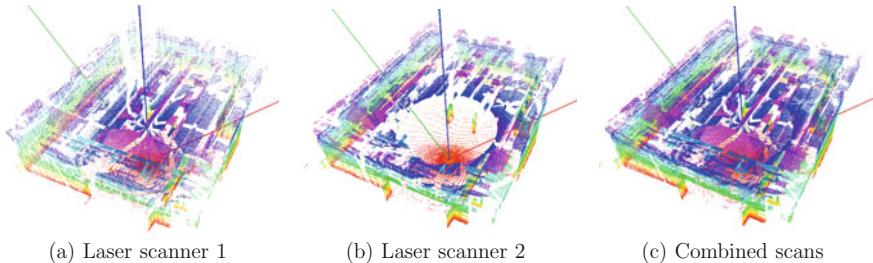


Fig. 7 Point clouds from the rotating 3D laser scanner. While the individual scanners show substantial blind spots, nearly no occlusions occur in the combined scan. **b** Especially laser scanner 2 shows a large blind spot above the MAV caused by the limited opening angle and the twisted mounting position. See also Fig. 6. The axes represent the pose of the MAV. Color encodes height

continuous rotation. For communication with the two individual laser scanners, we employ the driver provided by the ROS `urg_node` package.

The wide FoV of the laser scanner inherently leads to many measurements on the MAV itself. Considering the complex structure of the MAV, with moving parts like propellers, we remove measurements that belong to the robot's body. This so-called *self filter* approximates the model of the MAV by a cylinder with the diameter and height of the MAV. Furthermore, we use a modified shadow filter—based on the ROS `laser_filters` package—to remove not only incorrect measurements at the edges of the geometry, but also erroneous measurements caused by the fast rotating propellers. Filtering results are shown in Fig. 8.

The absolute position of the laser relative to `base_link` is calibrated manually. The length of the link, the 2D laser scanners are mounted on, is crucial for scan consistency. Since it is only approximately known, we iteratively tune this parameter.

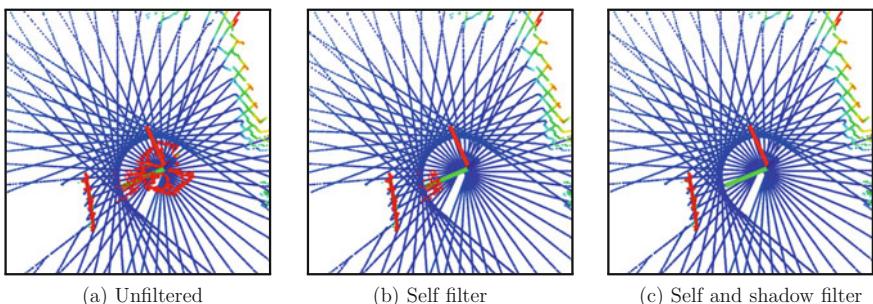


Fig. 8 Demonstration of the employed scan filters. A 3D scan assembled from one half rotation of the 3D laser scanner is shown from a top-view. Color encodes height. The MAV (depicted by the axes) passes the obstacle on the *left*. The *red points* close to the MAV are spurious measurements caused by the MAV itself and the occluded transition between the obstacle and the MAV. **a** Unfiltered 3D scan. **b** Filtered 3D scan using the self filter only. Spurious measurements remain. **c** Filtered 3D scan using self filter and modified shadowing filter. Spurious measurements are removed

By visualizing all single scanlines of a whole 3D scan in RViz, the parameter can be adjusted, until walls and ceilings have low variance.

We construct an MAV-centric multiresolution grid map that is used to accumulate sensor measurements [4]. We first register newly acquired 3D scans with the so far accumulated map and then update the map with the registered 3D scan. The map is utilized by our path planning and obstacle avoidance algorithms described in subsequent sections.

3D Scan Assembly

When assembling 3D scans from raw laser scans, we account for the rotation of the scanner w.r.t. the MAV and for the motion of the MAV during acquisition. Thus, scan assembling mainly consists of two steps.

First, measurements of individual scan lines are undistorted with regards to the rotation of the 2D laser scanner around the servo rotation axis (red axis in Fig. 6). Here, the rotation between the acquisition of two scan lines is distributed over the measurements by using spherical linear interpolation provided by `laser_geometry/LaserProjection`.

Second, we compensate for the motion of the MAV during acquisition of a full 3D scan. To this end, we incorporate a motion estimate from the low-level filters running on the Pixhawk incorporating inertial measurement unit (IMU) and visual odometry measurements. The 6D motion estimate is used to assemble the individual 2D scan lines of each half rotation to a 3D scan.

Local Multiresolution Map

The assembled 3D scans are aggregated in a local multiresolution grid map [4]. Local multiresolution maps have a high resolution close to the robot and a lower resolution farther away. Each grid cell represents both occupancy information and the most recent individual distance measurements. The measurements of each cell are summarized in a surface element (surfel) by the sample mean covariance (cf. Fig. 9). Compared to uniform grid-based maps, multiresolution leads to the use of fewer grid cells—without losing relevant information—and consequently results in lower computational costs. Figure 9 shows an example of our local multiresolution grid-based map.

Registration Approach

We register each newly acquired 3D scan with the local multiresolution map of the environment with our surfel-based registration method [4]. Instead of considering each point individually, we represent the 3D scan as local multiresolution grid and match surfels. A newly acquired scan (scene) is aligned to the local multiresolution map (model) by finding a rigid 6 degree-of-freedom (DoF) transformation $T(\theta)$ that best aligns the scene surfels to the model surfels.

Compared to dense RGB-D images [31] or high-resolution static 3D laser scans used in our previous work [27], 3D scans obtained from our laser scanner are much sparser. We cope with this sparsity through probabilistic assignments of surfels during the registration process. Observations are described by a mixture model, avoiding

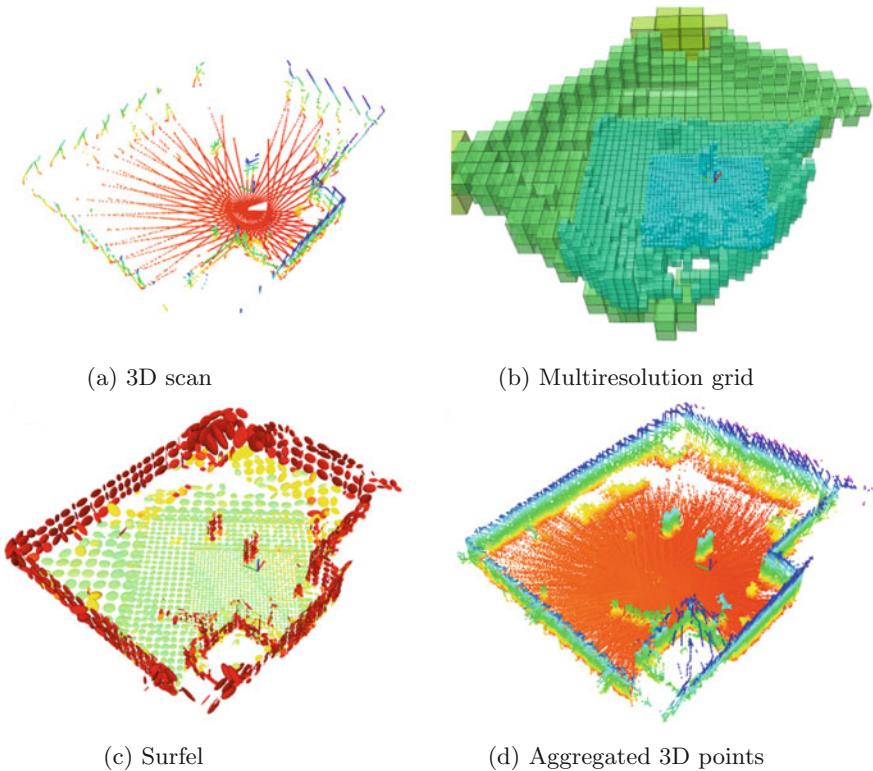


Fig. 9 Local multiresolution grid map. **a** The 3D scan acquired with our continuously rotating laser scanner, ceiling removed for better visibility. **b** The multiresolution grid structure of the map. Cell size (indicated by color) increases with the distance from the robot. **c** For every grid cell a surfel summarizes the 3D points in the cell. Color encodes the orientation of the surfel. **d** 3D points stored in the local multiresolution map. Color encodes height from ground

hard associations between surfels. The transformation $T(\theta)$ is recovered by *expectation maximization* (EM), where the E-Step finds new surfel assignments based on the last estimation of θ and the M-step optimizes θ based on the last assignments. This optimization is efficiently performed using the Levenberg–Marquardt (LM) method as in [31]. By summarizing measurements in surfels, and therefore considering significantly less elements for registration, we gain efficiency. When matching surfels, we choose the finest common resolution available between both maps to achieve accuracy.



Fig. 10 RFID sensor. The MAV is equipped with a lightweight RFID antenna (*left*) and a small RFID reader module (*right*), connected to the onboard PC via USB. The RFID system is used to map positions of RFID tags in the allocentric map attached to shelves or inventory

4.4 Radio-Frequency Identification

We inventory stock either by visually perceiving and mapping attached April Tags (Sect. 6.3) or by locating attached RFID Tags. To read RFID tags placed on shelves or inventory, our MAV is equipped with a ThingMagic M6e RFID module and an unboxed SkyeTek SP-AN-04-UF-BB6LP directional antenna (Fig. 10). The module can detect RFID tags at distances up to several meters, depending on transmit power. A ROS node, based on the ThingMagic Mercury API,⁵ decodes the RFID readings and converts them to ROS messages containing a header, the detected ID as string, and a signal strength indicator. Together with the MAV pose, these messages could be sent to a warehouse management system (WMS). To this end, we project received RFID detections into the allocentric warehouse map by means of a simple sensor model for visualization purposes.

5 Mapping

For fast estimation of the MAV motion, we incorporate IMU and visual odometry measurements into velocity and pose estimates. While these estimates allow us to control the MAV and to track its pose over a short period of time, they are prone to drift and thus are not suitable for localization on the time scale of a mission. Furthermore, they do not provide a fixed allocentric frame for the definition of mission-relevant poses independent from the MAV. Thus, we build an allocentric map by means of laser-based simultaneous localization and mapping (SLAM) before mission execution and employ laser-based pose tracking w.r.t. this map during autonomous operation.

This allocentric map is built by aligning multiple local multiresolution maps, acquired from different view poses [5]. We model the different view poses as nodes

⁵<http://www.thingmagic.com/index.php/mercuryapi>.

in a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ that are connected by edges. A node consists of the local multiresolution map from the corresponding view pose. Each edge in the graph models a spatial constraint between two nodes.

After adding a new 3D scan to the local multiresolution map as described in Sect. 4.3, the local map is registered towards the previous node in the graph using the multiresolution surfel registration with probabilistic assignments [4]. A new node is generated for the current local map, if the MAV moved sufficiently far. The registration result x_i^j between a new node v_i and the previous node v_j is a spatial constraint that we maintain as values of edges $e_{ij} \in \mathcal{E}$. In addition to edges between the previous node and the current node, we add spatial constraints between close-by view poses that are not in temporal sequence.

On each scan update, we check for one new constraint between the current reference v_{ref} and other nodes v_{cmp} . We determine a probability

$$p_{\text{chk}}(v_{\text{cmp}}) = \mathcal{N}(d(x_{\text{ref}}, x_{\text{cmp}}); 0, \sigma_d^2)$$

that depends on the linear distance $d(x_{\text{ref}}, x_{\text{cmp}})$ between the view poses x_{ref} and x_{cmp} . According to $p_{\text{chk}}(v)$, we choose a node v from the graph and determine a spatial constraint between the nodes using our surfel registration method.

From the graph of spatial constraints, we infer the probability of the trajectory estimate given all relative pose observations

$$p(\mathcal{V} | \mathcal{E}) \propto \prod_{e_{ij} \in \mathcal{E}} p(x_i^j | x_i, x_j).$$

Each spatial constraint is a normally distributed estimate with mean and covariance determined by our probabilistic registration method. This pose graph optimization is efficiently solved using the `libg2o` ROS package by Kuemmerle et al. [16], yielding maximum likelihood estimates of the view poses x_i .

After the MAV has traversed the environment, the allocentric map is built from the optimized pose graph by merging all local surfel maps. Here, we use surfels with



Fig. 11 SLAM point cloud. *Left* Resulting point cloud after pose graph optimization acquired by a manual flight along a warehouse aisle (color depicts height). *Right* Photo of the mapped aisle

uniform resolution. Figure 11 shows an example map acquired from a flight through a warehouse aisle. Our mapping pipeline is available as open-source ROS-based package.⁶

6 Localization and State Estimation

In order to navigate in indoor and outdoor environments, robust localization and state estimation, especially in GNSS-denied environments, is crucial. Our multimodal localization and state estimation pipeline exploits the specific characteristics of all sensors in terms of, e.g., accuracy and speed.

6.1 Triple Stereo Visual Odometry

Our visual odometry estimation is based on the ROS `viso2` package that wraps the visual odometry library LIBVISO2 [10], a fast feature-based method for monocular and stereo cameras. The approach does not require a motion model. The only prerequisites are that the input images are rectified and the extrinsic camera calibration is known.

We rectify the fisheye images with the method *epipolar image rectification* on a plane with an equidistant model as proposed by Abraham and Förstner [1]. The resolution of the rectified images is 640×512 . The rectified image pairs are fed into three instances of `viso2` running in parallel—one for each stereo camera pair—to obtain three velocity estimates.

Similar to other feature-based methods, `viso2` extracts and matches features over subsequent stereo frames and estimates the camera motion by minimizing the reprojection error. Four types of features (corners and blobs of two polarities) are detected using 5×5 filters and non-maximum suppression. Feature similarity is computed by sparse horizontal and vertical Sobel filters. As shown in Fig. 12, feature associations are searched in small prediction windows between frames and along epipolar lines between the stereo pairs and matches are only accepted if a circular match across two adjacent frames and the two cameras can be established. Based on all found circle matches, Geiger et al. [10] estimate the camera motion by minimizing the reprojection error using Gauss-Newton optimization in combination with RANSAC for outlier removal.

The estimated 3D velocities from the three stereo pairs are utilized in the state estimation pipeline. We weight each velocity estimate according to the number of correspondences that are tracked. When the number of features falls below a threshold, e.g. due to featureless or overexposed scenes, the weight is set to zero. In this

⁶https://github.com/AIS-Bonn/mrs_laser_map.

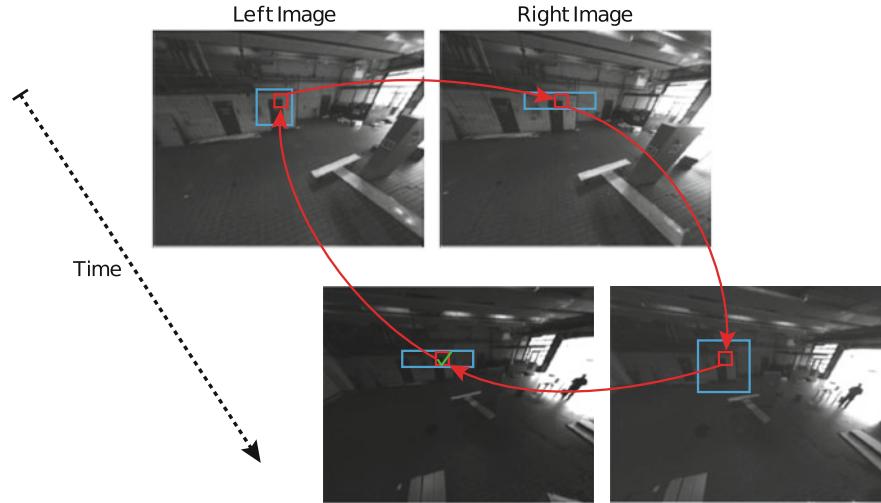


Fig. 12 Circular matching of feature points by viso2[10]: starting from a feature detected in the current *left* image (*lower left*), a windowed correspondence search (*blue box*) is performed on the previous *left* image (*upper left*). If a match has been found, it is matched along the epipolar line to the previous *right* image and from there to the current *right* image. The best match for this feature is searched along the epipolar line in the current *left* image. The match is accepted only if the loop is closed

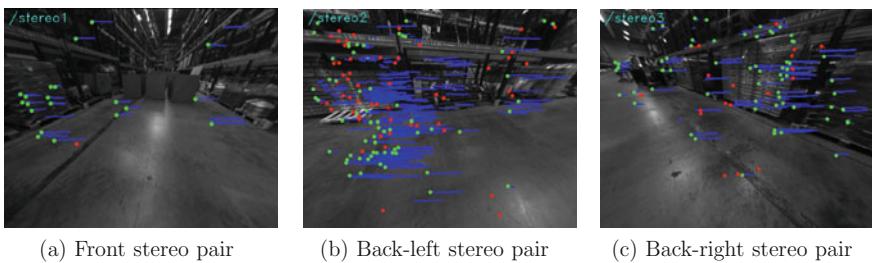


Fig. 13 Triple stereo visual odometry. While the forward facing camera tracks few features due to fast forward motion, the remaining stereo pairs can still estimate reliable feature correspondences. Correspondences within one stereo pair are colored *blue*. Feature correspondences tracked by viso2. RANSAC is used for outlier detection. Inliers are colored *green*, outliers are colored *red*

way, we obtain visual odometry even if two cameras fail at the same time. Moreover, especially at fast forward motions where the feature correspondence search with the frontal camera is challenging, the estimates of the lateral cameras allow for proper motion estimation, as shown in Fig. 13.

The independent odometry estimates are published in the `base_link` coordinate frame. As the transformation from the camera coordinate systems to the `base_link` is static, it is looked up once at the beginning by using the `TransformListener` of the ROS `tf` package.

6.2 Laser-Based Pose Tracking

In order to localize the robot in GNSS-denied environments, e.g., indoor environments, in an allocentric frame, we register local multiresolution maps to a global map employing multiresolution surfel registration (MRSR) [4]. In small environments, suitable maps can be built from the takeoff position before a mission. In larger environments, we perform laser-based SLAM (cf. Sect. 5).

Since the laser scanner acquires 3D scans with a relatively low rate of 2 Hz, we incorporate the egomotion estimate from the visual odometry and measurements from the IMU to track the pose of the MAV. The egomotion estimate is used as a prior for the motion between two consecutive 3D scans. In detail, we track the pose hypothesis by alternating the prediction of the MAV movement given the filter result and alignment of the current local multiresolution map towards the allocentric map of the environment.

To align the current local map with the allocentric map, we also use the surfel-based registration described in Sect. 4.3. The allocentric localization is triggered after a new 3D scan has been registered with and added to the local multiresolution map. We update the allocentric robot pose with the resulting registration transform. To achieve real-time performance of the localization module, we only track one pose hypothesis. We assume that the initial pose of the MAV is known, either by starting from a predefined pose, or by means of manually setting the pose. Figure 14 shows the registration of a 3D scan to the map and an estimated 6D trajectory.

The resulting robot pose estimate is used as a measurement update in a lower-level state estimation filter. We propagate this allocentric pose over time with visual

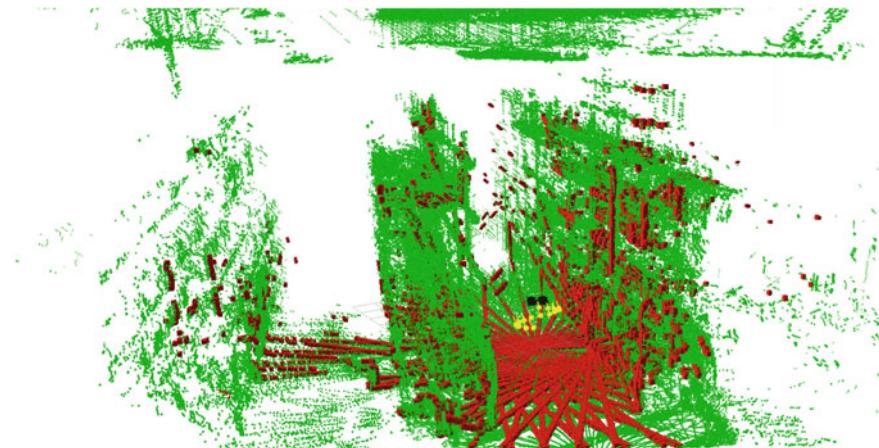


Fig. 14 Laser-based localization. A laser scan aggregated over 500 ms (red) is matched to an allocentric map (green) to track the MAV pose (black). The yellow dots depict the tracked MAV trajectory

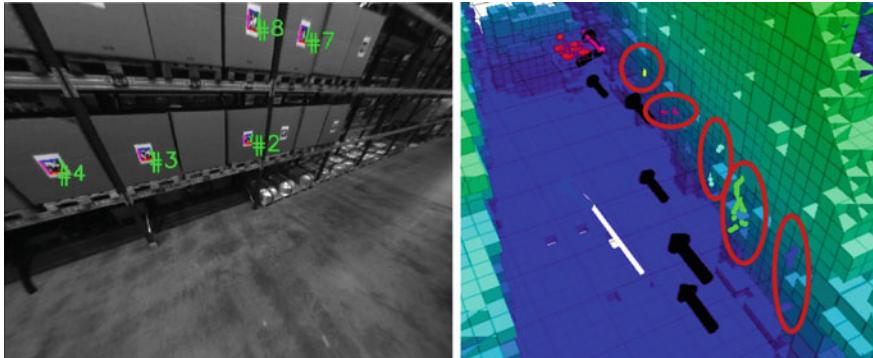


Fig. 15 AprilTag detections. *Left* Detected AprilTags in the rectified camera image with corresponding ID. *Right* Poses of the tag detections projected into the allocentric map with the MAV pose estimate before filtering. The colors correspond to the tag IDs. *Black arrows* depict mission view poses

odometry and IMU to obtain allocentrically consistent pose and velocity estimates at a sufficiently high rate for planning and control.

6.3 AprilTag Detection

In order to improve the indoor localization of our MAV in environments with repetitive structures, e.g., warehouses, and to localize tagged objects, we augment the environment with AprilTags. These tags can be robustly detected in real time with the wide-angle cameras. Figure 15 shows the detection of AprilTags with 164 mm edge length. The algorithm is able to detect and locate tags in distances of 0.5 to 5.0 m. The computation time is 10 ms per image. We build maps of AprilTags in an allocentric frame by mapping with known poses based on laser-based localization. Figure 15 also shows the resulting map after an example flight based on the observations from all six cameras.

6.4 State Estimation Filter

We use two filters for state estimation: A low-level extended Kalman filter (EKF) fuses measurements from accelerometers, gyros, and compass to one 6D attitude and acceleration estimate. The second, higher-level, filter fuses linear acceleration, velocity, and position information to a state estimate that includes 3D position. The low-level filter is supplied with the Pixhawk Autopilot. The higher-level filter extends

the original Pixhawk Autopilot position estimator by incorporating all the sensors present on the MAV into one state.

Here, we predict the state:

$$\mathbf{x} = \begin{pmatrix} p_x & p_y & p_z \\ v_x & v_y & v_z \\ a_x & a_y & a_z \end{pmatrix},$$

consisting of 3D position p , 3D velocity v , and 3D acceleration a under the assumption of uniform acceleration

$$\begin{aligned} p_k &= p_{k-1} + v_{k-1} \cdot dt + \frac{1}{2} a_k \cdot dt^2, \\ v_k &= v_{k-1} + a_k \cdot dt, \\ a_k &= a_{k-1}. \end{aligned}$$

If sensor measurements are available, the state is corrected accordingly. For 1D velocity estimates $v_{k,sens}$, coming from, e.g., visual odometry, the state correction is

$$\begin{aligned} v_k &= v_{k-1} + (v_{k,sens} - v_{k-1}) \cdot w \cdot dt, \\ a_k &= a_{k-1} + (v_{k,sens} - v_{k-1}) \cdot w^2 \cdot dt^2. \end{aligned}$$

Here, w is a weighting factor that indicates the reliability of the inputs. Table 3 shows the measurements that contribute to the filter result. Egocentric measurements are first transformed into the allocentric frame by the attitude estimate. We determined the weighting factors by iterative tuning.

This predictor/corrector design offers the following advantages. It

- delivers fast transient responses,
- works in GNSS-denied environments, and
- does not accumulate drift.

Table 3 Information sources for the state filter

Information			Update rate (Hz)	Frame	Weighting factor
Source	Type	Dim.			
Attitude EKF	Lin. Acceleration	3D	250	egocentric	20
Visual odometry	Velocity	3D	15	egocentric	0–2
GNSS	Velocity	3D	10	allocentric	2
Barometer	Position	1D	250	allocentric	0.5
Laser pose tracking	Position	3D	2	allocentric	2
GNSS	Position	3D	10	allocentric	1

As can be seen in Fig. 2, we use a USB-to-serial converter to communicate with the Pixhawk Autopilot. We use the maximum rate of 921,600 baud to achieve a measurement frequency of up to 250 Hz for attitude, velocity, and position updates.

7 Navigation

To facilitate efficient and safe operations without or with only small human interaction, we employ the multilayered navigation approach illustrated in Fig. 16. Each layer operates in a frequency suitable for the specific task and on a correspondingly updated and accurate environment representation. From top to bottom these layers are: Mission planning, allocentric path planning, egocentric path planning, reactive collision avoidance, and low-level control. The planning frequency increases from top to bottom, whereas the level of abstraction decreases.

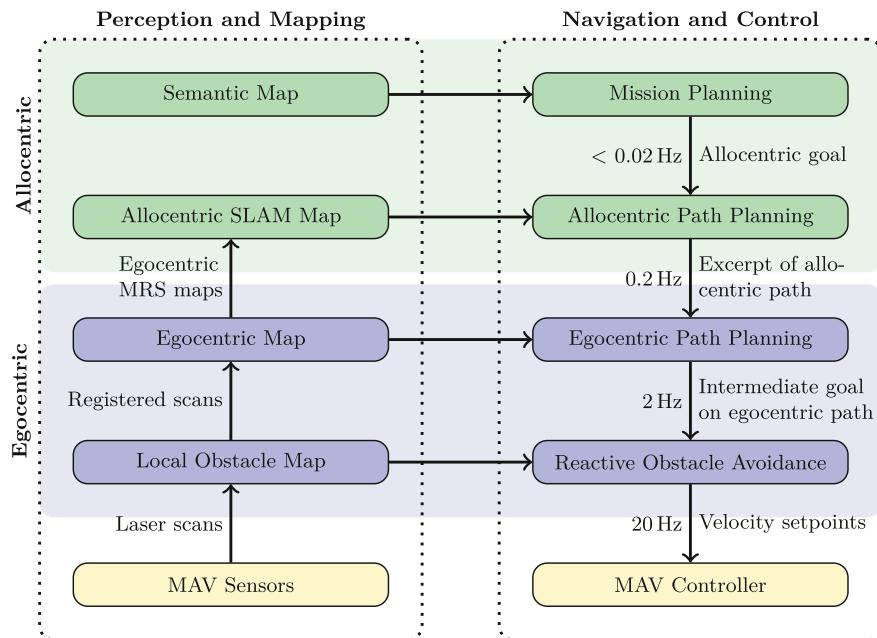


Fig. 16 Our navigation pipeline consist of five hierarchy levels. From *top to bottom*, the planning frequency increases, whereas the level of abstraction decreases

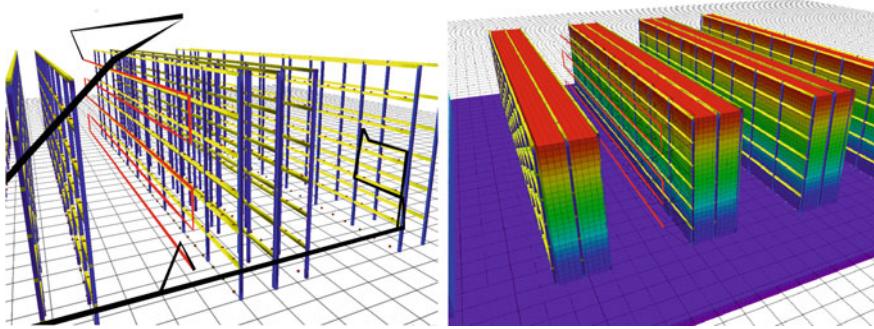


Fig. 17 Mission planning in semantic map. Based on warehouse parameters like shelf and storage unit dimensions, aisle width, etc., a semantic map of the shelves is generated. Dark red dots depict storage units. *Left* An operator can command coverage tours to scan complete shelves (red path in left aisle), flights to specific storage locations (black path to right aisle), or a combination as part of a more complex flight plan. *Right* To aid initial mission and path planning, we derive an OctoMap from the semantic map (color encodes height)

7.1 Mission Planning

The layout of large warehouses follows often a very structured pattern. Large shop floors are filled with shelves, containing standardized storage units, e.g., capable to store exactly one EUR-pallet of size $80 \times 120\text{cm}$. Thus, on the topmost layer, we describe equal parts of a warehouse by number of shelves, unit height, and the numbers of units in horizontal and vertical direction. If the storage unit IDs are assigned in a systematic way, we can derive a mapping between storage unit coordinates, scan positions, and IDs automatically. Figure 17 depicts such a model. We derive an initial OctoMap from the model for navigation planning. For development and debugging, flight plans containing flights to individual storage units and coverage paths for whole shelves can be assembled using an RViz-based interface. In real-world applications, IDs of shelves or units to inspect will be provided by a warehouse management system (WMS).

Coverage paths to inventory shelves are generated from a user-defined distance to the shelf and the sensor apex angles. A 10% overlap between scans allows detecting visual tags that could be cropped otherwise and mitigates the effects of small deviations from the flight altitude.

For missions involving flights to multiple individual storage units, we formulate the mission as traveling salesman problem (TSP). After calculating all pair-wise edge weights, the cost-optimal sequence of view poses is determined by means of Concorde [2], a fast TSP solver.

In order to define missions independent from a strictly structured warehouse model, an operator can define arbitrary 4D view poses in RViz using `interactive_markers`. A context menu at every marker allows to set a marker

to the current MAV pose—this is especially useful to teach-in missions during manual flight—and to modify, add, or remove view poses.

7.2 Path Planning

The next layer when descending the planning hierarchy is a global path planner. This layer plans globally consistent plans, based on I) the SLAM-based environment model (as OctoMap), discretized to grid cells with 0.5 m edge length, II) the current pose estimate of the MAV as `nav_msgs/Odometry`, and III) the next mission waypoint, including 3D position and yaw represented as `geometry_msgs/PoseStamped`. Planning frequency is 0.2 Hz and we use the A* algorithm to find cost-optimal paths.

In our application domain, most obstacles not represented in the allocentric map can be avoided locally, without the need for global replanning. Hence, it is sufficient to replan globally every five seconds to keep the local deviations of the planner synchronized to the global plan and to prevent the MAV from getting stuck in a local minimum that the local planner cannot escape due to its restricted view of the environment.

As via-points that are not mission critical can be blocked by locally perceived obstacles, it is not sufficient to send the next waypoint of the global path to the local planning layers. Instead, the input to the local planner is the complete global plan, which allows for skipping blocked via-points. The global path is cost-optimal with respect to the allocentric map. Hence, the path costs of the global path are a lower bound to path costs for plans refined based on newly acquired sensor information—mostly dynamic and static previously unknown obstacles. Locally shorter plans on lower layers with a local view on the map are not taken as they may yield globally suboptimal paths. Also, mission goals are not skipped as the local planner has to reach these exactly. If this is not possible, the mission planning has to resolve this failure condition.

7.3 Local Multiresolution Path Planning

On the local path planning layer, we employ a 3D local multiresolution path planner. This layer plans based on the allocentric path from the global path planner and local distance measurements which have been aggregated in a 3D local multiresolution map. It refines the global path according to the actual situation. The resulting more detailed trajectory is fed to the potential field-based reactive obstacle avoidance layer on the next level.

To resemble the relative accuracy of onboard sensors—i.e., they measure the vicinity of the robot more accurate and with higher density than distant space—we

plan with a higher resolution close to the robot and with coarser resolutions with increasing distance.

Local multiresolution for path planning is also motivated by map dynamics. Since the parts of the plan that are farther away from the MAV are more likely to change, e.g., due to newly acquired sensor measurements, it is reasonable to spend more effort on a more detailed plan in the close vicinity of the robot. Compared to uniform resolution, our approach reduces planning time and makes frequent replanning feasible.

Our planner operates on grid-based robot-centric obstacle maps with higher resolution in the center and decreasing resolution in the distance. We embed an undirected graph into this grid and perform A* search from the center of the MAV-centered grid to the goal. The edge costs are given by the base obstacle costs of the cells it is connecting and its length given by the Euclidean distance between the cell centers.

An obstacle is modeled as a core with maximum costs, determined by obstacle radius r_F that is enlarged by the approximate robot radius r_R , and a distance-dependent part r_D that models the uncertainty of farther-away perceptions and motions with high costs. Added is a part with linearly decreasing costs with increasing distance to the obstacle r_S that the MAV shall avoid if possible. The integral of the obstacle stays constant by reducing its maximum costs h_{max} with increasing radius. For a distance d between a grid cell center and the obstacle center, the obstacle costs h_c are given by

$$h_c(d) = \begin{cases} h_{max} & \text{if } d \leq (r_F + r_D) \\ h_{max} \frac{1-d-(r_F+r_D)}{2*(r_F+r_D)} & \text{if } (r_F + r_D) < d < 3 * (r_F + r_D) \\ 0 & \text{otherwise} \end{cases} .$$

The local planner is coupled to the solution of the allocentric path planner by a cost term h_a , which is the shortest distance between a grid cell and any segment of the allocentric plan (see Fig. 18). The total cost h for traversing a grid cell is $h = w_1 \cdot h_c(d) + w_2 \cdot h_a$.

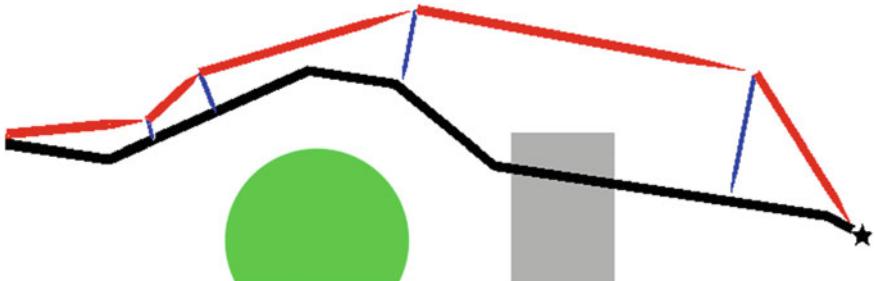


Fig. 18 The local plan (red) is coupled with the allocentric plan (black) by a cost term that penalizes deviations from the allocentric plan. The blue lines depict the deviation vectors at example points, the star is the planner's goal. The green circular obstacle is in the allocentric map, the gray rectangular obstacle has to be surrounded based on the local map

The output of the local navigation layer is the next waypoint along the planned path as `geometry_msgs/PoseStamped` in a robot-centric frame. This is further processed by a PID-controller to generate egocentric 4D velocity commands (v_x, v_y, v_z, v_{yaw}) published as `geometry_msgs/TwistStamped`. These commands are the input to the reactive obstacle avoidance layer.

7.4 Reactive Local Obstacle Avoidance

For safe navigation in complex environments, fast reliable obstacle avoidance is key. We developed a frequently updated local multiresolution obstacle map and a local reactive potential field-based collision avoidance layer to cope with dynamic and static obstacles. We transferred our previous work on obstacle perception and collision avoidance from our outdoor mapping MAV [6] to the system presented in this work.

To quickly react on obstacle perceptions, we use a version of the local multiresolution obstacle map (cf. Sect. 4.3) that is updated at the laser scanner frequency of 10 Hz. Obstacles represented in the map induce artificial repulsive forces to parts of the MAV, pushing it into free space. Figure 19 shows an example, where the MAV avoids an approaching person and the ground. To take the MAV shape into account, we discretize it into 32 cells and apply the force to each cell. The resulting force vector and the velocity control vector from a higher navigation layer yield a velocity command that avoids obstacles, independent of localization. The obstacle avoidance layer runs at 20 Hz, equal to the frequency target velocities are sent to the low-level control layer. Velocity setpoints are published as `geometry_msgs/PoseStamped`

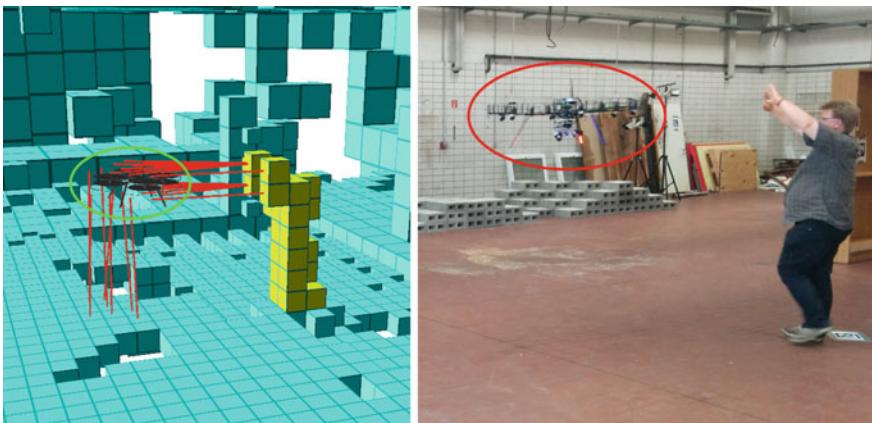


Fig. 19 The MAV is pushed away from an approaching person and the ground by potential field-based obstacle avoidance. Red lines in the left figure depict forces induced by the local obstacle map (cyan and yellow boxes, the yellow boxes depict the person) on the MAV

and received from our ROS-MAVLink bridge node. The commands are sent to the Pixhawk Autopilot via the MAVLink protocol over a serial bus.

7.5 Velocity Control

Low-level velocity control is executed on the Pixhawk Autopilot, which receives 4D velocity setpoints via the MAVLink protocol. For linear velocity control, we use a modified Pixhawk Autopilot position control node. The node implements a PID-controller which calculates a 3D thrust vector based on the 3D linear velocity error. This leads to a 3D attitude and total thrust setpoint which is then used by lower-level controllers.

We control the yaw Ψ of the MAV by a proportional controller $\Psi_{setp} = \Psi + K_p \cdot v_{yaw}$ with $K_p = 1$. Although the controller does not integrate the yaw rate v_{yaw} and thus shows a steady-state error when used open loop, it is well behaved in terms of steps in the resulting yaw setpoint Ψ_{setp} . Since we close the loop regarding yaw on a higher level, the described controller shows sufficient performance.

By limiting the maximum velocity setpoint received from the onboard computer to $2\frac{\text{m}}{\text{s}}$ in horizontal direction, $1\frac{\text{m}}{\text{s}}$ in vertical direction, and $0.2\frac{\text{rad}}{\text{s}}$ about yaw, even critical errors in ROS subsystems do not lead to severe effects at lower control layers on the MAV. We found these values to balance well between efficiency and safety in our application. Especially low yaw rates allow the safety pilot to intervene before the MAV rotates into an undesirable pose.

8 User Interfaces

8.1 Flight Operator Interfaces

Operating a complex robotic system in the field—especially for debugging and testing—requires a visualization of the system state easily monitored in real time and the possibility to quickly send the most important commands to the system. These include, but are not limited to, switching between “manual”, “velocity controlled”, and “fully autonomous” operation. Furthermore, the operator has the ability to command the MAV to “fly to specific point” determined by an interactive marker, and “stay at current pose”. The core visualization tool during flight is RViz, extended with several application-specific views/plugins.

Typical views, possibly shown in parallel distributed to several computers, are:

- Allocentric view, showing mission and allocentric path planning, OctoMap, and localization (similar to Fig. 20),
- Egocentric view, showing local obstacle map, local path planning, and reactive obstacle avoidance (similar to Fig. 19 left),

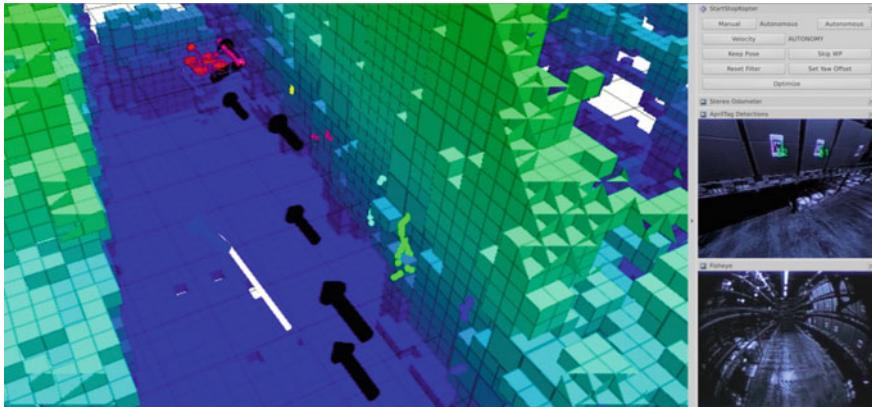


Fig. 20 Flight operator view. The RViz-based operator command and control interface depicts in the main window the allocentric obstacle map, the MAV pose (*red shape*), future mission waypoints (*black arrows*), obstacle-induced forces (not visible here since the MAV is sufficiently far away from obstacles), and 3D coordinates of detected AprilTags (*clustered colored dots*). Furthermore, approximate positions of RFID tags can be shown (not shown here). Other windows show visual AprilTag detections with corresponding ID in a rectified image (*center-right*) and the fisheye view from one front camera (*bottom-right*). An operator can choose between manual, velocity controlled, and fully autonomous operation. Quick commands—that have been identified as being especially useful during testing—include hovering at the current pose and skipping a waypoint

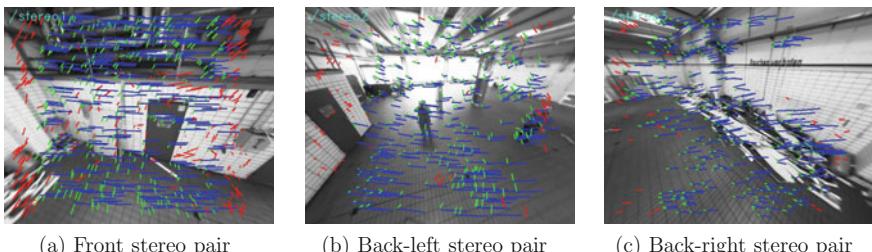


Fig. 21 Visualization of the correspondences in the three stereo camera pairs. Correspondences between one stereo pair are colored *blue*. Feature correspondences tracked by *viso2* are colored *green* (inliers) and *red* (outliers)

- Localization view, showing SLAM-map, 3D laser scans, and visual odometry trajectories (similar to Fig. 14),
- Planning view, showing allocentric and egocentric path planning, and an overlay of allocentric and egocentric maps (similar to Fig. 17), and
- Vision view, showing camera images, tracked features, tracked AprilTags and visual odometry trajectories (similar to Fig. 21).

Whereas the allocentric and egocentric views are mainly used in field-testing and actual mission execution, the localization and planning views are more

subsystem-specific and used during development and debugging. The vision view might be employed in both scenarios on demand.

Most nodes can be configured on-the-fly employing the `dynamic_reconfigure` framework. This is particularly important to parameterize lower-level systems, like the reactive obstacle avoidance and the camera system, but does also help to activate and deactivate features in high-level components.

The capabilities of ROS are not only used during a mission, but also during preparation and follow-up. As described in Sect. 5, we create an initial map by manually flying the MAV. During the manual flight, the MAV builds an allocentric map of the environment which is later used for localization and for defining a mission. An operator monitors the allocentric map using RViz to assure map coverage of the environment. We use an editing tool for post-processing the map.⁷ The point cloud can be moved and rotated. Furthermore, specific points can be deleted and the whole point cloud can be aligned to a plane, e.g., the ground plane. We use this tool to align the origin of the map with the ground level and to orient the map north—an important prerequisite to maintain a common frame between laser pose tracking, IMU, and compass measurements.

In preparation of a mission, we can define missions by either creating a job list containing storage units and shelves to cover using an RViz plugin, shown in Fig. 17, or by manually defining 4D view poses employing `interactive_markers`. Furthermore, we use an interactive marker to set the initial pose of the MAV before takeoff for pose tracking.

Repeatability of experiments is important for efficient debugging and testing. Thus, we save user-defined missions (ordered set of 4D-waypoints) and can load them for consecutive experiments. Loading and editing those stored missions have turned out to significantly reduce the operator workload when testing the system, reducing the idle time of the system and resulting in a much higher possible test frequency.

We experienced that the time needed for preparation of a mission and/or adjusting parameters and fixing bugs, often exceeds the actual time needed for the flight itself. Furthermore, consecutive short flights with short landings in between are often possible without restarting onboard systems. Thus, to minimize the time for maintenance on ground, we do not restart the logging to Bag files. After successful mission execution, we use the ROS tool `MAV_bag_filter` to cut out Bag file segments containing individual flights and discard segments where the MAV status indicates that it is not flying. In this way, we are able to (a) significantly reduce the size of the Bag files and (b) minimize the amount of time needed for reviewing the data.

⁷Point cloud editor can be downloaded from http://www.ais.uni-bonn.de/videos/ROS_book_2016.

8.2 Safety Pilot Interfaces

While the flight operators monitor higher-level states of the MAV like proper initialization of the SLAM system and correct mission planning, we rely on a safety pilot to keep the MAV in a safe state during the whole mission. The safety pilot is able to monitor all status information that is vital for safe operation of the MAV in real time. This includes battery level, velocity setpoints, flight state, and many more. Incoming MAVLink packages from the Pixhawk Autopilot are encapsulated in a ROS message and sent to the ground control station over the wireless link. Here, the messages are extracted and streamed to the local network via UDP. For real-time visualization of the data streams, we employ the software QGroundcontrol.⁸ Since this communication pipeline works bidirectionally, the safety pilot is also able to adjust flight parameters like, e.g., the maximum allowed vertical velocity during flight.

When an error occurs on a higher level or a subsystem fails, the safety pilot can always switch off the control authority of the onboard computer and recover the MAV. This can happen either with QGroundcontrol as well as with the manual remote control. We use this feature also during manual start and landing of the MAV. We manually start and land since we consider it to be safer than fully autonomous operation near the ground. By switching the control authority from manual mode to the onboard computer, we can totally eliminate the pilot in the loop. On the other hand, since we are able to completely switch off the autonomy, we can even deal with situations where the autonomy fails completely (e.g., if it should send velocity setpoints of NAN).

9 Experiments and Evaluation

We evaluated the individual components of our MAV in simulation and flight experiments in our lab. Furthermore, the integrated system was tested and demonstrated in a warehouse of a logistics company to achieve a realistic test environment. In addition to the evaluation results, we report lessons learned during development and testing of the system.

9.1 Data Acquisition

Figure 7 shows point clouds recorded with the 3D laser scanner. Due to the different angular mounting of the 2D laser scanners (cf. Fig. 6), we minimize the blind spots in the vicinity of the MAV. Occlusions, e.g., caused by the frame or propellers occur in

⁸<http://qgroundcontrol.com>.

Table 4 Camera frame rate is limited by exposure time

Exposure time (ms)	Frame rate (Hz)
40	17
23	25
17	30
3	50

different directions and can be compensated by measurements from different poses. This results in an omnidirectional FoV with a minimal blind spot.

We estimated the accuracy of the Hokuyo UST-20LX and compared it to the Hokuyo UTM-30LX-EW used in our previous work [6]. Indoors, both laser scanners show the same accuracy of $\sim \pm 10$ mm when measuring a 0.5 m distant object. Outdoors, the accuracy of the UTM-30LX-EW stays the same, but the accuracy of the Hokuyo UST-20LX degrades to $\sim \pm 35$ mm.

We evaluated the data acquisition speed of the synchronized cameras. Although the maximum frame rate is up to 55 fps, it is limited by the exposure time of the cameras. Table 4 reports the resulting frame rates.

Figure 21 shows a typical image set, captured during flight. It can be seen that the visual odometry finds most correspondences correctly, but some false correspondences are produced due to repetitive environment structures and strong illumination differences. Nevertheless, due to the redundant structure and the correspondence-dependent weighting, the visual odometry does not lose track, even if one instance finds no correspondence at all. The computation time for visual odometry including image rectification is 30 ms per stereo camera image pair.

In order to evaluate the robustness of the filter, we measured the visual odometry velocity while flying a sinusoidal trajectory. Only accelerometer, gyroscopes, and one visual odometry estimate are used to correct the filter. Figure 22 shows the visual

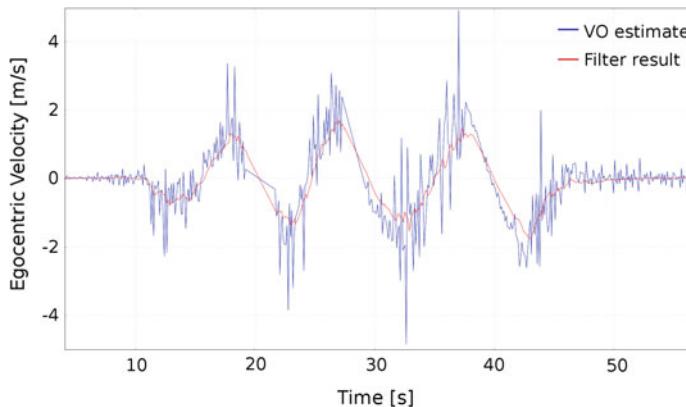


Fig. 22 Egocentric velocity estimate from visual odometry in forward direction and filter result

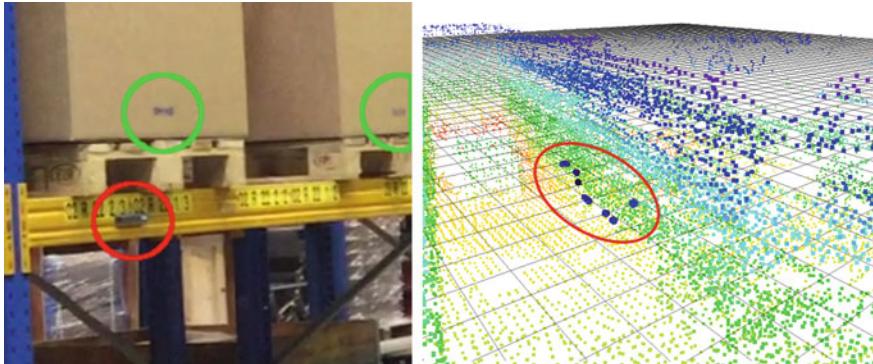


Fig. 23 RFID detections in a warehouse aisle. We map RFID tags during flight with the current MAV pose and a predefined scanning distance. *Left* Photo of the scene. Each storage unit (red circle) and every stock (green circle) is marked with RFID Tags. *Right* Representation of the scene in RViz. The detections are depicted as blue spheres

odometry input and the filter result. Although the visual odometry loses track (at $t = 19$ s and $t = 28$ s), the filter is able to bridge this information gap. In normal operation, this gap would also be filled by other velocity estimates.

9.2 RFID Detection

RFID tags are detected and mapped in the allocentric map. Instead of using an elaborated sensor model, we approximate the tag positions by a predefined offset from the RFID antenna. This is sufficient to match tag readings to storage places. For our specific case, we found an offset of 0.5 m to be appropriate. Figure 23 displays the detected tags, mounted on individual storage places during a mission in the warehouse depicted in Fig. 11. It can be seen that the achievable accuracy lies within the dimensions of one storage unit, capable of storing one EUR-pallet of size 80×120 cm. Therefore our system is capable of performing a per-storage-unit attribution of EUR-pallets.

9.3 Flight Time

We evaluated our system in flight experiments. When manually flying the MAV indoors, we measured a flight time between 6 and 8 min, depending on the flight dynamics. This is sufficient for typical indoor inspection tasks (described in detail in Sect. 9.6) and to scan one typical warehouse aisle with ~ 50 m length and ~ 5 m

height with an average horizontal velocity of $\sim 0.5 \frac{\text{m}}{\text{s}}$. Furthermore, the ability to hot-swap batteries compensates for the relatively short flight time.

9.4 Electromagnetic Compatibility

Several components on the MAV emit radio waves. We evaluated the influence of these components on each other by identifying the relevant frequencies in a series of tests. Table 5 gives an overview on the components and frequencies. Although it does not show the exact emission spectrum, it provides initial information which frequency ranges are prone to interference for further investigation.

Although our system is primarily built to work in GNSS-denied environments, our MAV is equipped with an optional GNSS antenna for use in external stock. It can be seen that the computer memory is working at the same clock frequency as the GNSS sources. We found that it emits interference radiation preventing a stable GNSS reception. Since we experienced strong interference especially with GPS, the GNSS antenna was placed as far as possible from the jamming source to reduce noise, which yielded sufficient reception of the GPS signal. We did not experience other noteworthy interferences.

Benchmarking the WiFi network gives a real throughput of 7.5 MB/s. Latency analysis gives an average ping of $1.22 \text{ ms} \pm 0.11 \text{ ms}$. We aim for a fully autonomous system, so no data has to be exchanged between the ground control stations and the MAV in normal operation modes, except for a mission specification before takeoff and data transfer to the ground station after landing. This benchmark shows that the communication infrastructure enables the operators to visualize point clouds or even view live video feeds with $\sim 2 \text{ Hz}$ for debugging purposes.

Table 5 MAV components emitting and/or receiving radio waves

Component	Frequency (GHz)
GPS L1	1.57542
GPS L2	1.2276
GLONASS L1	1.6
Computer CPU	0.8 – 3.2
Computer memory	1.6
WiFi	5.15 – 5.725
Remote control	2.4
RFID UHF	0.865 – 0.869 0.902 – 0.928

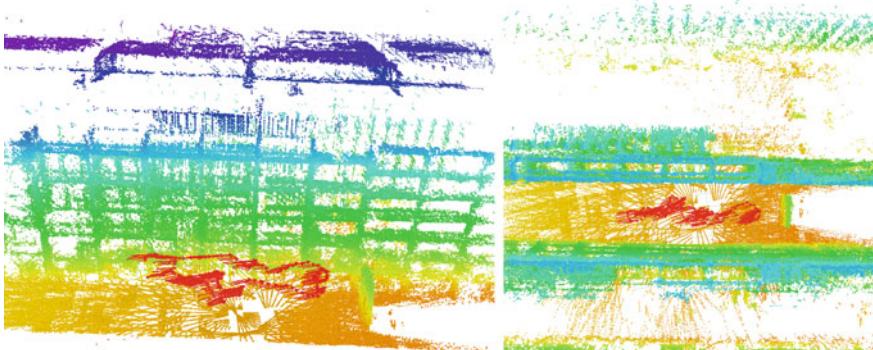


Fig. 24 Localization result. The MAV trajectory (red arrows) is tracked by means of laser scan registration, combined with visual odometry and IMU measurements. This yields 6D pose estimates. Shown is a flight through a warehouse aisle. In the side-view (*left*), the relation to the accurately mapped storage units can be seen. In the top-down view (*right*), it can be seen that the pose is tracked despite considerable self-similarity of the shelves. Map color encodes height

9.5 Mapping and Pose Tracking

We performed experiments with the integrated system. Figure 24 shows the resulting trajectory of our indoor localization experiment. We build a map with the onboard laser sensors before mission start. During a mission, the 3D laser scans—aggregated over 500 ms—are registered to the map yielding a 6D pose estimate at 2 Hz. The resulting trajectory is globally consistent.

In order to assess the performance of our global registration and allocentric mapping approach, we tested our method on a dataset of the parking garage.⁹ Without pose graph optimization, the trajectory aggregates drift which results in inconsistencies, indicated by a misalignment of the walls. Our registration method with graph optimization yields accurate results. Figure 25 shows details of a map of a garage environment. Here, even narrow structures like pipes can be identified in the globally aligned 3D scans. For a detailed comparison with other registration methods see [6].

9.6 Navigation

We evaluated the autonomous navigation by flying missions in a warehouse. The MAV visits several manually defined observation poses on different heights along a shelf based on an allocentric map created with our SLAM approach. Figure 20 shows

⁹Datasets recorded in-flight with an MAV are available at: http://www.ais.uni-bonn.de/mav_mapping.

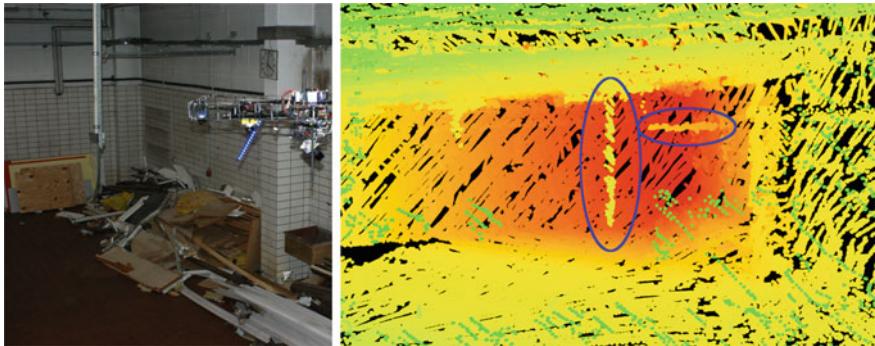


Fig. 25 Impressions of the quality of the built 3D map. Environmental structures are consistently mapped. Even details such as the narrow pipe structure and a cable canal (*circled*) are accurately modeled. Color encodes the distance to the view-point

one example mission. The MAV successfully accomplished multiple missions with a duration between ~ 2 to 5 min and a total trajectory length of ~ 40 to 80 m each.

To evaluate the local obstacle avoidance, we control the MAV with egocentric velocity commands, i.e., a zero velocity setpoint for movements in the plane and rotations, and a small descent velocity to keep the MAV close to the ground. The obstacle avoidance keeps the MAV at a safe distance to the ground. Figure 19 shows an experiment where a person approaches the MAV. The MAV avoided all static and dynamic obstacles based on the 3D laser scans.

A video showing autonomous mission execution and reactive obstacle avoidance can be found on our website.¹⁰

10 Lessons Learned

The use of ROS was extremely valuable for development and evaluation of the described MAV system. We experienced the MAV to be a very complex mechatronic system consisting of many individual hard- and software subsystems. Most subsystems offer no redundancy and show a Single Point of Failure (SPOF) characteristic.

Due to the modular and transparent ROS framework, development and error treatment was greatly simplified. Since the publish–subscribe pattern offers transparency, the effort for error analysis was reduced to a minimum. Logging of data to Bag files further simplifies error analysis.

The modular design and abstraction to ROS nodes facilitates the fast development of software and allowed us to transfer technology between multiple MAVs and even ground robots. Since the MAV is connected to several external hard- and software components, the loose coupling via ROS messages massively simplifies the

¹⁰http://www.ais.uni-bonn.de/videos/ROS_book_2016.

integration effort. Furthermore, we use standard message formats shipped with ROS and relied on third-party modules whenever possible. This facilitates both replacing submodules of the system with modules developed for other robots or even in other research groups with often low adaptation effort and the maximum use of already available ROS debugging and visualization tools.

Since ROS handles the transportation of messages, effortful data routing between the MAV and ground stations as described in Sect. 8 is not required. Nevertheless, since ROS is not real-time capable, we advice to use the `tcpNoDelay` transport hint for nodes that are crucial for real-time control. We use the no delay transport hint, e.g., regarding all communication with the Pixhawk autopilot.

Real-time visualization of data streams, especially camera images, laser scans, and planned trajectories with, e.g., RViz, and `rqt_plot`, made it possible to develop such a complex system. Real-time adjustment of crucial parameters like, e.g., camera exposure time, using `dynamic_reconfigure` sped up the development phase and also helped during evaluation.

During development of inherently unstable SPOF systems we made extensive use of simulation technology like, e.g., Gazebo, where failures are permitted.

Development of sophisticated software modules for, e.g., state estimation or action planning, was facilitated by the extensive software library which is already shipped with ROS. We rely on many standard components like drivers (e.g., `urg_node` for the laser scanners) that otherwise would be costly to develop.

Representing a complex mechatronic system in software benefited from tools like `tf` and `robot_state_publisher`. The kinematic tree represents not only statically calibrated nodes like `base_link` \Leftrightarrow `camera_1..6`, but also dynamic relations like `base_link` \Leftrightarrow `laser_scanner` or `base_link` \Leftrightarrow `map_origin`. By using the above mentioned ROS packages, we avert the cumbersome and error prone manual track keeping of a variety of multidimensional transformations. We want to advice here that although `tf` offers a transparent way to maintain transformations, to always check the `tf` tree for consistency with tools like `view_frames`.

We make extensive use of `screen` when operating the MAV. It proved to be very useful to start the `roscore` and, when working with multiple operators, all additional components in a respective screen session. Thus, if the WiFi connection drops, all components are easily recoverable and screens can be exchanged between operators.

When operating a complex robotic system in the field, it is inevitable to have well-organized processes and a tested hardware setup to not waste valuable testing time on site. In particular, clear responsibilities are important, e.g., who starts which subsystems and is responsible for their configurations—including parameter checking before takeoff and monitoring during flight. Furthermore, sufficient attention must be given to important infrastructure, like reliable networking and WiFi connections, standardized software setups on workstations, wiring of all operator station components, and if applicable, a directly available contact person on site to organize important prerequisites as power or networking and solve problems in a timely manner. The above mentioned precautions facilitate efficient usage of testing time and maximize the benefits of operations in the field.

11 Conclusions

In this chapter, we presented a cognitive MAV that is capable of semantically perceiving its environment and planning inventory missions.

We approached this challenge by employing a multimodal omnidirectional sensor setup to achieve situation awareness. The sensors have a high data rate for tracking the MAV motion and for quick detection of changes in its environment.

Our ROS-based mapping and navigation pipeline allows for fully autonomous flight even in GNSS-denied environments.

Ample onboard processing power in combination with a high bandwidth ground connection leads to a system that is suitable to deploy and debug custom algorithms and for conducting further research. The ability to hot-swap batteries and/or ground power supply makes developing and testing highly efficient.

We showed the system robustness in multiple indoor experiments where the only manual interactions were the starting and landing phases. Thus, the system is able to inspect areas in a fully autonomous mission.

References

1. Abraham, S., and W. Förstner. 2005. Fish-eye-stereo calibration and epipolar rectification. *ISPRS Journal of Photogrammetry and Remote Sensing* 59 (5): 278–288.
2. Applegate, D., R. Bixby, V. Chvatal, and W. Cook. 2006. *Concorde TSP solver*.
3. Chambers, A., S. Achar, S. Nuske, J. Rehder, B. Kitt, L. Chamberlain, J. Haines, S. Scherer, and S. Singh. 2011. Perception for a river mapping robot. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
4. Droeschel, D., J. Stückler, and S. Behnke. 2014. Local multi-resolution representation for 6D motion estimation and mapping with a continuously rotating 3D laser scanner. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
5. Droeschel, D., J. Stückler, and S. Behnke. 2014. Local multi-resolution surfel grids for MAV motion estimation and 3D mapping. In *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*.
6. Droeschel, D., M. Nieuwenhuisen, M. Beul, D. Holz, J. Stückler, and S. Behnke. 2016. Multilayered mapping and navigation for autonomous micro aerial vehicles. *Journal of Field Robotics* 33: 451–475.
7. Fiedler, M. 2016. Inventory. <http://www.inventairy.de/> [German].
8. Flores, G., S. Zhou, R. Lozano, and P. Castillo. 2014. A vision and GPS-based real-time trajectory planning for a MAV in unknown and low-sunlight environments. *Journal of Intelligent & Robotic Systems* 74 (1–2): 59–67.
9. Fossel, J., D. Hennes, D. Claes, S. Alers, and K. Tuyls. 2013. OctoSLAM: A 3D mapping approach to situational awareness of unmanned aerial vehicles. In *Proceedings of the International Conference on Unmanned Aircraft Systems (ICUAS)*.
10. Geiger, A., J. Ziegler, and C. Stiller. 2011. StereoScan: Dense 3D reconstruction in real-time. In *IEEE Intelligent Vehicles Symposium*.
11. Grzonka, S., G. Grisetti, and W. Burgard. 2012. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics* 28 (1): 90–100.
12. Hornung, A., K.M. Wurm, M. Bennewitz, C. Stachniss, and W. Burgard. 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots* 34: 189–206.

13. Huh, S., D. Shim, and J. Kim. 2013. Integrated navigation system using camera and gimbaled laser scanner for indoor and outdoor autonomous flight of UAVs. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*.
14. Jutzi, B., M. Weinmann, and J. Meidow. 2014. Weighted data fusion for UAV-borne 3D mapping with camera and line laser scanner. *International Journal of Image and Data Fusion*.
15. Kohlbrecher, S., J. Meyer, O. von Stryk, and U. Klingauf. 2011. A flexible and scalable SLAM system with full 3D motion estimation. In *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*.
16. Kuemmerle, R., G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. 2011. G2o: A general framework for graph optimization. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 3607–3613.
17. Magree, D., J.G. Mooney, and E.N. Johnson. 2014. Monocular visual mapping for obstacle avoidance on UAVs. *Journal of Intelligent & Robotic Systems* 74 (1–2): 17–26.
18. Meier, L. 2015. Micro aerial vehicle link protocol (MAVLink). mavlink.org.
19. Meier, L., P. Tanskanen, L. Heng, G. Lee, F. Fraundorfer, and M. Pollefeys. 2012. PIXHAWK: A micro aerial vehicle design for autonomous flight using onboard computer vision. *Autonomous Robots* 33 (1–2): 21–39.
20. Moore, R., K. Dantu, G. Barrows, and R. Nagpal. 2014. Autonomous MAV guidance with a lightweight omnidirectional vision sensor. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
21. Mori, T., and S. Scherer. 2013. First results in detecting and avoiding frontal obstacles from a monocular camera for micro unmanned aerial vehicles. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
22. Morris, W., I. Dryanovski, J. Xiao, and S. Member. 2010. 3D indoor mapping for micro-UAVs using hybrid range finders and multi-volume occupancy grids. In *RSS 2010 workshop on RGB-D: Advanced Reasoning with Depth Cameras*.
23. Olson, E. 2011. AprilTag: A robust and flexible visual fiducial system. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
24. Park, J., and Y. Kim. 2014. 3D shape mapping of obstacle using stereo vision sensor on quadrotor UAV. In *AIAA Guidance, Navigation, and Control Conference*.
25. Pons, J. 2016. DroneScan - Airborne Data Collection. <http://www.dronescan.co/>.
26. Ross, S., N. Melik-Barkhudarov, K.S. Shankar, A. Wendel, D. Dey, J.A. Bagnell, and M. Hebert. 2013. Learning monocular reactive UAV control in cluttered natural environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*.
27. Schadler, M., J. Stückler, and S. Behnke. 2013. Multi-resolution surfel mapping and real-time pose tracking using a continuously rotating 2D laser scanner. In *Proceedings of the IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*.
28. Schauwecker, K., and A. Zell. 2014. On-board dual-stereo-vision for the navigation of an autonomous MAV. *Journal of Intelligent & Robotic Systems* 74 (1–2): 1–16.
29. Schmid, K., P. Lutz, T. Tomic, E. Mair, and H. Hirschmüller. 2014. Autonomous vision-based micro air vehicle for indoor and outdoor navigation. *Journal of Field Robotics* 31 (4): 537–570.
30. Schwarz, M., M. Beul, D. Droschel, S. Schüller, A.S. Periyasamy, C. Lenz, M. Schreiber, and S. Behnke. 2016. Supervised autonomy for exploration and mobile manipulation in rough terrain with a centaur-like robot. *Frontiers in Robotics and AI, Section Humanoid Robotics*.
31. Stückler, J., and S. Behnke. 2014. Multi-resolution surfel maps for efficient dense 3D modeling and tracking. *Journal of Visual Communication and Image Representation* 25 (1): 137–147.
32. Tomić, T., K. Schmid, P. Lutz, A. Domel, M. Kassecker, E. Mair, I. Grix, F. Ruess, M. Suppa, and D. Burschka. 2012. Toward a fully autonomous UAV: Research platform for indoor and outdoor urban search and rescue. *IEEE Robotics & Automation Magazine* 19 (3): 46–56.
33. Tripathi, A., G. Raja, and R. Padhi. 2014. Reactive collision avoidance of UAVs with stereovision camera sensors using UKF. In *Advances in Control and Optimization of Dynamical Systems*, 1119–1125.

Author Biographies

Marius Beul received his M.Sc. degree in Electrical Engineering in 2013 from Cologne University of Applied Sciences. Since October 2013, he works as a member of the scientific staff in the Autonomous Intelligent Systems Group at the University of Bonn. His research interests include Aerial Robotics, State Estimation, Path Planning and Control.

Nicola Krombach obtained her M.Sc. degree in Computer Science from Rheinische Friedrich-Wilhelms Universität Bonn in 2016. Since March 2016, she is a researcher in the Autonomous Intelligent Systems Group at the University of Bonn. Her research interests include image processing and visual SLAM.

Matthias Nieuwenhuisen received his Diploma in Computer Science from Rheinische Friedrich-Wilhelms Universität Bonn in 2009. Since May 2009, he is a researcher in the Autonomous Intelligent Systems Group at the University of Bonn. His current research interests include path and motion planning for MAVs.

David Droseschel received a M.Sc. degree in Autonomous Systems from the University of Applied Sciences Bonn-Rhein-Sieg in 2009. Since May 2009, he is a researcher in the Autonomous Intelligent Systems Group of the University of Bonn. His research interests include efficient 3D perception and SLAM.

Sven Behnke received his Diploma in Computer Science from Martin-Luther-Universität Halle-Wittenberg in 1997 and Ph.D. from Freie Universität Berlin in 2002. He worked in 2003 as post-doctoral researcher at the International Computer Science Institute, Berkeley. From 2004 to 2008, he headed the Humanoid Robots Group at Albert-Ludwigs-Universität Freiburg. Since 2008, he is professor for Autonomous Intelligent Systems at the University of Bonn. His research interests include cognitive robotics, computer vision, and machine learning.

Robots Perception Through 3D Point Cloud Sensors

**Marco Antonio Simões Teixeira, Higor Barbosa Santos,
André Schneider de Oliveira, Lucia Valeria Arruda
and Flávio Neves Jr.**

Abstract This chapter brings a tutorial about use of Point Cloud data for the environment perception of mobile robots. Point Cloud is a powerful tool that gives robots the ability to perceive the world around them through a dense measurement. One advantage of this kind of sensor is the large measuring space, with faint or no external light. Although there are several works about Point Clouds, only a few of them speak about how this kind of information can be obtained and what can be extracted. This chapter aims to fill this gap and clarify how Point Clouds can be acquired, processed, transformed between coordinate systems and which these information can be easily extracted using ROS and Matlab. The codes used in this chapter are available in GitHub and can be found at https://github.com/air-lasca/ros_book_point_cloud. The videos developed with the experiments can be seen on YouTube, in Robot LASCA channel that can be accessed at <https://www.youtube.com/channel/UCtgnBqaodQAGtbh0HW9nJEA>.

Keywords Point cloud · Matlab · ROS · Kinect and SR4000

M.A.S. Teixeira (✉) · H.B. Santos · A.S. de Oliveira · L.V. Arruda · F. Neves Jr.
Federal University of Technology—Parana, Av. Sete de Setembro, 3165, Curitiba, Brazil
e-mail: marcoteixeira@alunos.utfpr.edu.br

H.B. Santos
e-mail: higorsantos@alunos.utfpr.edu.br

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

L.V. Arruda
e-mail: lvrarruda@utfpr.edu.br

F. Neves Jr.
e-mail: neves@utfpr.edu.br

1 Introduction

The Point Cloud data (PCD) can be considered as a 3D image, composed by a set of points that enable us to identify its position in the axes X, Y and Z relative to the sensor reference. There are several sensors able to obtain Point Cloud data, for example, the Kinect sensor, SR400 sensor and LIDAR sensor.

Point Cloud has a great number of useful applications in robotics. In [1] they use the PCD obtained with Kinect sensor to develop a Simultaneous Localization and Mapping (SLAM). In other works you can also see the use of PCD in SLAM, as in [2–4]. This kind of data can also be useful for working with Unmanned Aerial Vehicles (UAVs). In [5] made a comparison between two different techniques to get Point Cloud data being one of them to convert a 2D image into a 3D image. Other studies related to PCD and UAV can be seen in [6–8].

The Robot Operating System (ROS) is able to work with the Point Cloud Library (PCL) and thus has support to Point Cloud data through two types of messages: *sensor_msgs::PointCloud* and *sensor_msgs::PointCloud2*. The PCD can be used in complex applications such as those illustrated above, but it is first necessary that you understand how to work with this type of information and how to manipulate it.

This work illustrates the use of Matlab [9] for Point Cloud data processing through the ROS. This software is capable to communicate with ROS and allow an easily handling and processing the data obtained by sensors, because it is a high level programming language (or script). The use of this tool brings great advantages because allows inexperienced users to work with complex messages and the quick development.

This chapter is divided into six subsections. The Sect. 2 presents the depth sensors with focus on two sensors used in this tutorial, Kinect and SR4000. In section about environment configuration will be presented the system configuration used in this tutorial, as well as the detailed procedure of the sensors installation. In the section of examples of use and processing will be explained how to get sensor data and perform its processing, ensuring the correct transformation about coordinate systems.

All source codes shown in this tutorial can be found on GitHub at https://github.com/air-lasca/ros_book_point_cloud address. A tutorial on how to download the GitHub packages can be found in [10]. Packages are divided according to the name of the sections belonging. For example, the package related to installing the sensor SR4000 can be found in the folder

/ros_book_point_cloud/ConfigurationTheEnvornment/.

Videos for each experiment developed during this tutorial will be available on YouTube, on the channel

<https://www.youtube.com/channel/UCtgnBqaodQAGtbh0HW9nJEA> as well as a video showing the quality difference between used sensors.

2 Background

This chapter makes use of various ROS packages and works with several concepts to allow robot perception through 3D point cloud sensors. The most important concept to understand which contains a Point Cloud message. Several libraries and packages will be used in order to obtain and process the Point Cloud.

The Point Cloud Library (PCL) is an open project for processing of 2D/3D images and floating point [11]. The Point Cloud data is a set of points containing three coordinates (X, Y, Z) in a Cartesian plane. All points in this set form an image that can be converted into 2D or 3D, allowing the navigation between the dots and the depth perception.

The Point Cloud data can be obtained in the real world by sensors, such as cameras 3D Time-of-Flight (ToF). These cameras work by emitting a modulated light wave and observing its reflection. The changes between reflected and emitted waves are measured and converted in a distance [12] (as shown in Fig. 1). The Point Cloud data can also be obtained through other sensors such as sonar, laser, and others [13].

The Point Cloud data is presented in 2 different variables in ROS [14]. The difference between the two types of messages can be seen in Fig. 2 and is discussed follows.

1. **sensor_msgs::PointCloud**: is the first version developed for ROS, composed by a set of points that are represented by X-Y-Z coordinates in float format;
2. **sensor_msgs::PointCloud2**: is a novel way to build the Point Cloud data in a set of points having n-D dimensions of different formats (such as integer, float, double and others). This version allows, for example, adding an extra layer to define the color of each point.

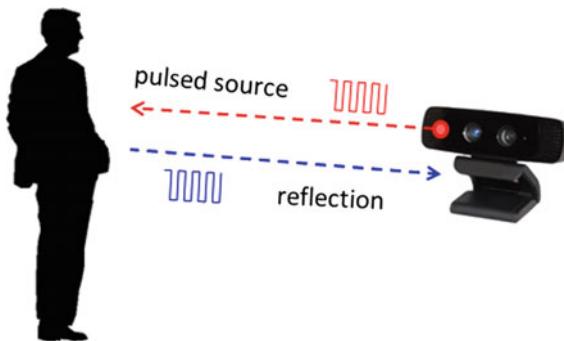


Fig. 1 3D ToF cameras (adapted from [12])

```

sensor_msgs::PointCloud
# Time of sensor data
acquisition, coordinate
frame ID.
Header header

# Array of 3d points. Each
Point32 should be
interpreted as a 3d point
# in the frame given in the
header.
geometry_msgs/Point32[] points

# Each channel should have the
same number of elements as
points array,
# and the data in each channel
should correspond 1:1 with
each point.
# Channel names in common
practice are listed in
ChannelFloat32.msg.
ChannelFloat32[] channels

sensor_msgs::PointCloud2
# Time of sensor data acquisition, and the
coordinate frame ID (for 3d points).
Header header

# 2D structure of the point cloud. If the
cloud is unordered, height is
# 1 and width is the length of the point
cloud.
uint32 height
uint32 width

# Describes the channels and their layout in
the binary data blob.
PointField[] fields

bool is_bigendian # Is this data
bigendian?
uint32 point_step # Length of a point in
bytes
uint32 row_step # Length of a row in
bytes
uint8[] data # Actual point data,
size is (row_step*height)

bool is_dense # True if there are no
invalid points

```

Fig. 2 Difference between *sensor_msgs::PointCloud* and *sensor_msgs::PointCloud2*

3 Common Depth Sensors for Robot Perception

3.1 Microsoft Kinect

The Kinect sensor is widely used in robotics due to its low cost. It has a traditional RGB camera with a resolution of 640×480 and a camera 3D structured light with a resolution of 320×240 points. There are two models of Microsoft Kinect: the Kinect 360 or Kinect 1 (first generation) and the Kinect One or Kinect 2 (second generation), as shown in Fig. 3. In this tutorial, we will focus on Kinect 360 due to its low cost and using standard USB 2.0 to have a larger compatibility with embedded systems actually applied to robotic applications.

Table 1 brings the main differences between the two models of Kinect. For Point Cloud data, the main difference is the resolution of depth camera, because with



Fig. 3 Different versions of Microsoft Kinect

Table 1 Comparison between the generations of Microsoft Kinect

Feature	Kinect 1 (360)	Kinect 2 (one)
Color camera	$640 \times 480 @30$ fps	$1920 \times 1080 @30$ fps
Depth camera	320×240	512×424
Max depth distance	~ 4.5 M	~ 4.5 M
Min depth distance	40 cm	50 cm
Horizontal field of view	57°	70°
Vertical field of view	43°	60°

higher the resolution is obtained a more detailed perception. On the other hand, with a higher resolution is needed transmit more information and this can overburden network traffic (mainly over Wi-Fi) in a architecture with multiple ROS node.

The original Kinect was developed for the Xbox 360 console in November 2010 with model number 1414. The model 1473 now has a Kinect-for-Xbox and Kinect-for-Windows version. Kinect 2 was presented in November 2013. Other differences can be found between the models, such as the serial number assigned to the components. More differences about Kinect models can be found in [15]. The version used for this tutorial is a version for the Xbox 360 console, with model number 1473. There may be some divergence from the methods presented in the chapter with use of another model.

3.2 SR4000

Other sensors can be used to catch the Point Cloud data and one of them is the SR4000 sensor. The company Mesa Imaging that in 2014 was bought by Heptagon Company initially developed this sensor. The company's data can be observed in [16].

The SR4000 sensor was developed with a focus on industry. It uses the Time-of-Flight (ToF) technology to get data, similar to what happens with the Kinect One sensor. A sensor photo can be seen in Fig. 4.

The sensor settings are shown in Table 2. Contrary to what happens with Kinect, the SR4000 does not have an integrated RGB camera and so it gives only the Point Cloud data, without the extra layer with RGB. In contrast, the degree of confidence of Point Cloud of SR4000 sensor is exceeds that of the Kinect.



Fig. 4 SR4000 sensor

Table 2 SR4000 data-sheet

Feature	SR4000
Depth camera	176 × 144
Max depth distance	~5 M
Min depth distance	30 cm
Horizontal field of view	Standard: 43.6°. Wide: 69°
Vertical field of view	Standard: 34°. Wide: 56°

The 3D ToF SR4000 camera can be acquired with standard or wide viewpoint. The model used during this tutorial is a wide model, although the steps that will be presented can be also followed for the standard model.

4 Configuring the Environment

This section aims to prepare the work environment with the tools used during the tutorial. The main tools used are:

- Ubuntu 14.04 LTS;
- Ros Indigo Igloo.

Robot Operating System (ROS) is a system with several tools and libraries to program robots, linked with commercial components and robots, in order to assist in writing software for robots [17]. The ROS allows compatibility with multiple operating systems (to see if your system is compatible, see [18]). The recommended

platform to use the ROS is the Ubuntu, which is a Linux open source system [19] considered stable. By the time of this tutorial, the latest version of Ubuntu is 16.04 LTS.

The version of Ubuntu used in this tutorial is 14.04 LTS because this has the Long Time Support (LTS) and it is recommended for Ros Indigo Igloo. The download of Linux Ubuntu can be found in [20] and a complete tutorial on how to install the system is shown in [21].

There are several versions of the ROS, being *Kinetic Kame* the most recent version at the moment of writing this paper. However the version used in this tutorial is the *Ros Indigo Igloo* that is very stable. A complete tutorial on how to install *Ros Indigo Igloo* can be found in [22].

In this section will be divided into four subsections. In subsection *Kinect*, the sensor Kinect will be presented, as well as the types of sensors contained therein. In the subsection *Install Kinect*, will be presented the procedure needed to use the Kinect with ROS. In subsection *SR4000*, the sensor SR4000 and its settings will be displayed. In subsection *Install SR4000*, will be presented the methods allowing the use of the sensor with ROS.

4.1 Install Kinect

The used system configurations are:

- Ubuntu 14.04 LTS;
- Ros Indigo Igloo;
- Kinect 360 or Kinect 1, model 1473.

Packages *freenect-camera* [23] and *freenect-launch* [24] are used together with the *libfreenect-dev* driver [25] for the purpose of obtaining the data provided by a Kinect sensor. This package has some modifications of the *openni_launch*[26] package and runs based on libfreenect driver. Some examples of how to install Kinect 1 can be found on the official ROS website or on other sites such as [27, 28].

To install *freenect-launch*, first is need to install the *libfreenect-dev* library. For this, open the console and type:

```
1 $ sudo apt - get install libfreenect - dev
```

Next *freenect-camera* package and *freenect-launch* package will be installed. To install *freenect-camera* package type:

```
1 $ sudo apt - get install ros - indigo - freenect - camera
```

To install *freenect-launch* package type:

```
1 $ sudo apt - get install ros - indigo - freenect - launch
```

Now the environment is ready for the Kinect. The previous two steps can be skipped if you run the following command to install all the packages for the *freenect*

even not all packages will be used. This procedure is only recommended for systems that do not worry use of space on the disk, the code is:

```
| $ sudo apt - get install ros - indigo - freenect *
```

To test if everything is working correctly, type the console command.

```
| $ roslaunch freenect_launch freenect.launch
```

It is possible to occur the following error.

```
| $ No devices connected .... waiting for devices to connect
```

This error can occur because the ROS not be allowed to read the USB ports in the computer, to solve this problem just type:

```
| $ sudo chmod -R 777 / dev
```

If the problem persists, it may be because the Kinect is not connected to an external power supply. The charger used for this tutorial is similar to that shown in Fig. 5. When the sensor is incorporated in a robot, it becomes possible to obtain energy through a 12 V battery.

Starter again the *freenect_launcher* command and a multitude of topics should appear, the topics for the camera always start with the initial */camera/*.

```
| $ rostopic list
```

Fig. 5 Converter used to connect the Kinect on the computer



If everything is correct, the following topics will be appear:

1. **/camera/depth/points**: this topic provides a set of points in *sensor_msgs/PointCloud2* format. While using *PointCloud2* format, this topic does not provide additional information such as the color in RGB format. Figure 6 bring an example from the topic;
2. **/camera/depth_registered/points**: this topic has a message such as *sensor_msgs/PointCloud2* fused to RGB camera data, forming a set of data known in literature to as RGB-D. An image can be seen in Fig. 7.
3. **/camera/rgb/image_color**: this topic contains a message like *sensor_msgs/Image* that it is an RGB image. An example of this topic can be seen in Fig. 8.

To facilitate this work, several copies of launch file based on the original can be created. To do this, the original launch file is copied to the new *catkin/srs* folder. First, to copy the launch file to the desired folder, enter the command:

Fig. 6 An example from the topic */camera/depth/points*



Fig. 7 An example from the topic */camera/depth_registered/points*

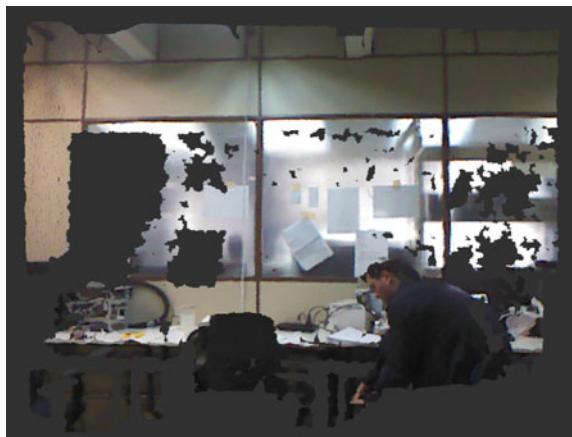




Fig. 8 An example from the topic /camera/rgb/image_color

```

1 <launch>
2   <include file="$(find freenect_launch)/launch/freenect.launch">
3     <arg name="rgb_processing" value="true" />
4     <arg name="ir_processing" value="false" />
5     <arg name="depth_processing" value="true" />
6     <arg name="depth_registered_processing" value="true" />
7     <arg name="disparity_processing" value="false" />
8     <arg name="disparity_registered_processing" value="false" />
9   </include>
10 </launch>
```

Fig. 9 kinect.launch

```

1 $ cp /opt/ros/indigo/share/freenect_launch/launch/
      freenect.launch ~/catkin_ws/src/kinect.launch
```

A copy of freenect.launch can now be found in `~/catkin_ws/src`. The file is edited to minimize the number of topics displayed. Thus, eliminating the unnecessary use of circulating data on the network and reduce the number of processing.

The file can be edit using the editor of your choice, for example the graphical environment *gedit*, and with a *nano* in console environment. To edit the file created, type the following command (changing the word editor by the name of the desired editor).

```

1 $ editor ~/catkin_ws/src/kinect.launch
```

Edit the file in order to look like the following in Fig. 9.

Finally, to run the launch file created, simply type the command:

```

1 $ roslaunch ~/catkin_ws/src/kinect.launch
```

Now you can have access to the main topics provided by *freenect* and Kinect sensor. From now on, we will deal with the acquisition and processing of data. Examples involving data collection and maintenance will be primarily developed using the

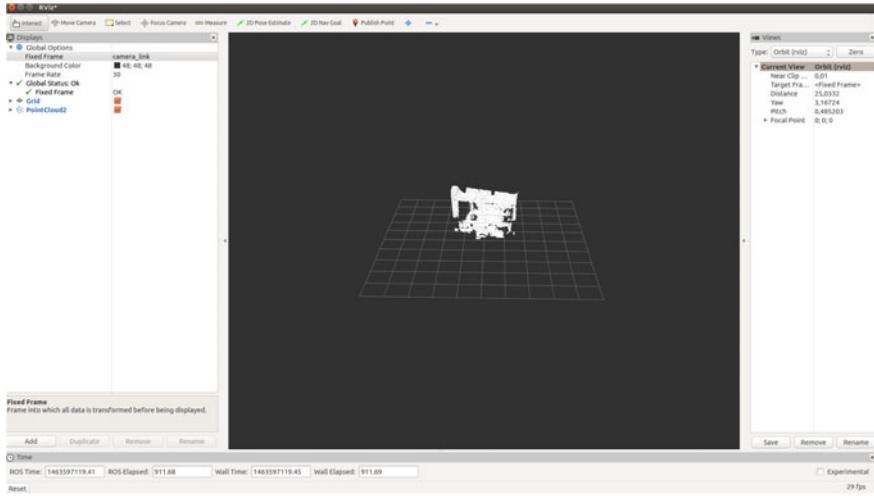


Fig. 10 Rviz window with the topic */camera/depth/points*

Matlab tool, except in cases where it requires the need to develop some code in C or creating launcher's, as it has already occurred.

The Rviz is used to easily view the topics. The Rviz is a ROS tool with many interesting features such as the ability to view the outputs of a topic. The Rviz will be fully explored during this tutorial. To start, let us open Rviz by typing at the terminal:

```
$ rviz
```

For everything to work properly, make sure that this *kinect.launch* has been initialized and that the Kinect is plugged into the computer. Now to open Rviz, click the *Add* button, followed by choosing the tab *by topic* in the floating opened window. Within this tab, you can see all the currently existing topics in ROS. Click */camera* then */depth*, then in */points*, select the *PointCloud2* and click *OK*. In *Global Options*, click in *fixedframe*, and select the option *camera_link*. After all these steps are carried out, the PCD from the Kinect is successfully displayed as shown in Fig. 10.

Repeat the procedure to the topics */camera/depth_registered/points* and */camera/rgb/image_color*. Explore the features of Rviz, switch between topics displayed, and change the size of the dots among other activities, in order to better understand the tool.

4.2 Install SR4000

To use the camera 3D ToF SR4000, the *libmesasr-dev-1.0.14-748.amd64* [29] driver was used along with the ROS *swissranger_camera* package [30]. This ROS package is no longer in the official repositories of ROS, therefore it will be available during

Table 3 Difference between drivers

Name	Recommended system
libmesasr-dev-1.0.14-747.i386.deb	32 bits
libmesasr-dev-1.0.14-748.amd64.deb	64 bits

the tutorial at GitHub. If you prefer the package *cob_camera_sensors* can be used, a tutorial on how to install can be found in [31] and how to use it for the SR4000 sensor can be found in [32].

For this session, it is necessary to download a specific folder from GitHub. A tutorial on how to download all files can be viewed at Background. To download the full project folder, open a Linux terminal and navigate to the chosen place where the package will be saved, then type:

```
$ git clone https://github.com/air-lasca/roslaunch_point_cloud
```

Inside this new folder, a folder for each subsection of this chapter has been created. All files required for installation and SR4000 camera setup are located in the */Configuringtheenvironment*.

The first step is installing the driver responsible for making the communication between the computer and the camera. This driver has two versions, one for 32-bit and other for 64-bit computers. The two drivers are available on GitHub of this tutorial. Table 3 establishes the recommended system for each driver.

The driver can also be found in the manufacturer's page: <http://hptg.com/industrial/>. In this page, navigate to the SR4000 camera and select the Downloads tab. These can be installed in two ways, one of them is through a double click on the file, and the second is through the command line in Linux terminal. To install from the command line, open the terminal and type:

```
$ sudo dpkg -i libmesasr-dev-$<version>.deb
```

This driver is required to perform the communication between the camera and the computer. After install this driver, the next step is to install the package *swissranger_camera*, this package makes the conversion of information from the camera in understandable topics by ROS.

The package *swissranger_camera* is not more present in the official repository, but a copy with slight modifications is within the folder */Configuringtheenvironment/InstallSR4000* with *swissranger_camera* name. To install the package, open a Linux terminal and navigate to the */Configuringtheenvironment/InstallSR4000* folder, then enter the command:

```
$ cp -R swissranger_camera ~/catkin_ws/src/
```

This command generates a copy of the folder *swissranger_camera* to *catkin_ws/src*. Once this is done, the next step is to compile the file, in a terminal type:

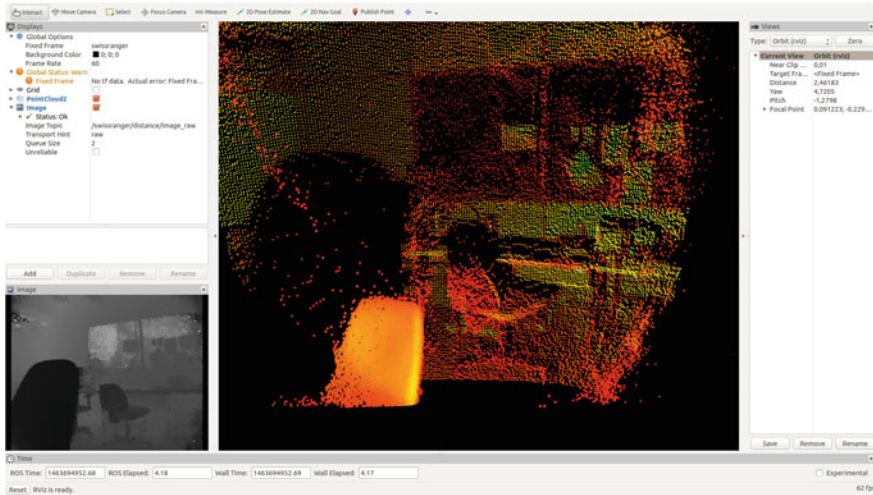


Fig. 11 Rviz window with sensor information SR4000

```
1 $ cd ~ / catkin_ws /
2 $ catkin_make
```

If an error occurs permission, type:

```
1 $ cd ~ / catkin_ws / src /
2 $ sudo chmod -R 777 swissranger_camera
```

Re-enter the command:

```
1 $ cd ~ / catkin_ws /
2 $ catkin_make
```

All packages required to use the camera SR4000 are already installed and configured. The next step is to connect the camera to a PC by USB interface. Remembering as with Kinect, the camera needs an external source to run. A tutorial on how to plug the camera to the computer can be found in [33].

With the camera connected to the computer, run the package *swissranger_camera* and verify that occurred as expected. For this open a Linux terminal and type:

```
1 $ roslaunch swissranger_camera sr_usb.launch
```

The Rviz window, as shown in Fig. 11, should be opened. In this window, you can view the Point Cloud data in the center, and in the right the same Point Cloud data image in gray scale. The package *swissranger_camera* does this conversion and it is a user option.

The following error can occur:

```
1 $ [ 1 4 6 3 6 9 5 5 4 8 . 8 4 2 6 7 9 1 7 1 ] : Exception thrown while
  connecting to the camera : [ SR::open ] : Failed to
  open device !
```

This error may be caused by the fact of the sensor is not connected to power. Checks on the back of the sensor if the light is flashing green. If it is flashing, it means that the sensor is working properly. Make sure the USB cable is connected in the right place. Another cause for this failure can be the fact that the user does not have sufficient permission to access the camera interface. Entering the following command can easily solve this problem:

```
$ sudo chmod 777 /dev/tty*
```

This command gives reading and writing permission for all tty devices connected to the computer. If you know which device, just replace `tty*` by the correct port name. Then, again run the command:

```
$ roslaunch swissranger_camera sr_usb.launch
```

If everything goes well, the camera is already in place and ready for use. To avoid the hassle of giving permission to the device at a time, you can add the command `~/.bashrc`. In this way, every time you open a new Linux terminal, the permissions will be given and the device will be ready for use. To do this, type the command:

```
$ sudo < editor > ~/.bashrc
```

With open `.bashrc` file add at the end the following code.

```
$ sudo chmod 777 /dev/tty*
```

We will develop a new custom launcher. If you install the package on a computer running Linux server, it interesting disables the opening of Rviz, since the system does not have graphics support. To copy the original launcher to our `/catkin_ws/src`, type:

```
$ cp ~/catkin_ws/src/swissranger_camera/launch/sr_usb.launch ~/catkin_ws/src/sr4000.launch
```

This command copies the original launch file to the folder `/catkin_ws/src` with the name of `sr4000.launch`. This code in the file can be seen in Fig. 12. The code responsible for the sensor opening is between the fourth and seventh line. The lines from nine to eleven have the codes responsible for opening the Rviz, if you want to open the Rviz just remove the lines. The new launch file will look like the Fig. 13. To test the launch file, type:

```
$ roslaunch ~/catkin_ws/src/sr4000.launch
```

In a new terminal, type:

```
$ rostopic list
```

Note that several topics were created. It is noteworthy that the SR4000 camera only has the type sensor 3D ToF. The content of other topics are nothing more than the Point Cloud data with some kind of processing. Some of the topics deserve to be highlighted, by providing interesting content and they are:

```

1 <!-- -- mode: XML -->
2
3 <launch>
4   <node pkg="swissranger_camera" type="swissranger_camera" name="swissranger"
5     output="screen" respawn="false">
6     <param name="auto_exposure" value="1" />
7   </node>
8
9   <!-- ROS visualizer -->
10  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find swissranger_camera
11    )/cfg/swissranger.rviz" />
12</launch>

```

Fig. 12 sr_usb.launch

```

1 <launch>
2   <node pkg="swissranger_camera" type="swissranger_camera" name="swissranger"
3     output="screen" respawn="false">
4     <param name="auto_exposure" value="1" />
5   </node>
6 </launch>

```

Fig. 13 sr4000.launch

1. **/swissranger/pointcloud_raw**: this topic provides a message of type *sensor_msgs/PointCloud* that can be considered one of the main topics. All other information are processed from the Point Cloud data present in this topic;
2. **/swissranger/pointcloud2_raw**: this topic gives a message from *sensor_msgs/PointCloud2*. The only difference between this topic and */swissranger/pointcloud_raw* is that this topic is in PointCloud2 format. In some situations, it is necessary to use a message such as PointCloud1 and other PointCloud2 of message type, having two topics to avoid the need for possible conversions. An example of an image of both threads can be seen in Fig. 14;

Fig. 14 PointCloud2.
topic: */swissranger/pointcloud2_raw* **message of type:** *sensor_msgs/PointCloud2*

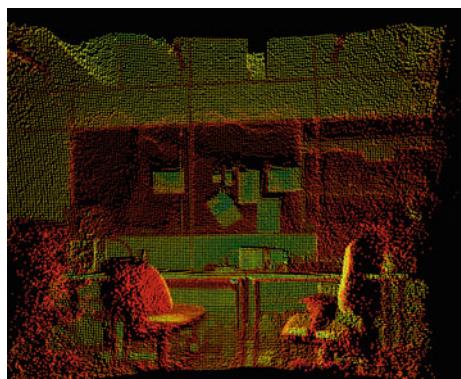


Fig. 15 Image. **topic:** /swissranger/distance/image_raw
message of type: sensor_msgs/Image



3. **/swissranger/distance/image_raw:** this topic brings a message of *sensor_msgs/Image*. This is a conversion of the Point Cloud data in to gray-scale, leaving dark objects near and clear objects away. An example can be seen in Fig. 15;
4. **/swissranger/confidence/image_raw:** this topic provides a message of *sensor_msgs/Image*. This topic is processed from the Point Cloud data. Jerky movements leave a trail on the image, recording a movement. An example can be seen in Fig. 16;
5. **/swissranger/intensity/image_raw:** This topic gives a message of *sensor_msgs/Image* and it is processed from the Point Cloud data. This topic brings the image intensity. A figure of this topic can be found in Fig. 17.

Fig. 16 Confidence. **topic:** /swissranger/confidence/image_raw **message of type:** sensor_msgs/Image



5 Examples of Point Cloud Processing

This section aims to demonstrate and explain possible processing techniques to be performed using the SR4000 sensor (the techniques in this section can be executed also using Kinect). Matlab will be used to perform all processing and to view the results. The results can also be viewed on Rviz if desired.

5.1 Commands in Matlab

This subsection aims to bring the main commands used during this chapter. The commands explained here are from Matlab R2015a tool, conflicts can appear if the commands are used with other versions. First, the main commands involving the PCL in Matlab will be explained. The methodology will be used as follows: First, the explanation of the command, and then the command is used. The following command closes the open connection to the ROS, if one is active.

```
| rosshutdown ;
```

The next command opens a communication with the ROS, providing information such as the content of topics.

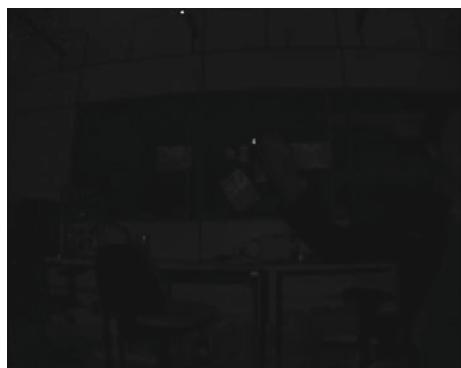
```
| rosinit ;
```

The command *rossubscriber* allows you to take the topic of reference, the variable topic now becomes the chosen type, having the same format.

```
| topic = rossubscriber('Topic Name');
```

The value variable receives the next data read by the topic in the message format, in the case of Point Cloud data, the format is *sensor_msgs/PointCloud* or *sensor_msgs/PointCloud2*.

Fig. 17 Intensity. **topic:**
/swissranger/intensity/
image_raw message of type:
sensor_msgs/Image



```
| value = receive(topic);
```

The command *readXYZ* transfer for the *xyzData* only the coordinates of each point contained in the Point Cloud data. This command is extremely important and widely used, because without it, working with the message type *sensor_msgs/PointCloud* is difficult enough.

```
| xyzData = readXYZ(pcloud);
```

The command *scatter3* displays a picture containing the Point Cloud data, and you can browse it in Matlab without the need to use the Rviz. This command only works with variables of type *sensor_msgs/PointCloud2*.

```
| scatter3(value);
```

Next command also enables the visualization of Point Cloud data in Matlab, but different from what happens with the *scatter3*, this command allows you to view data in XYZ matrix format.

```
| showPointCloud(xyzData);
```

The command *rospublisher* creates a variable that can be seen in the ROS. You should specify the topic name and type of message, such as:
RosPub = rospublisher('/pcl','sensor_msgs/PointCloud').

```
| RosPub = rospublisher('Topic NameT','Message Type')
;
```

Next command allows creating a variable the same type of existing message ROS, simply specify the message name, such as:

```
| msg = rosmessage('RosPub');
```

Also is possible to specify the message type, as:

```
| msg = rosmessage('sensor\_\_msgs\_\_PointCloud')
```

Where the variable msg become a variable of type *sensor_msgs/PointCloud*.

It is also possible to place a sample in place on the variable name, for example:

```
| msg = rosmessage('name of message');
```

The following command sends the contents of the variable created for the topic created. After this command, it becomes possible to verify the ROS data sent, as in Rviz if the format is compatible with the tool. An example use of this command is *send(RosPub,msg)* that is being sent to the topic RosPub the existing content in the msg variable.

```
| send(topic created, msg created);
```

5.2 ROS Subscriber with Matlab

The next step is the data collect and process step, as mentioning earlier, the tool used for this is Matlab. Matlab allows direct communication with the Ros, and it facilitates data post processing. First, to collect and display the results of Kinect sensor, open Matlab and type the command like the Fig. 18.

The code will be briefly explained. The command in the first line closes the connection to the ROS, if any old connection is still alive. In line two, a new connection with the ROS is made, it is through this command becomes possible to visualize ROS in Matlab. In line three the topic */camera/depth/points* is transferred to the variable *topic*, where it can be handled the same way as a topic. In line four, the variable *pcloud* is receiving the next value read by the variable *topic*, in this case the variable *pcloud* if has a variable of type *sensor_msgs/PointCloud2* specifies the ROS. On line five displays an image *Pcl* obtained from the sensor, the image is similar to the view in Fig. 19.

The same command can be used for other messages from the Kinect, with slight changes in time to display images. The codes for the data collection of topics */camera/depth_registered/points* and */camera/rgb/image_color* can be seen in Fig. 20. The result can be seen in Fig. 21. Note that different from what it has happened with the topic */camera/depth/points*, this figure has color. It is characteristic of PointCloud2 and it allows adding a dimension in the data matrix containing any type of information, such as color, which is the case.

The process changes to obtain data from the topic */camera/rgb/image_color*, this is because the message type is different. The message now is type *sensor_msgs/Image*. One more step needs to be used to convert the collected message topic in an image understandable by Matlab. The code with the explanations is show in Fig. 22, the Fig. 23 brings the result.

The procedure for get the Point Cloud data in Matlab for sensor SR4000 is similar to that used for the Kinect, because it is the same message type in ROS. The code for this procedure can be seen in Fig. 24, the Fig. 25 brings the result.

For the data of the topics */swissranger/distance/image_ra*, */swissranger/confidence/image_raw* and */swissranger/intensity/image_raw* the procedure is the same because all topics are the same type of message, which is the *sensor_msgs/Image*. The code in Matlab to get the message of this type was introduced. Please see the section or the package available on GitHub for this section.

It is desirable that the reader already knows how to initialize the Kinect sensor or sensor SR4000, depending on which sensor the reader intends to use. The user

```

1 rosshutdown;
2 rosinit;
3 topic = rossubscriber(' /camera/depth/points');
4 pcloud = receive(topic);
5 scatter3(pcloud);

```

Fig. 18 Command in Matlab to get the Point Cloud data from the topic */camera/depth/points*

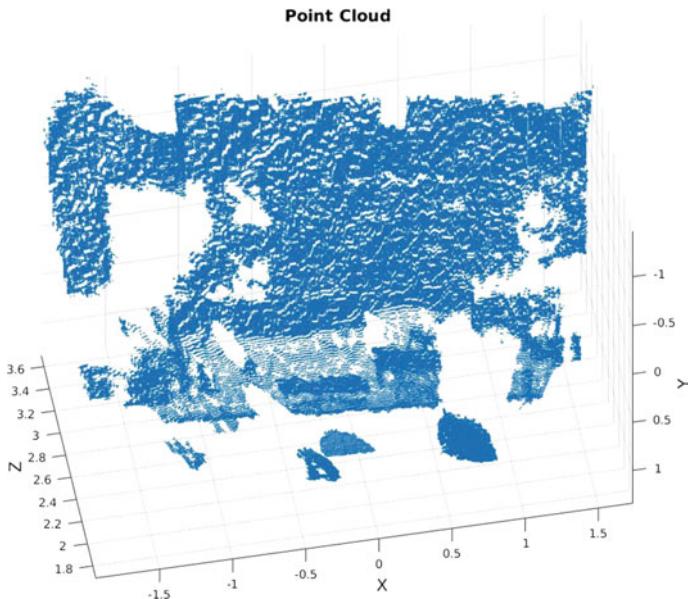


Fig. 19 Viewer of Point Cloud from topic */camera/depth/points* in Matlab

```

1 %Close the ROS communication
2 rosshutdown;
3 %Open the ROS communication
4 rosinit;
5 %transfer the topic for the variable depth_registered
6 depth_registered = rossubscriber(' /camera/depth_registered/points' );
7 %Receives the value of the topic
8 pcloud = receive(depth_registered);
9 %Displays data
10 scatter3(pcloud);

```

Fig. 20 Command in Matlab to get the Point Cloud data from the topic */camera/depth_registered/points*

must already know how to use Rviz to view the topics, know the difference between the types of messages provided by the sensors and know which topic belong to each sensor, and know how to get data in Matlab. If any of these ideas is not clear, please review the corresponding section.

5.3 ROS Publishing with Matlab

This subsection aims to create a function that converts the array in XYZ format in a message of type *sensor_msgs/PointCloud*. The function will be developed in Matlab

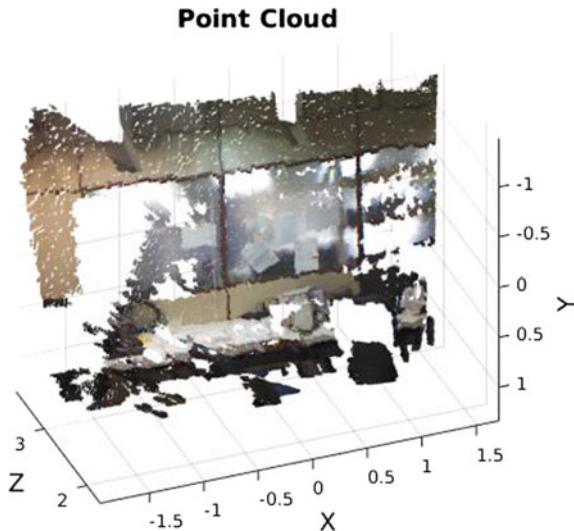


Fig. 21 Viewer of Point Cloud from topic */camera/depth_registered/points* in Matlab

```
1 %Close the ROS communication
2 rossshutdown;
3 %Open the ROS communication
4 rosinit;
5 %transfer the topic for the variable image_color
6 image_color = rossubscriber('camera/rgb/image_color');
7 %Receives the value of the topic
8 sensor_msgs_Image = receive(image_color);
9 %Read the Image data
10 imageFormatted = readImage(sensor_msgs_Image);
11 %View figure
12 imshow(imageFormatted);
```

Fig. 22 Command in Matlab to get the image from the topic */camera/rgb/image_color*



Fig. 23 Viewer of image from topic */camera/rgb/image_color* in Matlab

```

1 %Close the ROS communication
2 rosshutdown;
3 %Open the ROS communication
4 rosinit;
5 %transfer the topic for the variable pointcloud_raw
6 pointcloud = rossubscriber('/swissranger/pointcloud2_raw');
7 %Receives the value of the topic
8 pcloud = receive(pointcloud);
9 %Displays data
10 scatter3(pcloud);

```

Fig. 24 Command in Matlab to get the Point Cloud data from the topic of SR4000 /swissranger/-pointcloud_raw

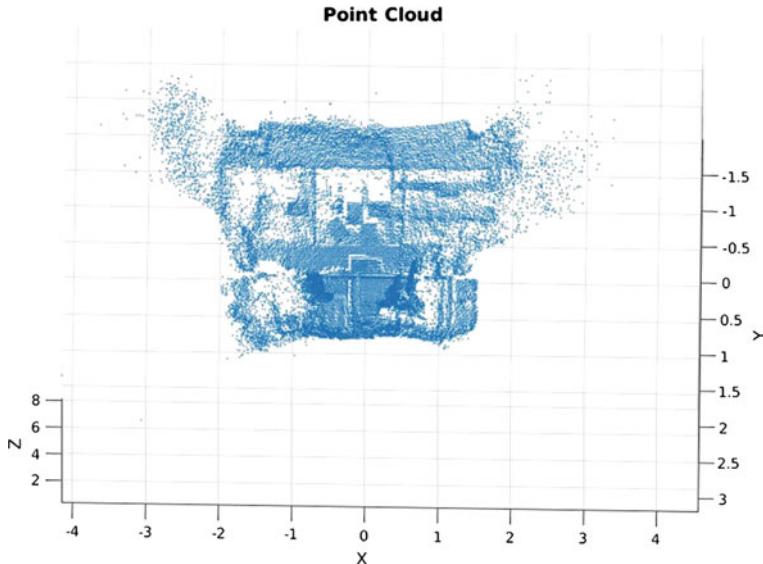


Fig. 25 Viewer of Point Cloud from topic of SR4000 /swissranger/pointcloud_raw in Matlab

and it will be used for other experiments, so that the result of the experiments is visible in Rviz.

For the experiments and tests presented in this subsection, SR4000 sensor shall be used. First the function code will be presented in Fig. 26 after an explanation of the code and how to use the function. An explanation of this function is given in its header. It has as input the matrix containing the points on the XYZ format, the desired name for the topic in ROS, the name of the reference and the time to leave visible the topic.

Let us explanation about the code. Line 10 defines the function name, as well as the input and output attributes. This function will have as output a message such *sensor_msgs/PointCloud* containing conversion made of XYZ for the message. This output is interesting in other situations.

```

1 %This function convert a NX3 array dimension in a message
2 %of type sensor_msgs/PointCloud understandable in ROS.
3 %xyz = Nx3 matrix:
4 %Nx1 = X;
5 %Nx2 = Y;
6 %Nx3 = Z;
7 %TopicName = topic name in ROS.
8 %FrameName = Name of reference, examples: map, World.
9 %Time = Time the topic had been visible.
10 function msg = XYZ_to_sensor_msgs_PointCloud(xyz,TopicName,FrameName,Time)
11
12 xyzvalid = xyz(~isnan(xyz(:,1)),:);
13 PCLmensage = rosmessage('geometry_msgs/Point32');
14 for i=1:size(xyzvalid,1)
15     PCLmensage(i).X = xyzvalid(i,1);
16     PCLmensage(i).Y = xyzvalid(i,2);
17     PCLmensage(i).Z = xyzvalid(i,3);
18 end
19
20 msg = rosmessage('sensor_msgs/PointCloud');
21 msg.Header.FrameId = FrameName;
22 msg.Points = PCLmensage;
23 pub = rospublisher(strcat('/' ,TopicName), 'sensor_msgs/PointCloud');
24 send(pub,msg);
25 pause(Time);
26 end

```

Fig. 26 Function XYZ_to_sensor_msgs_PointCloud

Line 12 removes undesirable values such as NaN, from the variable *XYZ*. These values can arise when using the command *readXYZ* and a message of type *sensor_msgs/PointCloud2*. The sensor possessed a defined resolution, but if some points are not unavailable during the time of capture of information, the sensor returns *Nan*. This situation is not desirable since the array typically has a too high size, discarding these points reduces the processing cost.

Line 13 creates a message of type *geometry_msgs/Point32*, to be able post a message of type *sensor_msgs/PointCloud* it is necessary convert each point of *XYZ* in a message type *geometry_msgs/Point32*, which is the activity of this function.

The Lines 14 to 18 create a repeating loop. This loop is intended to go through the whole *XYZ* array and convert each set of points in a message type *geometry_msgs/Point32*.

Line 20 creates the message type *sensor_msgs/PointCloud*, this message will be published in ROS and it will be the returning message. Line 21 defines the frame according to the name sent to the function. Line 22 transfers the created variable *PCL1mensage* of type *geometry_msgs/Point32* for the message created in line 20.

Line 23 creates the visible topic in ROS with the name you specify when calling the function. The topic is the type *sensor_msgs/PointCloud*. The 24 line sends the created message to the topic, at this time the information processed by the function will be available in ROS.

Line 25 breaks code execution. Refers to the desired pause time for the life topic. This pause is necessary because as the topic has been created in the function when the function end the topic die. A change in the code can be done if desired, by simply remove the lines 23:24 and post the topic in your code.

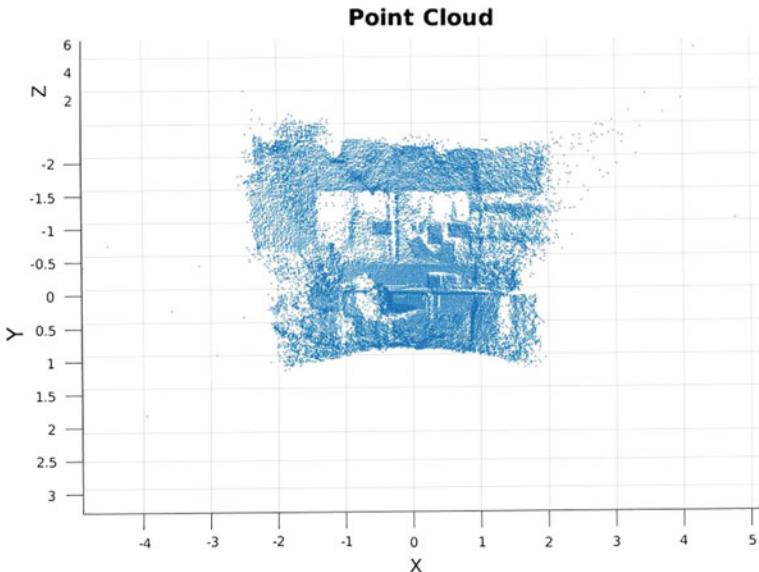


Fig. 27 Point Cloud data obtained of the topic */swissranger/pointcloud2_raw*

We will test the created function. The function can be obtained through the package available on GitHub, or just copying the text previously provided in a text file, and save it with *XYZ_to_sensor_msgs_PointCloud.m* name in your workspace. With open Matlab and function saved in your workspace and being recognized by Matlab, type in the command window:

```
1 rosshutdown;
2 rosinit;
```

These commands can close any open communication with the ROS, and open a new statement. We now get the Point Cloud data through a topic of ROS, for this type:

```
1 topic = rossubscriber ('/swissranger/pointcloud2 \
2 _raw');
pcloud = receive (topic);
```

Replacing */swissranger/pointcloud2_raw* by the desired topic, if you are using the sensor SR4000 any changes need to do. Let us see the PCL before processing, for this type the following command. A window appears with the PCL, similar to Fig. 27.

```
1 scatter3 ( pcloud );
```

The next step is the extraction of the Point Cloud data points, for that enter:

```
1 xyz = readXYZ ( pcloud );
```

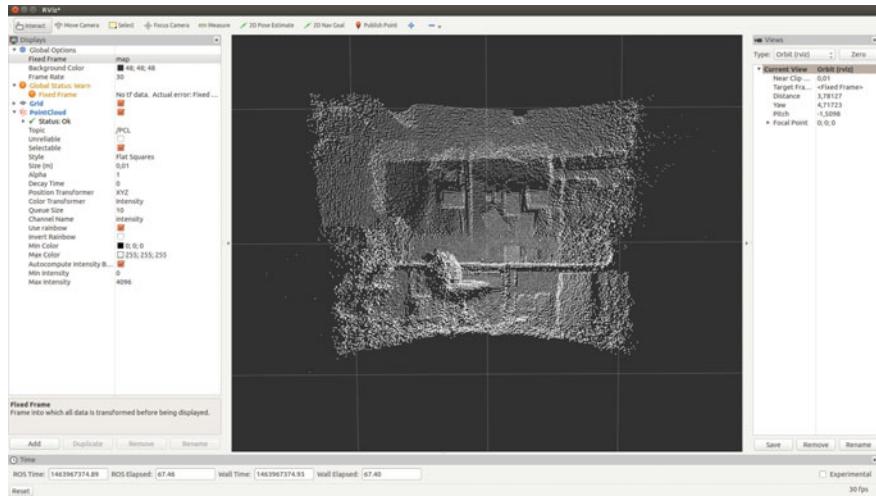


Fig. 28 Point Cloud data created with the function *XYZ_to_sensor_msgs_PointCloud* and published on the topic */PCL*

The variable *XYZ* now has a NX3 array containing all the Point Cloud data points. N refers to the resolution of the sensor by multiplying the number of lines multiplied by columns. In the case of the sensor SR4000, N is equal to 176×144 resulting in 25344 lines containing the XYZ data in each line.

The next step is to call the function for this type:

```
XYZ_to_sensor_msgs_PointCloud( xyz , ' PCL ' , ' map '
, 500 ) ;
```

This command will call the function *XYZ_to_sensor_msgs_PointCloud* sending our variable *XYZ*. The name set for the new topic is *PCL* and the desired orientation is in reference to the map, lasting 500 s. The choice of high duration time fact becomes possible to display the topic in Rviz. Open Rviz, add the topic and change the orientation of the application to map, something similar to Fig. 28 will appear.

In this subsection, read and re-create the Point Cloud data without any change steps will be carried out. This methodology was used to illustrate the operation of the function. In the next subsections, Point Cloud data will be modified before being published.

5.4 Creating Markers

The Markers are a type of existing messages in ROS. It can be found isolated in a single Marker as with message *visualization_msgs/Marker* [34] or a vector containing multiple Markers as with *visualization_msgs/MarkerArray* message [35]. The Markers can be viewed in Rviz as with Point Cloud data [36], the conversion of the Point Cloud data in Markers is very useful for robotics, mainly on the creation

```

1 %create a marker
2 function [marker] = marker(x,y,z,r,g,b,type,scale,id,frame)
3
4 %Create a marker
5 marker = rosmessage('visualization_msgs/Marker');
6
7 %Set the type of the marker
8 marker.Type = type;
9
10 % Set the pose of the marker.
11 marker.Pose.Position.X = x;
12 marker.Pose.Position.Y = y;
13 marker.Pose.Position.Z = z;
14
15 %set the orientation of the marker
16 marker.Pose.Orientation.X = 0.0;
17 marker.Pose.Orientation.Y = 0.0;
18 marker.Pose.Orientation.Z = 0.0;
19 marker.Pose.Orientation.W = 1.0;
20
21 % Set the scale of the marker
22 marker.Scale.X = scale;
23 marker.Scale.Y = scale;
24 marker.Scale.Z = scale;
25
26 % Set a RGB color of the marker
27 marker.Color.R = r;
28 marker.Color.G = g;
29 marker.Color.B = b;
30
31 % Set the transparency
32 marker.Color.A = 1;
33
34 % Set the frame
35 marker.Header.FrameId = frame;
36
37 % Marker id
38 marker.Id = id;
39
40 end

```

Fig. 29 Function *marker*

of maps, the *OctoMap* [37] for example, can be considered a set of Markers. This subsection is not intended to create a *OctoMap*, but to show the conversion of a Point Cloud data in Markers.

We will create two functions in this subsection, a function to create an isolated Marker and a function to convert Point Cloud data into Makers. It is interesting to use two functions for ease of learning, as with an only function, the code would become extensive. Let's start with the code needed to create a single Marker as shown in Fig. 29.

The function is annotated and explanatory. The result of the function is a single Marker. The inputs function are **x**, **y**, **z**; **r**, **g**, **b**; **type**, **side**, **id** and **frame** referring:

1. **x, y, z:** Refers to the position of Marker in the case of a conversion of Point Cloud data refers to existing data in the XYZ;
2. **r, g, b:** Refers to the color desired for the market, this color is in the RGB format;
3. **type:** Refers to the type of marker. The Marker can take various formats, the Table 4 brings the value of the variable type must assume for conversion to Point Cloud data, together with the format of Marker;

Table 4 The variable *type* value and corresponding format

Type	Format
0	Arrows
1	Cube
2	Sphere
3	Cylinder

4. **scale**: Refers to Marker size;
5. **id**: It refers to the number id the Marker assume, this number is important because it is possible to modify a Marker already published in ROS;
6. **frame**: Refers to the orientation of the maker, the same type of reference used to create a PointCloud.

Let us now create a function that transforms our XYZ data in Markers. Due to the large number of parameters used to create the Maker, our function worked with fixed values for color. The developed function is show in Fig. 30. This function creates a message of type *visualization_msgs/MarkerArray* and converts each point of the XYZ array in a Marker, and then adds the Marker into the created message. Finally, the message publishes the result. The input parameters are XYZ, topicName, type, scale, frame, numberPoints and time to refer you:

1. **xyz**: Refers to the coordinated input matrix containing type X, Y and Z;
2. **topicName**: Refers to the topic name published in ROS;
3. **type**: It refers to the type Marker the same as found in Table 4;
4. **scale**: Refers to Marker size;
5. **frame**: Refers to the reference point;
6. **numberPoints**: Refers to the number that will be converted from XYZ data. As the Marker occupies a size larger than a point of Point Cloud, it is not necessary to convert all elements to Marker.
7. **time**: Refers to the time the topic is visible on the ROS.

Go to the experiment, copy the contents of the function and paste into a text document in your workspace with *convertPCLtoMarkersROS* name, make sure it is set on your path in Matlab. First, we have to get a xyz matrix for this type:

```

1 rosshutdown;
2 rosinit;
3 topic = rossubscriber('Topic_Name_PointCloud2');
4 pccloud = receive(topic);
5 xyz = readXYZ(pccloud);
```

Now just call the function, enter the following code, changing the parameters if you want:

```

1 convertPCLtoMarkersROS(xyz, 'Markers', 1, 0.08, 'map',
1000, 10);
```

Figure 31 brings converted into a Point Cloud to Markers with different possible types. The number of points used in the tests was 1500 and size of Markers was 0.08.

```

1 %responsible function to create a Marker Array
2 function [Markers] = convertPCLtoMarkersROS(xyz,topicName, type, scale, frame
3 , numberPoints ,time)
4
5 %Create topic
6 pub = rospublisher(strcat(' / ',topicName),'visualization_msgs/MarkerArray');
7
8 %Creates MakeArray
9 markers = rosmessage('visualization_msgs/MarkerArray');
10
11 %Sets the loop jump
12 jump = round(size(xyz,1)/numberPoints);
13
14 %Loop responsible for creating the Markers
15 contMarker = 1;
16 for i=1:jump:size(xyz,1)
17     points(contMarker) = marker(xyz(i,1),xyz(i,2),xyz(i,3),1,1,1,type,scale,
18     contMarker,frame);
19     contMarker = contMarker+1;
20 end
21
22 %Pass the vector for Markers
23 markers.Markers = points;
24
25 %Sends ROS
26 send(pub,markers);
27
28 %Active time
29 pause(time);
30

```

Fig. 30 Function *convertPCLtoMarkersROS*

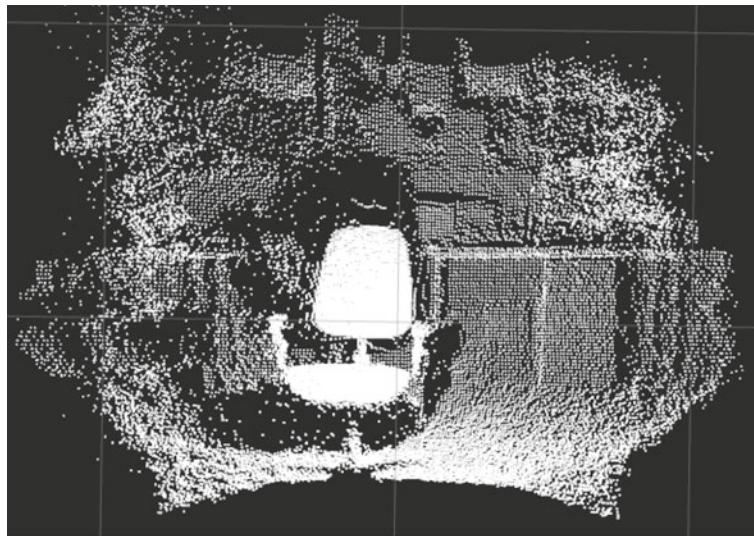
5.5 Filter XYZ Data

In some situations, we need parts of the Point Cloud data. This can happen when we want to remove a wall for example, or when we want to work with only the nearest points. In mobile robotics PCL can be used to obtain information about the distance of an object and for that, you need to filter the PCL to decrease the noise number, for example.

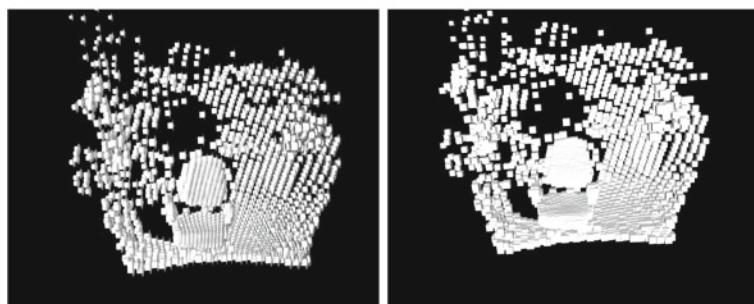
This subsection will bring developed a function in Matlab with of limiting the Point Cloud data in one of the axes XYZ, or even in all three axes simultaneously. Following the methodology already used in the chapter, it will first be presented to function, followed by explanations of their development and code examples and then actual use. The Rviz tool will be used to observe the results, for it will be used the previously created function *XYZ_to_sensor_msgs_PointCloud*.

To develop the filter the *find* [38] command will be used. The *find* command lets you search in an array indexes that satisfaction the condition desired. A filter sample in the X-axis of a matrix on XYZ format can be seen in Fig. 32. Line 16 takes the all indexes of matrix *xyz* that satisfy the condition. The condition is to have X between [-1: 1]. In this way, I cut my Point Cloud data in the X-axis leaving a total of two meters, one meter to the right and one meter left side of the center of the sensor. An example of this code can be seen in Fig. 33.

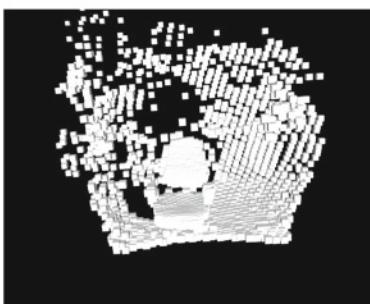
The same filter can be used in any of the axes. The Y-axis is responsible for height, having zero as the center respect to the sensor, the above of center it has a positive



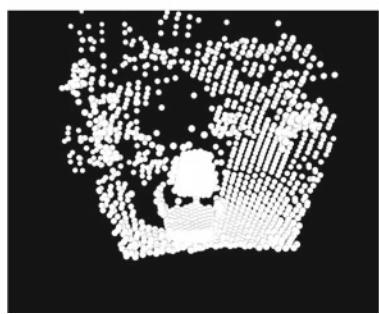
(a) PCL used in this experiment.



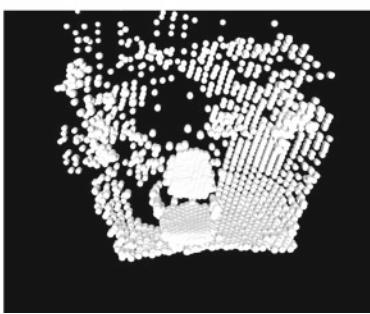
(b) Conversion of PCL in Markers of type Arrows.



(c) Conversion of PCL in Markers of type Cube.



(d) Conversion of PCL in Markers of type Sphere.



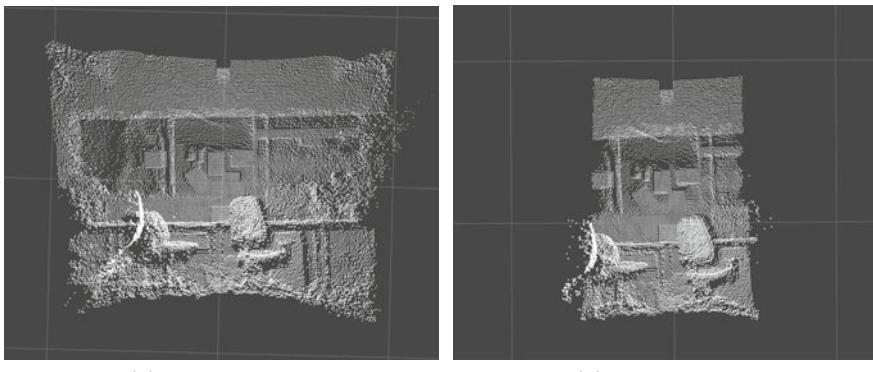
(e) Conversion of PCL in Markers of type Cylinder.

Fig. 31 Rviz window with sensor information SR4000

```

1 %ROS start
2 rosshutdown
3 rosinit('localhost')
4 %Get PCL
5 topic = rossubscriber('/swissranger/pointcloud2_raw');
6 pointcloud = receive(topic);
7 %Convert to XYZ matrix
8 xyz = readXYZ(pointcloud);
9 %Sets the filter size
10 xFilter = 1;
11 %Apply filter
12 index = find(xyz(:,1)>(xFilter*-1) & xyz(:,1)<xFilter);
13 %Create a new XYZ matrix
14 xyzFiltred = xyz(index, 1:3);
15 %Displays the result in Rviz
16 XYZ_to_sensor_msgs_PointCloud(xyzFiltred, 'xFilter', 'map', 10);

```

Fig. 32 XFilter

(a) Original PCL.

(b) Filtered PCL

Fig. 33 Filter on the X axis

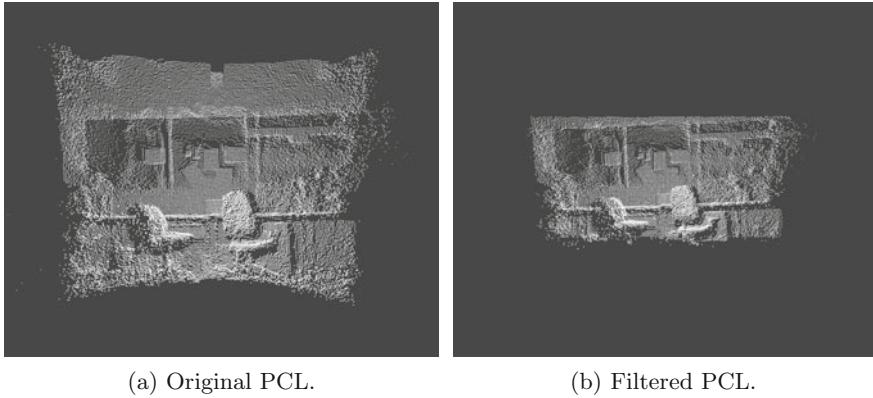
```

1 %ROS start
2 rosshutdown
3 rosinit('localhost')
4 %Get PCL
5 topic = rossubscriber('/swissranger/pointcloud2_raw');
6 pointcloud = receive(topic);
7 %Convert to XYZ matrix
8 xyz = readXYZ(pointcloud);
9 %Sets the filter size
10 yFilter = 1;
11 %Apply filter
12 index = find(xyz(:,2)>(yFilter*-1) & xyz(:,2)<yFilter);
13 %Create a new XYZ matrix
14 xyzFiltred = xyz(index, 1:3);
15 %Displays the result in Rviz
16 XYZ_to_sensor_msgs_PointCloud(xyzFiltred, 'yFilter', 'map', 10);

```

Fig. 34 YFilter

value and below the center is a negative value. The following code performs a filter on the Y-axis can be seen in Fig. 34 and its result can be seen in Fig. 35.



(a) Original PCL.

(b) Filtered PCL.

Fig. 35 Filter on the Y axis

```

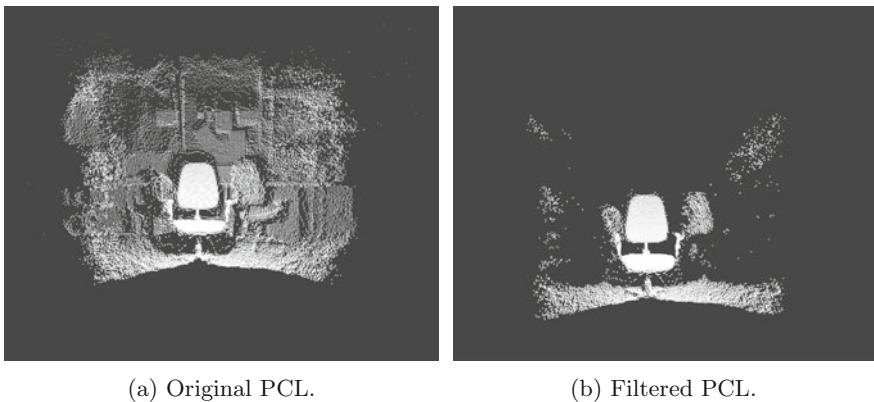
1 %ROS start
2 rosshowdown
3 rosinit('localhost')
4
5 %Get PCL
6 topic = rossubscriber('/swissranger/pointcloud2_raw');
7 pointcloud = receive(topic);
8
9 %Convert to XYZ matrix
10 xyz = readXYZ(pointcloud);
11
12 %Sets the filter size
13 zFilter = 2;
14
15 %Apply filter
16 index = find(xyz(:,3)<zFilter);
17
18 %Create a new XYZ matrix
19 xyzFiltred = xyz(index, 1:3);
20
21 %Displays the result in Rviz
22 XYZ_to_sensor_msgs_PointCloud(xyzFiltred, 'zFilter', 'map', 10);

```

Fig. 36 ZFilter

The Z-axis is responsible for the distance from the object to the sensor. The SR4000 sensor that is being used for the test has a maximum distance range of five meters. Limit the Z-axis is important because eliminate any unwanted noise. Most robots are designed to operate at a distance not far from your body, so you do not need to work with all the Point Cloud data in these cases. The code for limiting the Z axis can be seen in Fig. 36, the result of this code can be seen in Fig. 37.

It is also possible to develop a filter on all three axes. The following code shows an example of how to do in Fig. 38.



(a) Original PCL.

(b) Filtered PCL.

Fig. 37 Filter on the Z axis

```

1 %ROS start
2 rossshutdown
3 rosinit('localhost')
4
5 %Get PCL
6 topic = rossubscriber('/swissranger/pointcloud2_raw');
7 pointcloud = receive(topic);
8
9 %Convert to XYZ matrix
10 xyz = readXYZ(pointcloud);
11
12 %Sets the filter size
13 xFilter = 0.5;
14 yFilter = 1;
15 zFilter = 1.8;
16
17 %Apply filter
18 index = find(xyz(:,1)>(xFilter*-1) & xyz(:,1)<xFilter & xyz(:,2)>(yFilter*-1)
19 & xyz(:,2)<yFilter & xyz(:,3)<zFilter);
20
21 %Create a new XYZ matrix
22 xyzFiltered = xyz(index, 1:3);
23
24 %Displays the result in Rviz
XYZ_to_sensor_msgs_PointCloud(xyzFiltered, 'xyzFilter', 'map', 10);

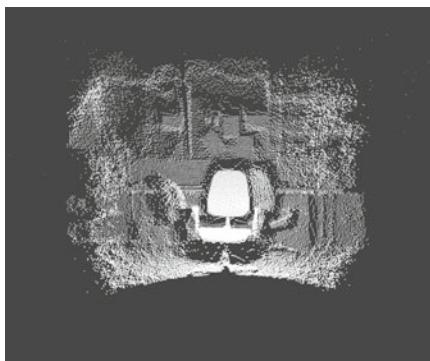
```

Fig. 38 XYZFilter

This code was defined that the new Point Cloud data told with a depth of 1.8 m, width of 0.5 m to the left and right of the center and the height of 1 m above and below the center. The result of this code can be seen in Fig. 39.

5.6 Transformation

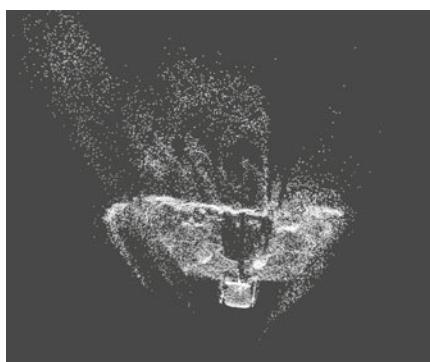
The sensor data need to be adjusted when used in robots. It is necessary that data be provided in relation to the center robot, and the robot turn to be converted in relation



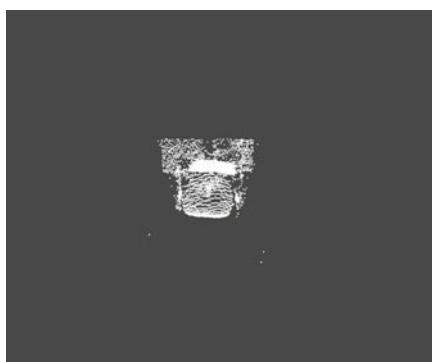
(a) Original PCD.



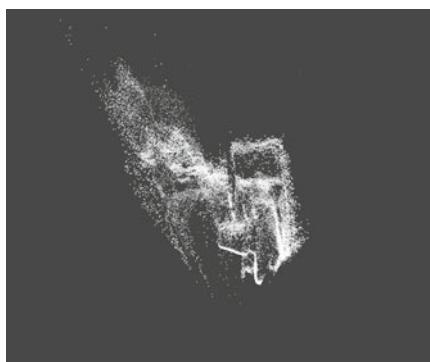
(b) Filtered PCD.



(c) Original PCD view from the top.



(d) Filtered PCD view from the top.



(e) Original PCD side view.



(f) Filtered PCD side view.

Fig. 39 Filter on the Z axis

```

1 %Receives tf
2 tftree = rostf;
3 pause(1);
4 %Creates a reference sr4000_link
5 tfSR4000 = rosmessage('geometry_msgs/TransformStamped');
6 tfSR4000.ChildFrameId = 'sr4000_link';
7 tfSR4000.Header.FrameId = 'base_link';
8 %Sets the distance between the robot and the sensor
9 tfSR4000.Transform.Translation.X = 0;
10 tfSR4000.Transform.Translation.Y = 1.5;
11 tfSR4000.Transform.Translation.Z = 0;
12 %Send the tf for ROS
13 tfSR4000.Header.Stamp = rostime('now');
14 sendTransform(tftree, tfSR4000)
15 pause(1);
16 %Creates a reference kinect_link
17 tfKinect = rosmessage('geometry_msgs/TransformStamped');
18 tfKinect.ChildFrameId = 'kinect_link';
19 tfKinect.Header.FrameId = 'base_link';
20 %Sets the distance between the robot and the sensor
21 tfKinect.Transform.Translation.X = 0;
22 tfKinect.Transform.Translation.Y = -1.5;
23 tfKinect.Transform.Translation.Z = 0;
24 %Send the tf for ROS
25 tfKinect.Header.Stamp = rostime('now');
26 sendTransform(tftree, tfKinect)
27 pause(1);

```

Fig. 40 *createTF*

to its position on the map. To make these adjustments, the ROS has a set of tools called *tf*. These settings are nothing more than geometric calculations, converting the position and the sensor rotation relative to the robot.

In this subsection, we will initialize the two cameras, Kinect and SR4000, and add them to the same robot with a stipulated distance between them. First we need to define all the transformations, in the center of the robot will be the name *base_link*, the sensor Kinect the name *Kinect_link* and the SR4000 sensor the name *sr4000_link*. The code to create these transformations can be seen in Fig. 40.

The distance from the sensor SR4000 this sitting as +1.5 m from the center of the robot on the Y axis, i.e., above the center, while the Kinect sensor is sitting as -1.5 m from the center of the robot on the Y axis, i.e. below the center. You can view the changes by *rqt_tf_tree* tool for that type in your terminal the command:

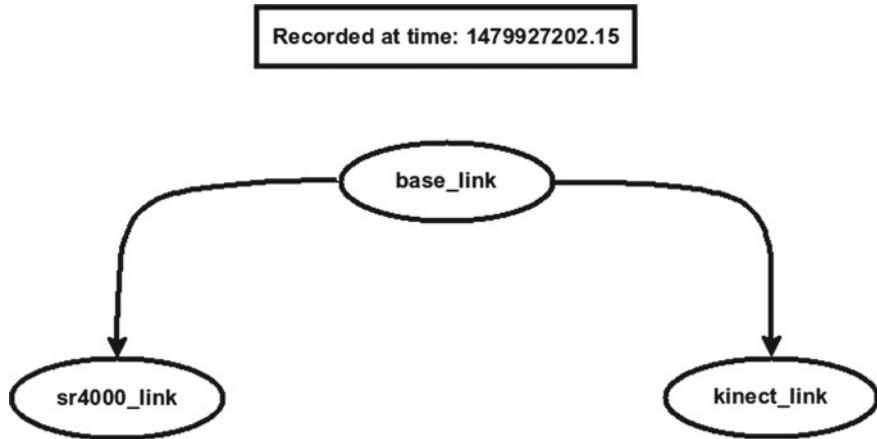
```
rosrun rqt_tf_tree rqt_tf_tree
```

A window as shown in Fig. 41 can be observed. Note that the two sensors are connected to *base_link*, which is the center of the robot.

The next step is the start of the two sensors, for these start two launchers created by typing:

```
1 roslaunch ~/catkin_ws/src/kinect.launch
2 roslaunch ~/catkin_ws/src/sr4000.launch
```

To apply the transformation, we must first change the reference sensor for reference to create, and then apply the transformation. This can be seen in Fig. 42. The result of code can be observed in Fig. 43.

**Fig. 41** Rqt tf tree

```

1  %ROS start
2  rosshutdown
3  rosinit('localhost')
4  %Creates the tf
5  creatTF ;
6  pause(1);
7  % Read the topics
8  kinect = rossubscriber('/camera/depth/points');
9  sr4000 = rossubscriber('/swissranger/pointcloud2_raw');
10 %Create new topics
11 pub = rospublisher('Kinect','sensor_msgs/PointCloud2');
12 pub2 = rospublisher('sr4000','sensor_msgs/PointCloud2');
13
14 while true
15     %takes the Point Cloud and modify your reference
16     pointcloudKinect = receive(kinect);
17     pointcloudKinect.Header.FrameId = 'kinect_link';
18     % Applies tf and publishes
19     pointcloudKinect2 = transform(tftree, 'base_link', pointcloudKinect);
20     send(pub,pointcloudKinect2);
21     %takes the Point Cloud and modify your reference
22     pointcloudSR4000 = receive(sr4000);
23     pointcloudSR4000.Header.FrameId = 'sr4000_link';
24     % Applies tf and publishes
25     pointcloudSR40002 = transform(tftree, 'base_link', pointcloudSR4000);
26     send(pub2,pointcloudSR40002);
27     pause(0.5);
28 end

```

Fig. 42 Apply TF

6 Conclusion

The Point Cloud can be used for various tasks in mobile robotics, for example for mapping, for SLAM and even with the location reference. The Point Cloud can also be used to identify the distance of objects or object recognition. However, for advanced work with the Point Cloud, you must first be able to use it in simple jobs. This tutorial brought a set of techniques and tools that allow the reader to start its activities with

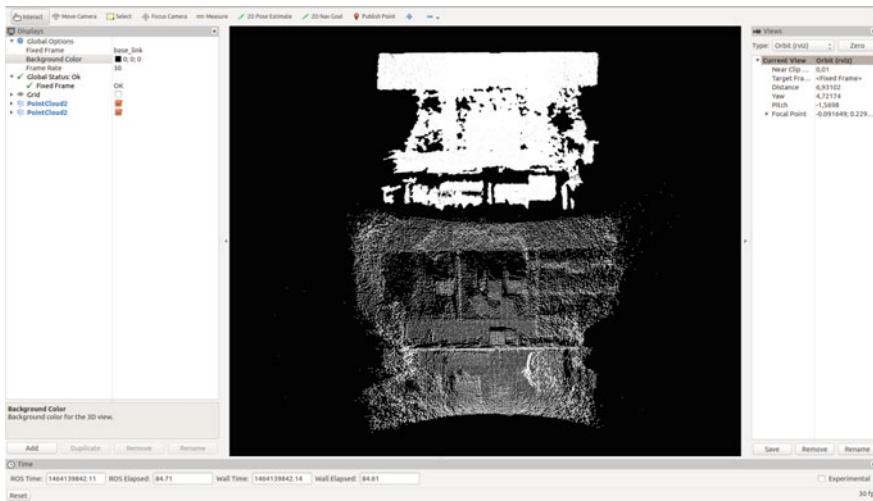


Fig. 43 Transformation applied to Point Cloud data seen in RVIZ

the Point Cloud using Matlab. At the end, the reader will know everything you need about the Point Cloud.

References

1. Oliver, A., S. Kang, B.C. Wünsche, and B. MacDonald. 2012. Using the kinect as a navigation sensor for mobile robotics. In *Proceedings of the 27th conference on image and vision computing New Zealand*, CM, 509–514.
2. Endres, F., J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard. 2012. An evaluation of the RGB-d slam system. In *2012 IEEE international conference on robotics and automation (ICRA)*, 1691–1696. New York: IEEE.
3. Whelan, T., M. Kaess, M. Fallon, H. Johannsson, J. Leonard, and J. McDonald. 2012. Kintinuous: Spatially extended kinectfusion.
4. Whelan, T., M. Kaess, J.J. Leonard, and J. McDonald. 2013. Deformation-based loop closure for large scale dense RGB-D slam. In *2013 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, 548–555. New York: IEEE.
5. Wallace, L., A. Lucieer, Z. Malenovský, D. Turner, and P. Vopěnka. 2016. Assessment of forest structure using two UAV techniques: A comparison of airborne laser scanning and structure from motion (SfM) point clouds. *Forests* 7 (3): 62.
6. Wang, Q., L. Wu, Z. Xu, H. Tang, R. Wang, and F. Li. 2014. A progressive morphological filter for point cloud extracted from UAV images. In *IEEE international geoscience and remote sensing symposium (IGARSS), 2014*, 2023–2026. New York: IEEE.
7. Nagai, M., T. Chen, R. Shibasaki, H. Kumagai, and A. Ahmed. 2009. UAV-borne 3-d mapping system by multisensor integration. *IEEE Transactions on Geoscience and Remote Sensing* 47 (3): 701–708.
8. Tao, W., Y. Lei, and P. Mooney. 2011. Dense point cloud extraction from UAV captured images in forest area. In *2011 IEEE international conference on spatial data mining and geographical knowledge services (ICSDM)*, 389–392. New York: IEEE.

9. Matlab, “Matlab”. 2016. <http://www.mathworks.com/products/matlab/>.
10. GitHub. 2016. Installation from GitHub on debian. <https://github.com/Singular/Sources/wiki/Installation-from-GitHub-on-Debian>.
11. PCL. 2016. What is PCL? Abr. <http://pointclouds.org/about/>.
12. Li, L. 2014. Time-of-flight camera—an introduction, *Technical White Paper*.
13. PCL. 2016. Module io, Abr. http://docs.pointclouds.org/trunk/group__io.html.
14. ROS. 2016. PCL overview, Abr. <http://wiki.ros.org/pcl/Overview>.
15. Kreylos. 2016. Kinect hacking. <http://idav.ucdavis.edu/~okreylos/ResDev/Kinect/>.
16. HEPTAGON. 2016. Our history. <http://hptg.com/about-us/#history>.
17. ROS. 2016. About ROS, Abr. <http://www.ros.org/about-ros/>.
18. ROS. 2016. ROS jade installation instructions, Abr. <http://wiki.ros.org/ROS/Installation>.
19. ubuntu. 2016. About ubuntu, Abr. <http://www.ubuntu.com/about/about-ubuntu>.
20. Ubuntu. 2016. Ubuntu 14.04.4 LTS (trusty Tahr), Abr. <http://releases.ubuntu.com/14.04/>.
21. Ubuntu. 2016. Install Ubuntu 16.04 LTS, Abr. <http://www.ubuntu.com/download/desktop/install-ubuntu-desktop>.
22. ROS. 2016. Ubuntu install of ROS indigo, Abr. <http://wiki.ros.org/indigo/Installation/Ubuntu>.
23. ROS. 2016. freenect_camera. http://wiki.ros.org/freenect_camera.
24. ROS. 2016. freenect_launch. http://wiki.ros.org/freenect_launch.
25. UBUNTU. 2016. Package: libfreenect-dev (1:0.0.1+20101211+2-3). <http://packages.ubuntu.com/precise/libfreenect-dev>.
26. ROS. 2016. openni_launch. http://wiki.ros.org/openni_launch.
27. Robot, B.R. 2016. Kinect basics. http://sdk.rethinkrobotics.com/wiki/Kinect_basics.
28. ROS. 2016. Kinect: Using microsoft kinect on the evarobot. <http://wiki.ros.org/Robots/evarobot/Tutorials/indigo/Kinect>
29. HEPTAGON. 2016. Swissranger. <http://hptg.com/industrial/>.
30. ROS. 2016. swissranger_camera. http://wiki.ros.org/swissranger_camera.
31. ROS. 2016. cob_camera_sensors. http://wiki.ros.org/cob_camera_sensors.
32. ROS. 2016. Care-o-bot: Configuring and using the swissranger 3000 or 4000 depth sensor. http://wiki.ros.org/cob_camera_sensors/Mesa_Swissranger.
33. M. imaging. 2016. Sr4000/sr4500 user manual. http://www.realtechsupport.org/UB/SR/range_finding/SR4000_SR4500_Manual.pdf.
34. ROS. 2016. visualization_msgs(marker message. http://docs.ros.org/api/visualization_msgs/html/msg/Marker.html.
35. ROS. 2016. visualization_msgs(markerarray message. http://docs.ros.org/api/visualization_msgs/html/msg/MarkerArray.html.
36. ROS. 2016. The marker message. <http://wiki.ros.org/rviz/DisplayTypes/Marker>.
37. OctoMap. 2016. An efficient probabilistic 3d mapping framework based on octrees. <http://octomap.github.io/>.
38. MathWorks. 2016. find. <http://www.mathworks.com/help/matlab/ref/find.html>.

Part VI

ROS Simulation Frameworks

Environment for the Dynamic Simulation of ROS-Based UAVs

Alvaro Rogério Cantieri, André Schneider de Oliveira, Marco Aurélio Wehrmeister, João Alberto Fabro and Marlon de Oliveira Vaz

Abstract The aim of this chapter is to explain how to use the Virtual Robot Experimentation Platform (V-REP) simulation software with the Robot Operating System (ROS) to create and collect signals and control a generic multirotor unmanned aerial vehicle (UAV) in a simulation scene. This tutorial explains all the steps needed to select an UAV model, assemble and configure the propellers, configure the dynamic parameters, add sensors, and finally simulate the scene. The final part of the chapter presents an example of how to use MATLAB to create control scripts using ROS and also collect data from sensors such as accelerometers, gyroscopes, GPS, and laser scanners.

Keywords Multirotor simulation · Multirotor ROS · Hexacopter PID · V-REP hexacopter

1 Introduction

Unmanned aerial vehicles (UAVs) are currently one of most interesting areas of robotics, with many studies currently taking place in the scientific community. This kind of research is popular because the vehicles and control electronics are becoming increasingly smaller and cheaper. One common difficulty of working with UAVs is the fragility of this type of aircraft. If a control strategy works badly, an accident can occur and damage the vehicle. Another difficulty when beginning real tests with a UAV is the need for a large controlled test area, which is not commonly accessible for research labs and education centers. Virtual environments are a powerful tool for UAV simulation and enable the careful assessment of new algorithms without the

A.R. Cantieri (✉) · M. de Oliveira Vaz
Federal Institute of Paraná, Rua João Negrao, 1285, Curitiba, Brazil
email: alvaro.cantieri@ifpr.edu.br
URL: <http://www.utfpr.edu.br/>

A.S. de Oliveira · M.A. Wehrmeister · J.A. Fabro
Federal University of Technology - Paraná, Av. Sete de Setembro, 3165, Curitiba, Brazil



Fig. 1 Main UAV topologies

risk of damage. This kind of software minimizes the time spent on tests and the risks associated with them, making development more efficient.

This chapter presents the development of a virtual environment for dynamic UAV simulation using the Virtual Robot Experimentation Platform (V-REP) simulation software and Robot Operating System (ROS). This chapter is divided in five sections, beginning with a basic review of multirotor flight and finishing with a summary of all the steps needed to assemble and control a virtual multirotor aircraft.

Section 2 presents some basic background regarding multirotor aircraft movement and flight, explaining the concepts of roll, pitch, yaw, and throttle. It also describes how it is possible to create movement in the desired direction by changing the propeller rotations.

Section 3 presents the principal topologies of UAVs, focusing on multirotor or multicopter aircraft (Fig. 1), and explaining their particular characteristics, assembly, and propulsion forces. The objective is to present the basic concepts behind this kind of aircraft and their flight, making it easier to understand the necessary actions for motion control and stabilization.

Section 4 introduces V-REP and shows how to create an environment scene and a virtual UAV model fully compatible with ROS on V-REP, as shown in Fig. 2. The step-by-step construction of an example model is shown, which includes the assembly of the UAV frame, propeller and motor system, and sensors. Some important considerations about the simulation software are discussed, showing the details of the model manipulation and its configuration parameters.



Fig. 2 Virtual hexacopter style UAV

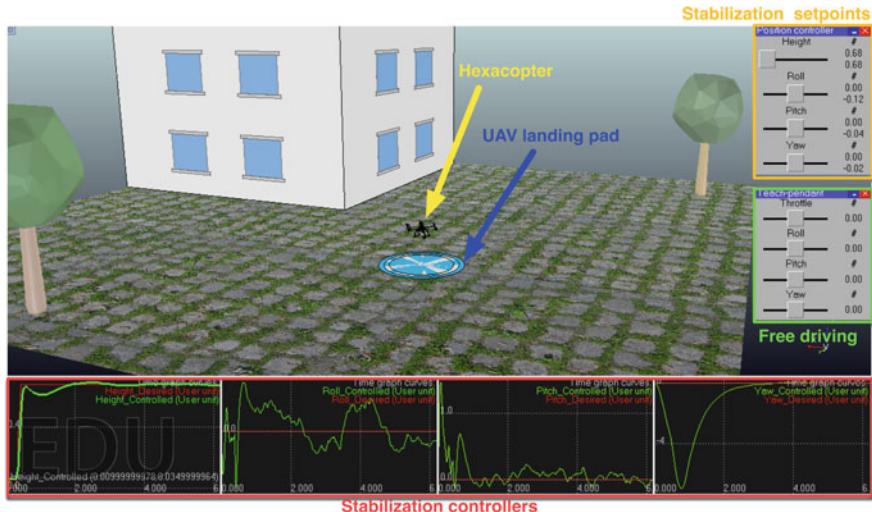


Fig. 3 Virtual environment to dynamic simulation of UAVs

The final section focuses on the interconnection between the virtual UAV and ROS, with careful discussion about the design of the ROS nodes (publishers and subscribers). The main objective of this section is to create nodes that are nearly identical to the ones for real equipment with respect to the message type and its update frequency. The simulation is based on a distributed PID stabilization controller running on MATLAB scripts through ROS nodes. The control signals are presented in V-REP's simulation time graphs to express the UAV behavior during flight, as shown in Fig. 3.

2 Basic Multirotor Flight Concepts

Modeling the flight of a multirotor aircraft is a complex problem. It depends on the kinematic and dynamic characteristics of the aircraft, which change for every new model or configuration. In this tutorial chapter, the three-dimensional motion of the aircraft is considered only in terms of the basic moves, roll, pitch, yaw, and throttle, created by the difference in rotation of the propellers. The study of the dynamic characteristics of the aircraft and how to create these characteristics in the model are beyond the scope of this chapter.

To understand multirotor control and flight, we need first to understand the different moves of a multirotor in space. The three angles of rotation, roll, pitch, and yaw, are used to describe the aircraft's orientation in space. Figure 4 shows the orientation of these axes relative to the center of the multirotor body.

The multirotor body rotates around these axes to move horizontally in space. For instance, to make the aircraft move forward, we rotate the frame around the pitch axis, elevating its back, which creates a force that pushes the aircraft forward. If the multirotor body rotates around the roll axis, the aircraft will move sideways. Rotation around the yaw axis reorients the front of the aircraft in another direction.

The vertical moves of the aircraft are an effect of an imbalance between gravity and thrust. When these forces are equal, the aircraft stays static at the same altitude. To make the multirotor go up, it is necessary to increase the velocity of all the propellers

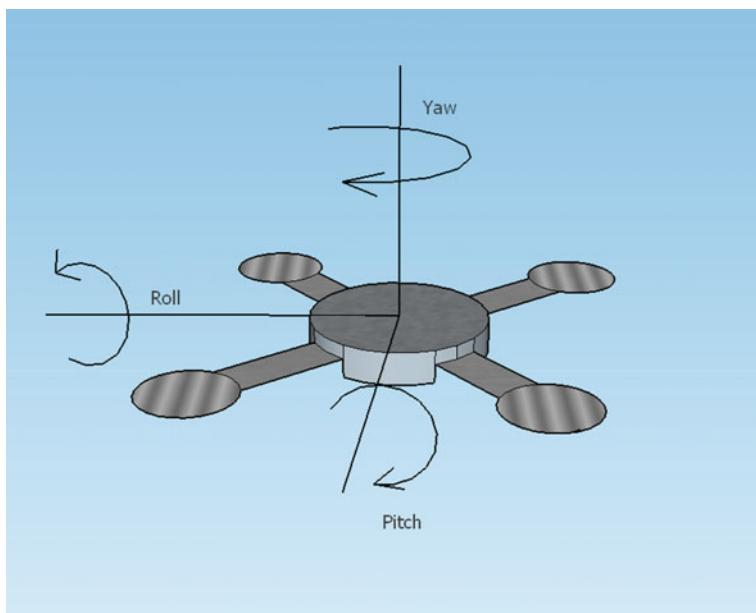


Fig. 4 Roll, Pitch and Yaw

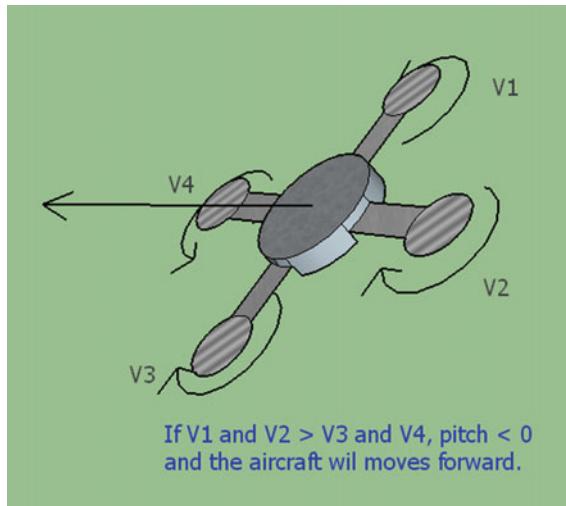


Fig. 5 Pitch rotation example

by the same amount. This increases the thrust and makes the aircraft gain altitude. In contrast, to make the multirotor dive, we decrease the velocity of all the propellers, making the thrust force smaller and allowing altitude to be lost.

All the spatial moves of a multirotor aircraft are achieved by changing the propeller rotation velocities. If the number of propellers of the multirotor configuration is even, as on quadcopters, hexacopters, and octocopters, the moves are created by increasing the velocity of the motors on one side and decreasing the velocity of those on the other side. In practical situations, changes must be small to avoid the loss of flight control and stability. Figure 5 shows an example pitch rotation and its associated move.

Yaw rotation occurs in a somewhat different way. This kind of rotation results from the rotational torque created by the propellers of the aircraft. To prevent this situation, multirotor aircraft are assembled such that the rotation of each rotor on the frame compensates for another rotor that is located on the other side of the frame and rotates in the opposite direction. This neutralizes the rotational torques and keeps the yaw rotation at zero. Figure 6 shows the propellers with the opposite rotation directions to balance the rotational forces.

When we want to re-orientate the front of the aircraft in another direction, we must create yaw rotation by increasing or decreasing the rotation of a specific group of propellers. Figure 7 shows this effect.

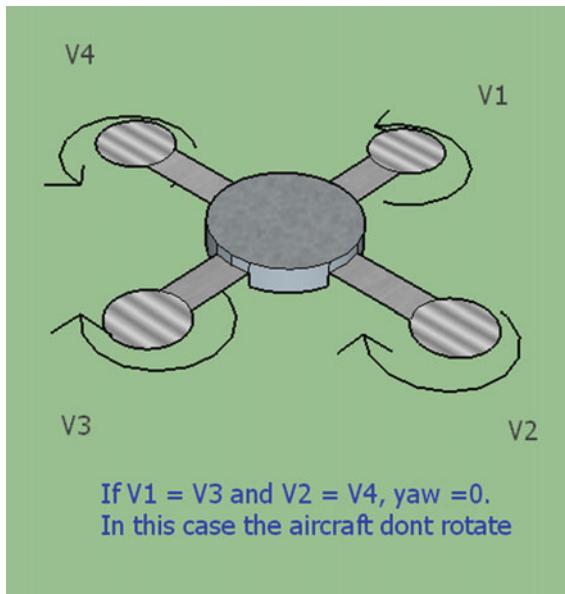


Fig. 6 Rotational torques of created by the helices

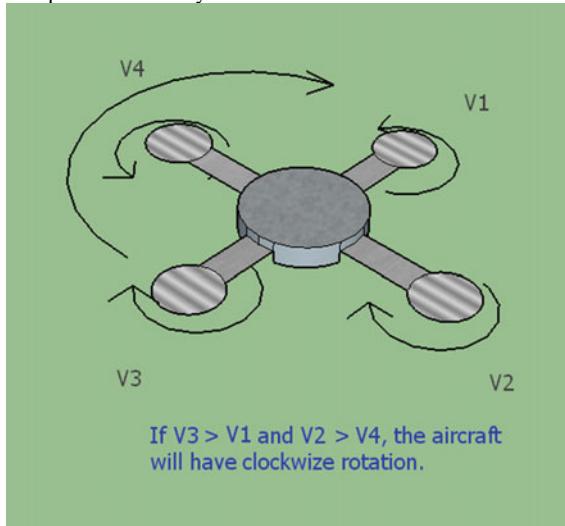


Fig. 7 Unbalanced propellers velocity and yaw rotation of aircraft

3 Multirotor Configurations and Specific Characteristics

Several models of multirotors are available today, with many frame configurations and specific characteristics. The most common models are quadcopters, hexacopters, octocopters, tricopters, and V-tails. The several kinds of configurations, construction



Fig. 8 Tricopter multirotor example

materials, motors, helix and other components create a different flight performance for each aircraft. The thrust force, payload, velocities, and other characteristics will change for each aircraft, but in general, the fundamental characteristics like total payload and external perturbation reaction capacity can be compared for different basic multirotor configurations. This section briefly describes the most common configurations and their fundamental characteristics.

The tricopter configuration is one of the cheaper assemblies and was popular for some time. This configuration is shown in Fig. 8. It has an odd number of propellers, demanding additional care to control yaw rotation. In all other cases, there are two opposite rotating propellers canceling the rotational torque one of each other.

The two front propellers rotate in opposite directions, mutually canceling the rotational torque. The back propeller does not have torque cancellation, and it is hence necessary to create a counter-torque on the frame by rotating the propeller by an appropriate angle. This is achieved by mounting this propeller on a horizontal fixed servo-motor that turns the propeller to the necessary inclination. This kind of problem can be fixed by adding a second back propeller on the frame on the same axis, but directed towards the ground and rotating in the opposite direction. This arrangement is called a Y-4 multirotor, and decreases the mechanical complexity of the arrangement.

The currently most popular arrangement of propellers is the quadcopter. This configuration provides good thrust force and stability, and its components are relatively low cost. The quadcopter can be assembled in two different ways, in an “x” or “+” arrangement. Figure 9 shows these two configurations.

Controlling the movement of a quadcopter is relatively simple because of its motor symmetry. The energy consumed by four propellers is less than that consumed by the hexacopter and octocopter configurations, but the thrust force is smaller for most cases. Another disadvantage is the fact that if one propeller fails, the electronic flight controller cannot stabilize the aircraft, resulting in a crash.

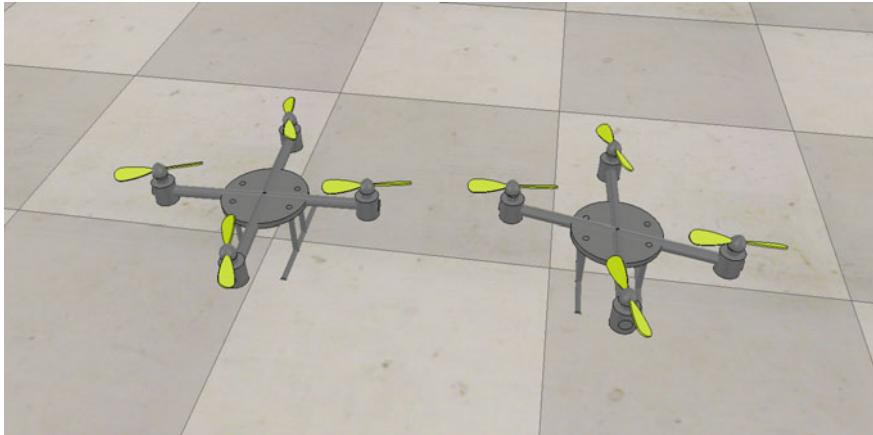


Fig. 9 Quadricopter frames X and +

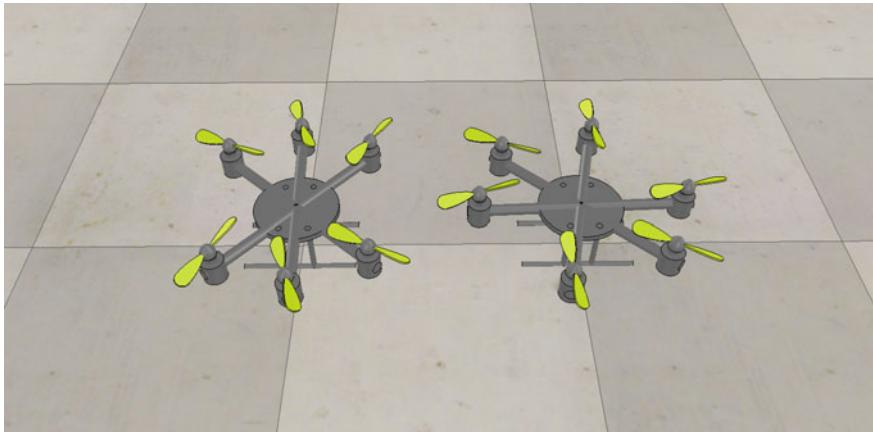


Fig. 10 Hexacopter frames X and +

The hexacopter is a good option for achieving higher thrust values and flight stability. Like the quadcopter, a hexacopter can be assembled using x or + configurations, as shown in Fig. 10. The presence of six propellers assures better flight security, because if one propeller fails, the others are able to maintain the thrust force and balance necessary to avoid a loss of control. Although the power consumption is greater than a quadcopter, the use of larger batteries can assure equivalent flight time, as the thrust force of the six propellers can carry more weight. This weight capacity also allows the use of a heavier sensor payload, which makes it a good choice for scientific and research applications.

The octocopter is very similar to the hexacopter, but the eight motors provide more thrust force, and a better immunity to external perturbation. It can be assembled on

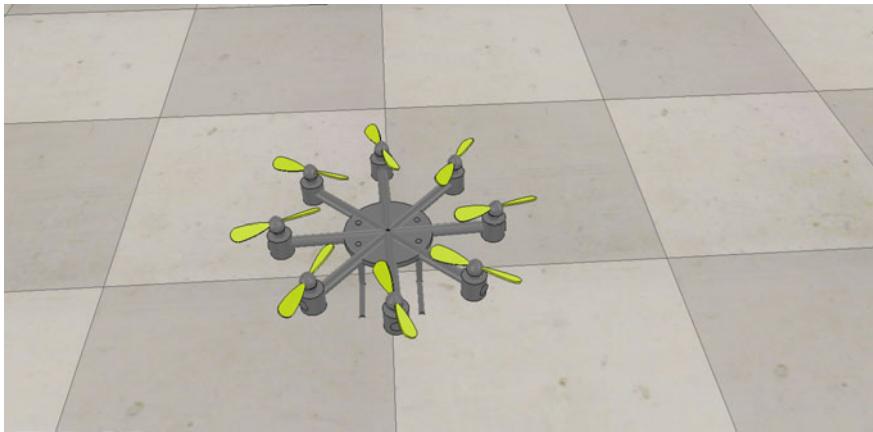


Fig. 11 Octocopter frame

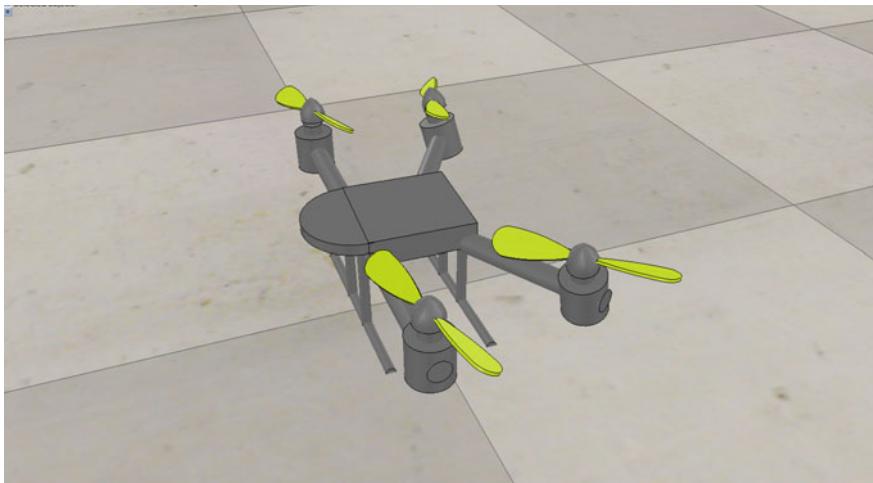


Fig. 12 V-tail multirotor

x and + configurations, like the hexacopter and quadcopter. The eight propellers consume a larger amount of energy than the other multirotor configurations, but this disadvantage is compensated for by its flight stability, which makes it a good choice for outdoor applications. Figure 11 shows common octocopter configurations.

A special case of a quadcopter is the V-tail configuration. In this kind of multirotor, the two back propellers are inclined outside of the frame. This inclination offers some additional acrobatic performance, but leads to lower power efficiency and air blow interference between the two back propellers. Figure 12 shows this model.

For all the above configurations except the V-tail, it is possible, for each upward-facing propeller, to add another propeller pointing downward, resulting in a dual-

propeller multirotor configuration. These configurations enable more stability and thrust force, and have the additional advantage of helping the associated propellers cancel the rotational torque. The problem with this kind of configuration is the higher cost of components and power consumption, making them less common.

4 Multirotor Model Creation and Scene Composition in V-REP

V-REP is an abbreviation of the Virtual Robot Experimentation Platform, a software platform created for professional robot systems development. The V-REP simulator offers good flexibility, strong simulation tools, and a wide number of robot and component models, which makes it an optimal choice for developing applications in all robotic areas and a good alternative to other simulation software like the GAZEBO simulator.

V-REP is available for Linux, Windows, and macOS operational systems. The educational version is free to use, and can be download from the Coppelia Robotics website (www.coppeliarobotics.com). For Linux users, is sufficient to download the software and run it on a terminal using the “sh vrep.sh” command. When running, the VREP interface will appear like the example in Fig. 13. This figure shows the components of the interface such as the simulation window, menus, and other tools used to created and run the simulation.

An interesting feature in this figure is the presence of some commercial robot models in the model browser window. These models are created and offered in the

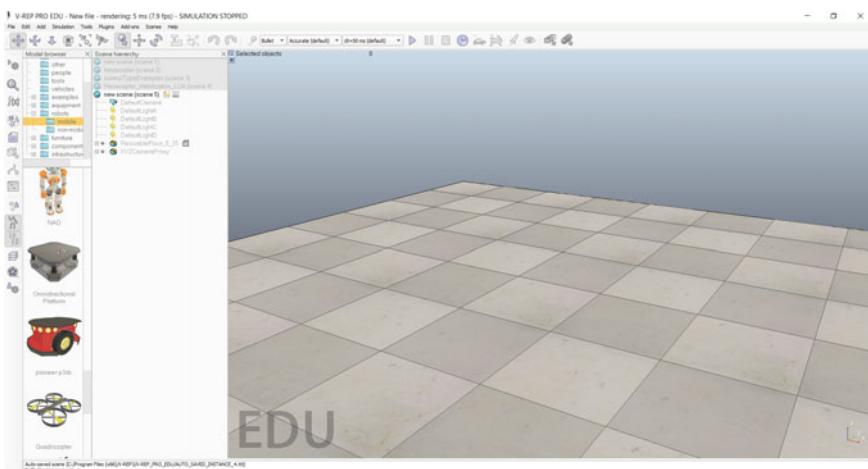


Fig. 13 V-REP interface

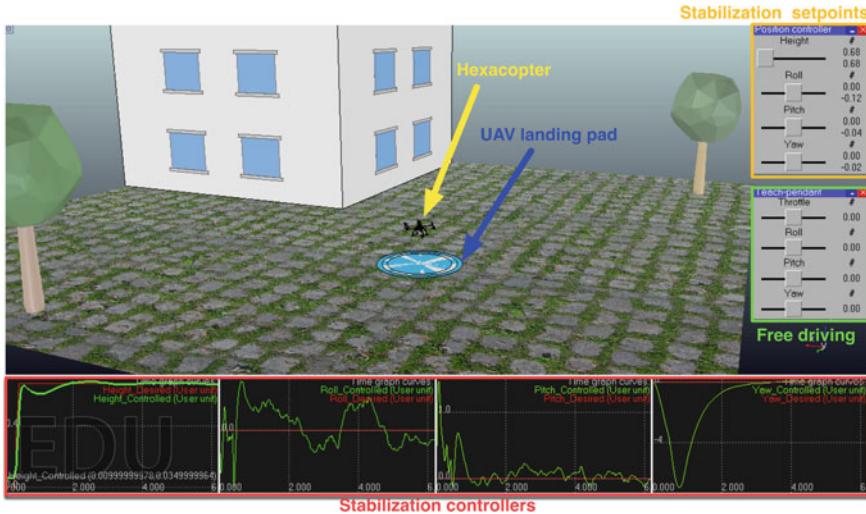


Fig. 14 One complex scene

software by users or partners and provide a great opportunity to work with some expensive robots without the necessity of purchasing a real one.

A good knowledge of all the tools, components, and resources available in the software is important before beginning to work with V-REP. Reading the VREP User Manual and studying the basic tutorials available on software's website are the best ways to begin working with the software. For a better understanding of the actions and steps shown in this tutorial, we strongly recommend this study before starting the experiments [1].

To begin the hexacopter simulation, first the scene must be created and the desired components must be included in it. A scene is the virtual environment of a simulation on V-REP, which contains all the simulation elements, objects, and scripts that make the simulation work. A default scene contains cameras and illumination objects as well as the main script for the simulation. The main script is responsible for running all the associated scripts for the simulation, and it is not supposed to be modified. Environment objects can be added to the scene as static or dynamic objects. A collection of object models is available in V-REP's model menu. To add an object in the scene, we can drag and drop the object model to locate it at the desired position. Figure 14 shows a complex environment created in a scene with common V-REP models.

The behavior of the model is described by its model script and properties. The properties of one model are configurable by accessing the "Object Properties Toolbox." There are several configuration parameters available for each model, and we strongly recommend reading the V-REP User Manual to understand them better. The most relevant parameters for the example in this chapter are as follows:

- Collidable: Allows software collision detection for the selected object.

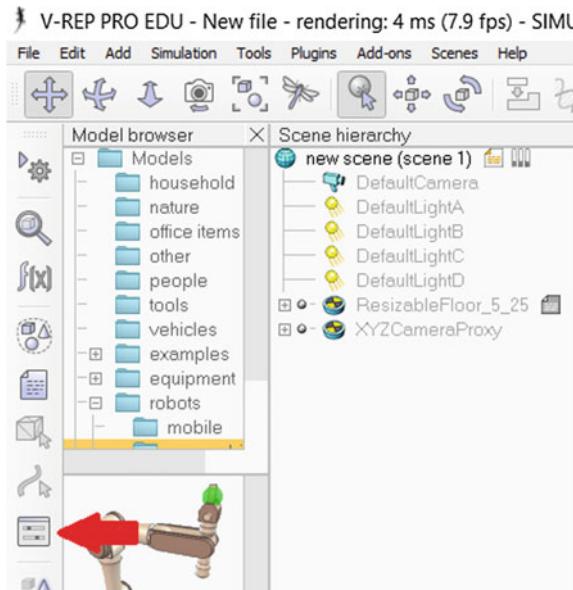


Fig. 15 Toggle custom user interface button

- Measurable: Allows software minimum distance calculation for the selected measurable object.
- Detectable: Allows proximity sensor detection for the selected detectable object.
- Renderable: Allows vision sensor detection for the selected renderable object.

Let us now create the scene with the necessary components to perform the example simulation. To create it, the following tools are necessary:

- Four slider buttons to control the horizontal and vertical position of the hexacopter.
- Four graphs components to record the position data of the hexacopter.
- Four floating view components to show the data recorded by the graphs.
- The hexacopter model.

To create the slider button, click on the “Toggle custom user interface edit module” button on the toolbar. This button is shown in Fig. 15. The pop-up menu started by this button is shown in Fig. 16.

We now create the buttons by clicking on the “Add new user interface,” setting the cells to 10×4 . This action will create a one-button model, as shown in Fig. 17.

To create a slider on the button base, click on the cells of one line holding the Ctrl key to select them. Next, click on “Insert merged button” and select type “Slider.” On the “Button label” box, enter the button label, which is “Throttle” for the first button. Repeat these operations for the other three buttons, labeling them “Roll,” “Pitch,” and “Yaw.” On the left side of the screen, the “Custom User Interface” window shows the name of this button, in this case “UI.” This name can be changed by clicking on the text, but for the example simulation, it must be retained, because the hexacopter

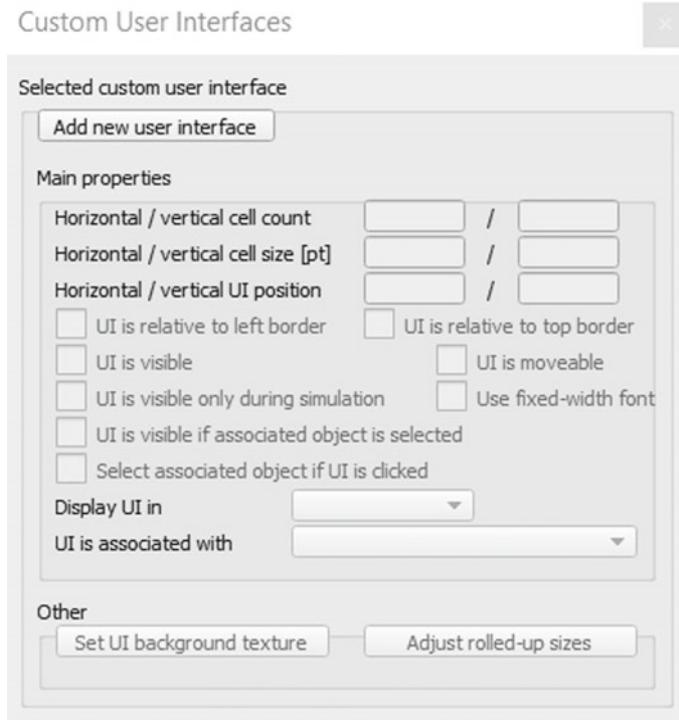


Fig. 16 Pop up menu

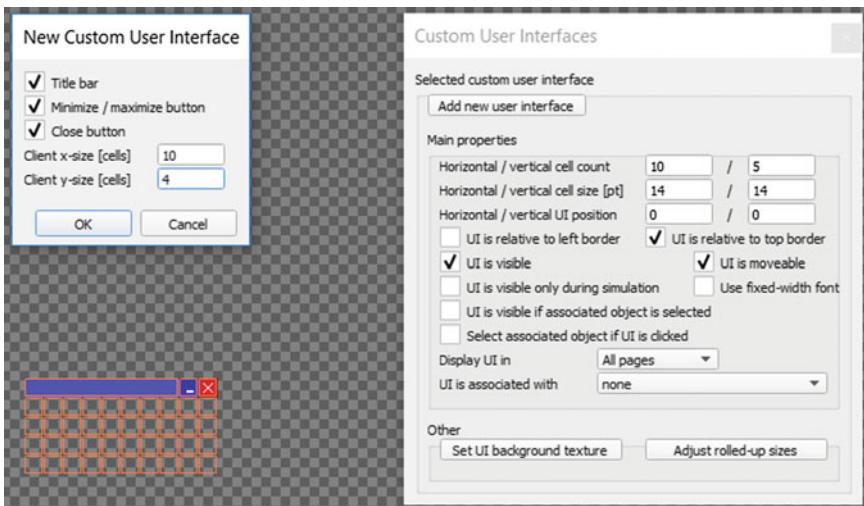


Fig. 17 Slider button model

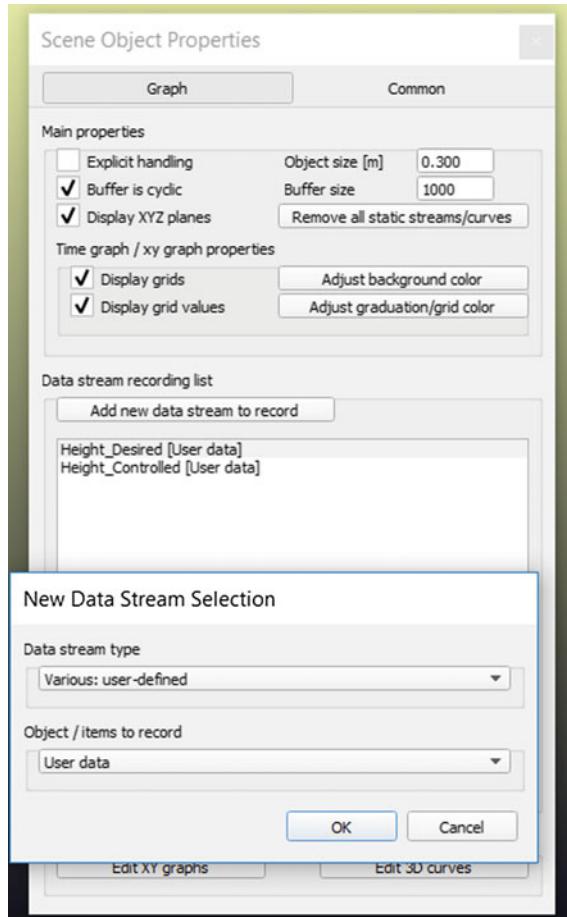


Fig. 18 Scene Object Properties window

script uses this name to get the values from these buttons. In this example, another slider-button is needed to control the moves of the hexacopter. Repeat the steps above, labeling these buttons “Height,” “Roll,” “Pitch,” and “Yaw.” The name of this button is “UI0.” Now we create the graphs to record the hexacopter position data. This can be done using

add-> graph

The graph will appear in the “Scene Hierarchy” window. Rename it to “Graph Height.” To associate one data stream with this graph, click on the graph icon in the “Scene Hierarchy” window. A “Scene Object Properties” box like the one shown in Fig. 18 will appear. Click on “Add new data stream to record,” select the item “Various: user defined” on the “Data Stream Type” button and select “User Data”

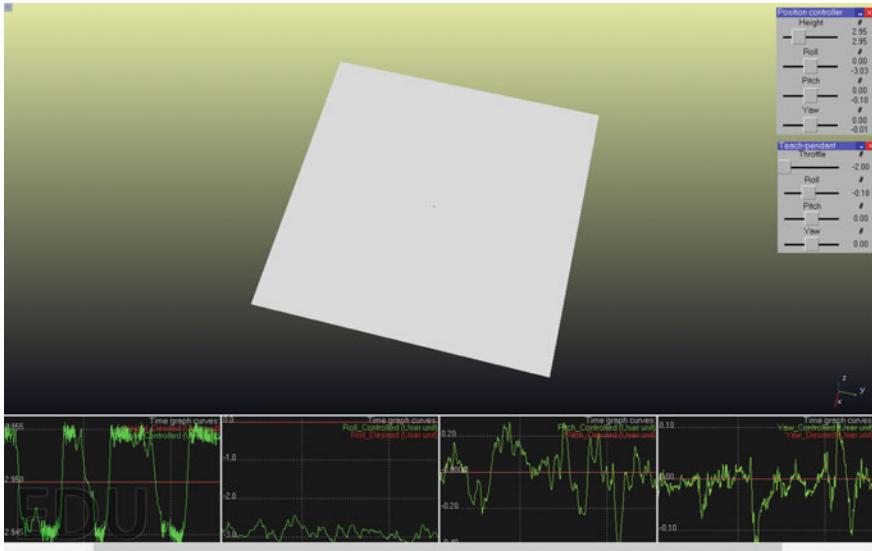


Fig. 19 Completed scene

on the “Object/Items to record” button. On the “Data stream record list” window, rename the label “Data [User data]” to “Height _Desired [User data].” To finish, add a new data record on the same graph and rename it “Height _Controlled [User data].”

Now it is necessary to associate the graph with a “floating view” window, where the data will be drawn. Right click on the scene and click on

add-> floating view

A empty floating window will appear on the screen. To associate the graph with this window, click on the window and click on

view-> associate view with selected graph

The graph must be selected in the scene to perform this association. This step completes the creation of the height graph display window used in the example. To complete the scene, repeat all the steps to create the other three graph windows for showing the roll, pitch, and yaw data. After these steps, the basic scene is ready. Figure 19 shows the result of these actions.

Once the scene is ready, the virtual multirotor model may be created. All multirotor models can be simulated in V-REP in a similar way if the motion characteristics of each kind of aircraft are correctly considered.

The most common way to create this kind of model is by drawing it directly in the simulator using primitive forms. We can also import a previously designed CAD model, as done in this example. Some websites offer this kind of model for download, like the Grabcad Community website [2], so it is easy to find some common

configuration models for use in this kind of simulation. The other alternative is to draw a specific model in CAD software and import it into V-REP.

The models drawn in V-REP provide better computational performance during the simulation because of their low complexity. Nevertheless, the creation of a detailed model can be laborious, and the task of importing a prepared CAD frame is attractive. Currently, the CAD data formats supported by V-REP are OBJ, STL, DXF, 3DS, and Collada Windows. This importation is made using

edit → file → import → mesh

The frame models shown in Figs. 8, 9, 10, 11, and 12 are available in <https://sourceforge.net/projects/rosbook-2016-chapter-4/files/>. All these models were created in V-REP using simple form object tools (except the helix), and can be easily extended to more complex models. For the initial simulation tests, they offer good performance and are easy to use.

For the tutorial example, a CAD model of a hexacopter + configuration was downloaded and imported in V-REP. This was done to keep the tutorial's presentation and use of the tools in V-REP simple.

The imported CAD models are treated by the software as a single object composed of non-pure non-convex forms, considered by V-REP to be the most complex kind of object that can be simulated. A close inspection of this model shows that it is composed of a large number of triangles assembled on a complex mesh. The more detail this model has, the more triangles are needed to make it. Figure 20 shows one example of this composition in detail.

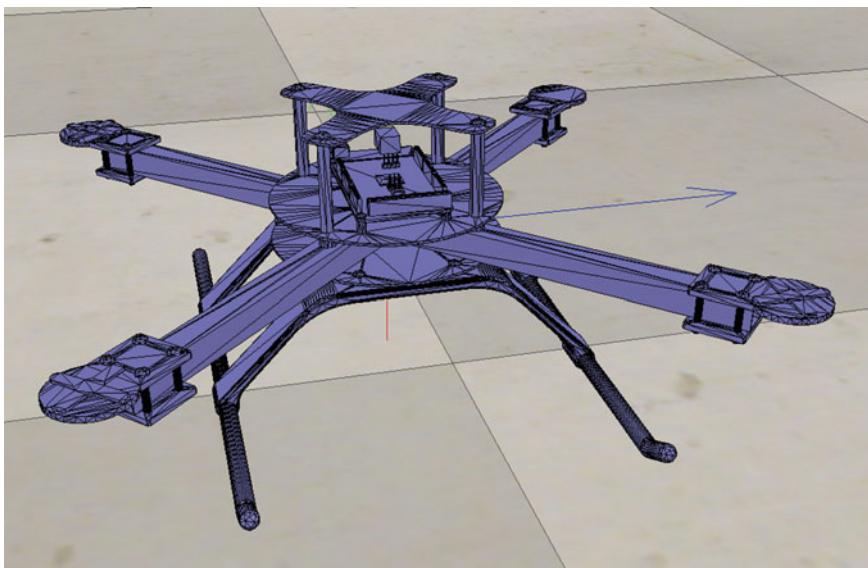


Fig. 20 Details of the model frame composition before triangle minimization

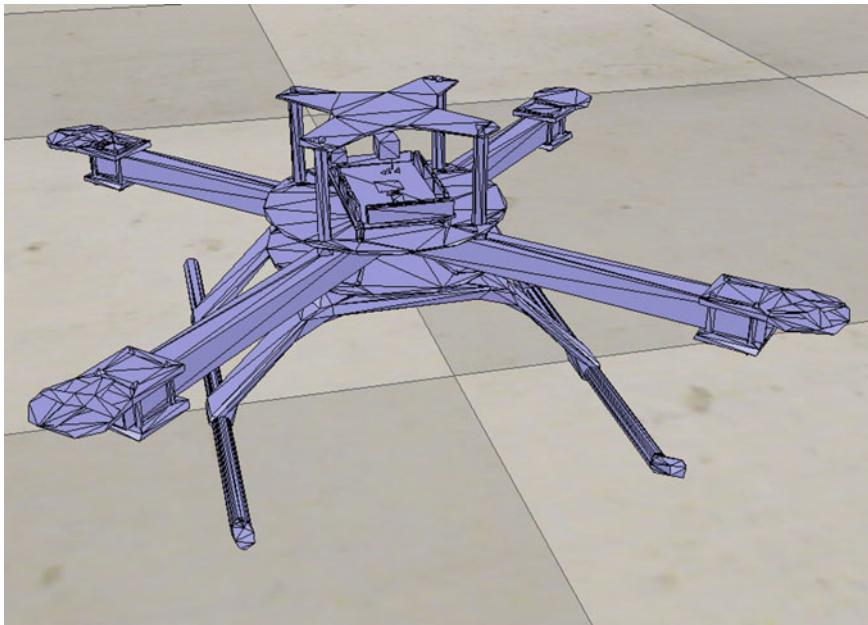


Fig. 21 Details of the model frame composition after triangle minimization

To minimize this effect, we can reduce the complexity of the frame using the tools “decimate selected shape” and “extract inside selected shape,” both available on the Edit menu. The first tool reduces the total number of triangles in the frame by associating the triangles that belong to the same elements of the frame. This leads to a loss of detail, but such a loss may not be significant when simulating the mechanical behavior of the aircraft. This reduction can be greater or smaller in magnitude according to the chosen parameter settings in the toolbox. For this example, the default reduction of 20% downsizes the model from 16,029 to 3,205 triangles.

The second tool removes the triangles that compose the inner parts of the object not visible in the simulation. After applying these tools in sequence, the final number of model triangles is reduced to 351. Figure 21 shows the frame after these operations have been performed.

The next step is to associate the propellers with the frame at their specific positions. A propeller attached to the frame will create a force and torque on it according to the reference frame position. The force is created in the propeller’s Z-direction, and the torque is created in the XY-plane and applied to the center of mass of the frame. Hence, for the hexacopter example, six upward forces are added to push the aircraft up, and six torques make the aircraft rotates around its center of mass. If the six forces are set to be the same value, the hexacopter stays at a relatively static inclination but, without a control script, the random velocities present in the propeller

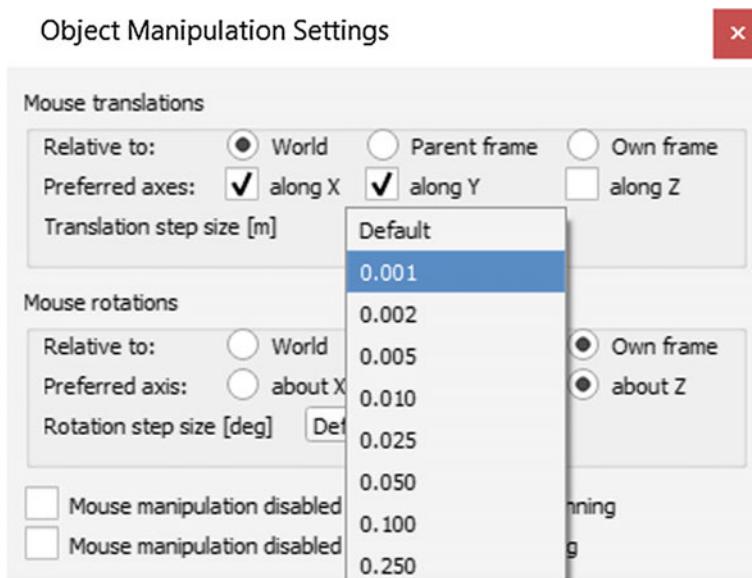


Fig. 22 Changing the object translation step factor

scripts create unbalanced forces, and the hexacopter moves in a random direction. To avoid undesired yaw rotation, it is necessary to put negative and positive signals in the propeller scripts, depending on their position on the frame, just as for real aircraft propeller rotation. Figure 25 shows the hexacopter frame and the six propellers before they are attached to their positions, and the association details.

In the scene hierarchy, the six propellers are not part of the frame assembly. The next step is to associate each one with the frame by first selecting the desired propeller and then the frame model and clicking on

edit → make last selected object parent

After associating all the propellers with the frame, we can rename them for better identification in the scene hierarchy. To do this, we simply double click on the object name in the hierarchy list, change the name, and hit “Enter.” We must now put each propeller in the correct position on the frame. We simply select the desired propeller and click the “object shift item” toolbar button.

For a more precise positioning of the propeller, the step size translation parameter can be reduced by clicking on

tools → object manipulation settings

Here, the value was changed to 0.001, as shown in Fig. 22.

After all the propellers have been properly positioned, the example hexacopter looks like the one in Fig. 23.

When the simulation is running, each propeller applies an upward force in the frame position where it is allocated that is proportional to the speed signal set by the



Fig. 23 Assembled hexacopter

script. Thus, the mechanical behavior of the hexacopter will simulate the interactions of the six propeller forces and external forces, such weight or inertial forces. To achieve a simulation that is closer to reality, we must define the hexacopter weight according to a real model. This is done by modifying the properties in the dialog box “Rigid Body Dynamic Properties,” in

Tools → Scene Object Properties → Show Dynamic Properties Dialog

This dialog box, shown in Fig. 24 allows us to change the configuration of the dynamic parameters of the object as necessary. We can specify model mass, change the object inertia matrix, or set a specific material for its composition. The values of these properties change the way the object behaves in certain situations during the simulation. The calculation of the dynamic parameters is a complex issue and beyond the scope of this chapter, but for practical purposes, we can define the mass and center of mass for the hexacopter model. For this example, the mass was set to 0.5 kg and the center of mass to $x = 0$, $y = 0$, and $z = 0.5$ m from the origin of the coordinate system. These values were based on a real hexacopter with characteristics similar to the simulation model.

If these values are difficult to determine, the alternative is to use the software to automatically calculate the parameters. The software is able to take a specific mass for the model material and use it to calculate the values of the total mass, center of mass, and inertia matrix. The calculation is performed only on a simplified model mesh, and some differences between the calculated values and values of a real model should always be expected. To simulate the dynamics characteristics and behavior of the aircraft, a model that is closer to reality must be created, but for a great number

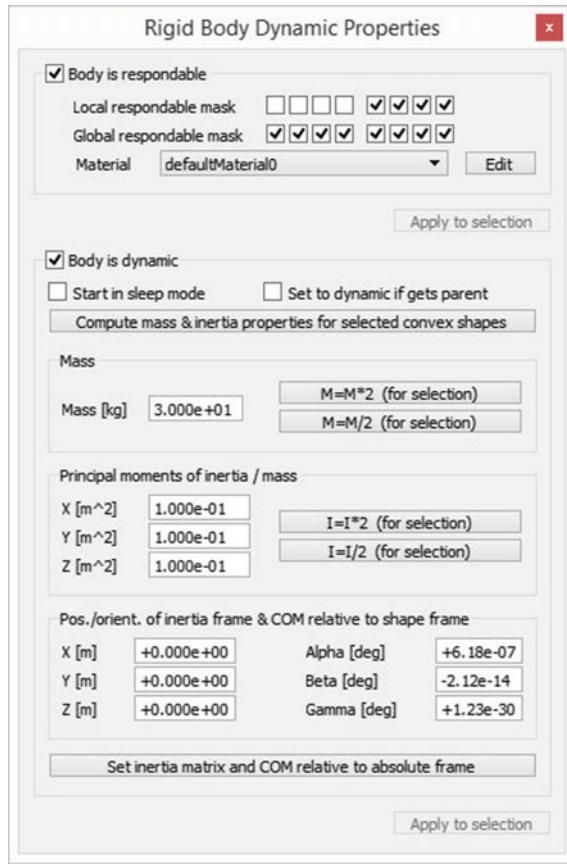


Fig. 24 Shape dynamics properties dialog box

of experiments, such as testing artificial intelligence algorithms, image processing based navigation, simultaneous localization and mapping, among others, this kind of simplified aircraft model is usually adequate (Fig. 25).

To complete our model, is necessary to associate the sensors for the control and application data achievement. V-REP has a set of interesting sensor models, including accelerometers, gyroscopes, Global Positioning System (GPS), and some commercial sensors such as laser scanners and Kinect. To use any of these sensors, it is only necessary to add it to the scene and, if applicable, associate it with the aircraft frame. For this example, some basic sensors were associated with the hexacopter frame: an accelerometer, gyroscope, GPS, and laser pointer distance sensor, not all of which were used for aircraft control. The laser range finder and gyroscope are sufficient to control the vehicle position and stabilize the flight in this case, but other sensors can be used to improve the control and movement capacities. The group of sensors collects signals and sends them to the hexacopter control script in V-REP as well as to ROS via the ROS node. Each of these sensors runs a specific script for its work. By

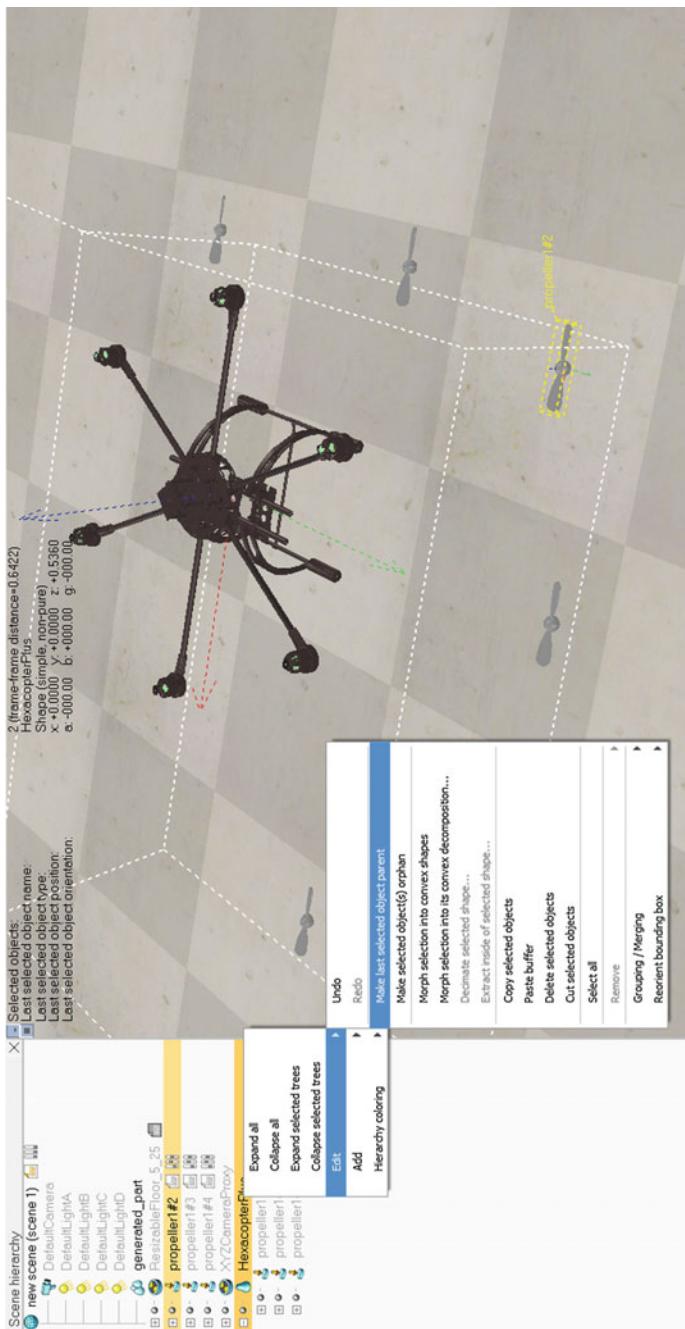


Fig. 25 Scene showing the hexacopter frame plus six propellers

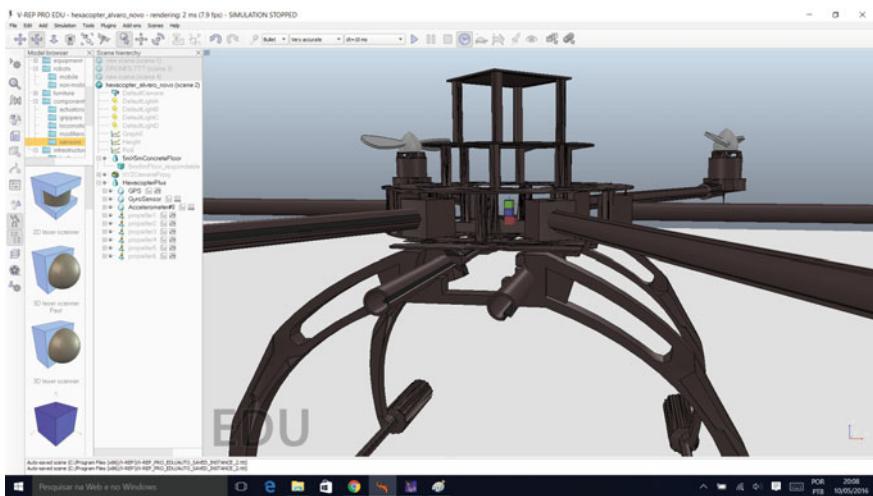


Fig. 26 Sensors on the hexacopter frame

changing these scripts, a user is able to achieve some different measuring parameters for a specific sensor, or even change its global operation.

Figure 26 shows the three selected sensors positioned at the center of the hexacopter frame. Each one of these sensors is represented as a small cube in the scene. On the left side of the image, the hierarchical list of the hexacopter objects shows the sensors and sensors scripts as a part of the frame. We now analyze the behavior of the individual sensors by looking at the sensor scripts.

The first sensor we analyze is the GPS sensor. This sensor returns the value of the spatial X, Y, and Z position related to the origin of the scene's world coordinates. One interesting point is that the script adds some “noise” to the position measurement to simulate the position measurement error of a real GPS sensor. This sensor is not based on a commercial model, but is a simple model provided by the software, so its performance will not be close to real GPS systems. For specific experiments, more accurate models probably will be needed.

The second sensor we consider is the accelerometer. This sensor calculates the object acceleration about the X-, Y-, and Z-axes of the hexacopter frame. This sensor does not add noise to the measured values like the GPS sensor does, but noise can be added if necessary by changing its associated script, just as for all the other sensors provided by the software.

The third sensor is the gyroscope. The gyroscope measures the angular velocity of the object about the X-, Y-, and Z-axes and returns the results. These values are essential for the correct operation of aircraft controller.

The laser range finder sensor is added to the base of the hexacopter frame, pointing at the ground. The function of this sensor is to obtain the absolute height of the aircraft. The laser pointer sensor measures the distance between the laser emitter and

the object that reflects the laser beam. In the MATLAB control script example, this sensor is pointed toward the ground and returns the height of the aircraft.

To run a simulation of the scene, it is necessary to associate the V-REP control scripts with the hexacopter model and the propellers. The scripts used in this example are available in the book's online repository. Download all the scripts, then copy and paste the text associated with each element into the appropriate script. To open an object script such as the one for the hexacopter, click on the object script icon present in the scene hierarchy window. Replace all the text present in this script with that of the downloaded one.

The hexacopter script implements a simple position control based on a PID algorithm. The operation of this PID is not the focus of this chapter, but it is needed to allow the hexacopter model to fly relatively in a stable manner and move during the simulation. To change the position of the hexacopter, move the slider buttons of the "Position controller." Small changes are better at each change to avoid losing control of the model. The second slider buttons allow us to set "perturbations" in the hexacopter position to show the work of the PID algorithm when it searches for the original position.

5 ROS Virtual Hexacopter Control

After creating the hexacopter virtual testing environment, it is necessary to make the vehicle compatible with the ROS. The ROS is the communication tool between the control nodes (i.e., any software, in this case MATLAB Simulink) and the real or virtual vehicle. For this example, a set of ROS nodes was specified to provide information about the propeller speeds, hexacopter linear and angular speed twists, transformations between the reference systems, absolute position of the hexacopter in the map, hexacopter position relative to the reference odometry (starting position), and the laser sensor reading, which is responsible for determining the distance from the ground. All these nodes are illustrated in the text below.

```
/hexacopter/ground_distance  
/hexacopter/laserscan  
/hexacopter/odom  
/hexacopter/pose  
/hexacopter/propeller1  
    /hexacopter/propeller2  
/hexacopter/propeller3  
/hexacopter/propeller4  
/hexacopter/propeller5  
/hexacopter/propeller6  
/hexacopter/twist  
/rosout  
/rosout_agg  
/tf  
/vrep/info
```

Once the ROS nodes have been specified, it is necessary to create the V-REP script to enable communication between them. Two functions are used to make the script work with the ROS, one for the task of publishing (*simExtROS_enablePublisher*) and the other for subscription (*SimExtROS_enableSubscriber*). A more detailed description of these functions can be found in the simulator tutorials in [3, 4]. The full script can be seen in the code below.

V-REP script for Hexarotor interface with ROS

```

if (sim_call_type == sim_childscriptcall_initialization) then

    --- Start motor velocities
    motor1 = 0
    motor2 = 0
    motor3 = 0
    motor4 = 0
    motor5 = 0
    motor6 = 0

    --- Create float signal with propeller velocities
    simSetFloatSignal('prop1',motor1)
    simSetFloatSignal('prop2',motor2)
    simSetFloatSignal('prop3',motor3)
    simSetFloatSignal('prop4',motor4)
    simSetFloatSignal('prop5',motor5)
    simSetFloatSignal('prop6',motor6)

    --- Handles for sensors and frame
    hexaHandle = simGetObjectHandle('Hexacopter_ROS')           --
        hexarotor handle
    hokuyoHandle = simGetObjectHandle('Hokuyo')                 --
        laserscan handle
    acelHandle = simGetObjectHandle('Accelerometer')           --
        accelerometer handle
    gyroHandle = simGetObjectHandle('GyroSensor')              --
        gyroscope handle
    laserHandle = simGetObjectHandle('LaserPointer_sensor')     --
        ground sensor handle

    --- Reference handles
    odomHandle = simGetObjectHandle('odom')          -- odometry
        reference
    mapHandle = simGetObjectHandle('map')            -- map reference

    --- Create ROS subscribers for propellers velocities
    simExtROS_enableSubscriber('/hexacopter/propeller1',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop1')
    simExtROS_enableSubscriber('/hexacopter/propeller2',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop2')
    simExtROS_enableSubscriber('/hexacopter/propeller3',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop3')
    simExtROS_enableSubscriber('/hexacopter/propeller4',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop4')
    simExtROS_enableSubscriber('/hexacopter/propeller5',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop5')
    simExtROS_enableSubscriber('/hexacopter/propeller6',1,
        simros_strmcmd_set_float_signal,-1,-1,'prop6')

    --- Create ROS publishers for odometry, pose and ground sensor
    simExtROS_enablePublisher('/hexacopter/odom',1,
        simros_strmcmd_get_odom_data,hexaHandle,odomHandle,'')

```

```

simExtROS_enablePublisher('/hexacopter/pose',1,
    simros_strmcmd_get_object_pose,hexaHandle,odomHandle,'')
simExtROS_enablePublisher('/hexacopter/ground_distance',1,
    simros_strmcmd_get_float_signal,-1,-1,'laserPointerData')

--- Create ROs publishers for transformations
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    laserHandle,hexaHandle,'')
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    gyroHandle,hexaHandle,'')
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    acelHandle,hexaHandle,'')
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    hokuyoHandle,hexaHandle,'')
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    hexaHandle,odomHandle,'')
simExtROS_enablePublisher('tf',1,simros_strmcmd_get_transform,
    odomHandle,mapHandle,'')

end

```

In this script, the vehicle transformation sequence is also created, according to the ROS standard specified in [5], which results in the transformation trees shown in Fig. 28.

The ROS interface with MATLAB is performed using the Robotics Systems Toolbox, available in the Simulink Toolbox, which is present in 2015 and later versions of MATLAB, only for Linux operating systems. For more information about this topic, see [6–8]. The toolbox consists of three building blocks, one to publish information to the ROS (*Publish*), another to read information from the ROS (*Subscribe*), and the last one to create messages (*BlankMessage*), as illustrated in Fig. 27 (Fig. 28).

The hexacopter can be operated using a set of read/write actions, performed by Simulink on ROS, using the Robotics Systems Toolbox. The result is shown in Figs. 29 and 30. These diagrams can be grouped into a subsystem that deals with the connection interface of the virtual hexacopter using the ROS, represented by a

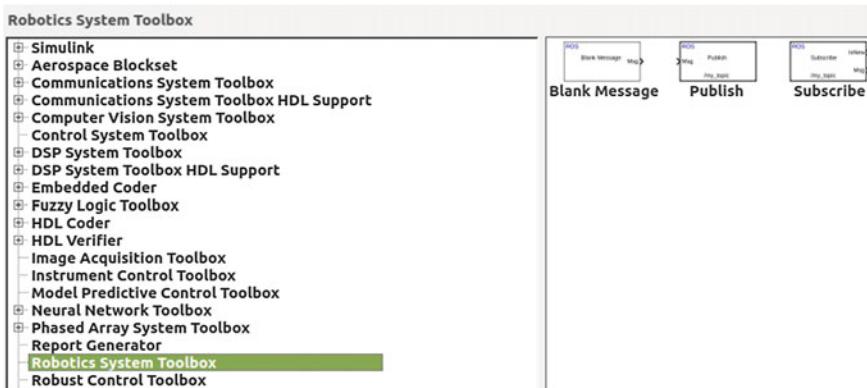


Fig. 27 Robotics System Toolbox

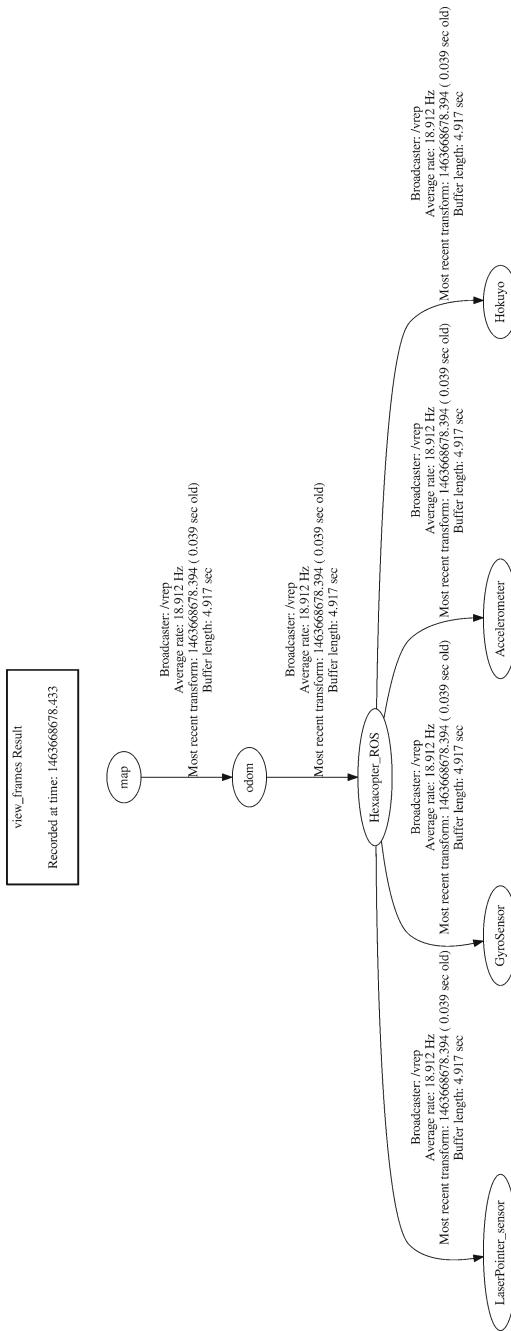


Fig. 28 Hexacopter frames (ROS TS)

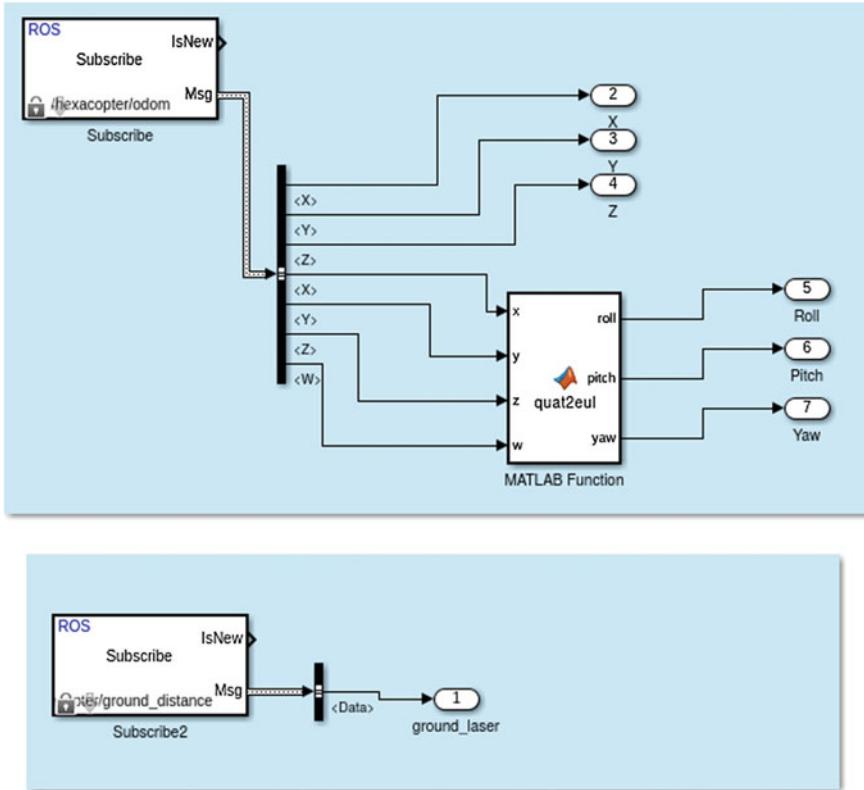


Fig. 29 Matlab interface for ROS publishers and subscribers - 1

large block with outputs (actuators for the publishers) and inputs (subscribers for the sensors).

The odometry represents the hexacopter position and orientation relative to the initial position. This information is a *nav type information* (*_msgs/Odometry*) to ROS.

The orientation is expressed using a vector that consists of four elements (q_x, q_y, q_z, q_w), used to extract the hexacopter orientation. The necessary calculations are done by a Simulink block called *MATLABfunction*. In this block, a MATLAB script was written to calculate the Euler angles, as presented below.

MATLAB script for conversion of quaternion to RPY angles

```
function [roll,pitch,yaw] = quat2eul(x,y,z,w)
% Convert the quaternion to a roll-pitch-yaw
eul = quat2eul([w,x,y,z]);
roll = eul(2);
pitch = eul(1);
yaw = eul(3);
end
```

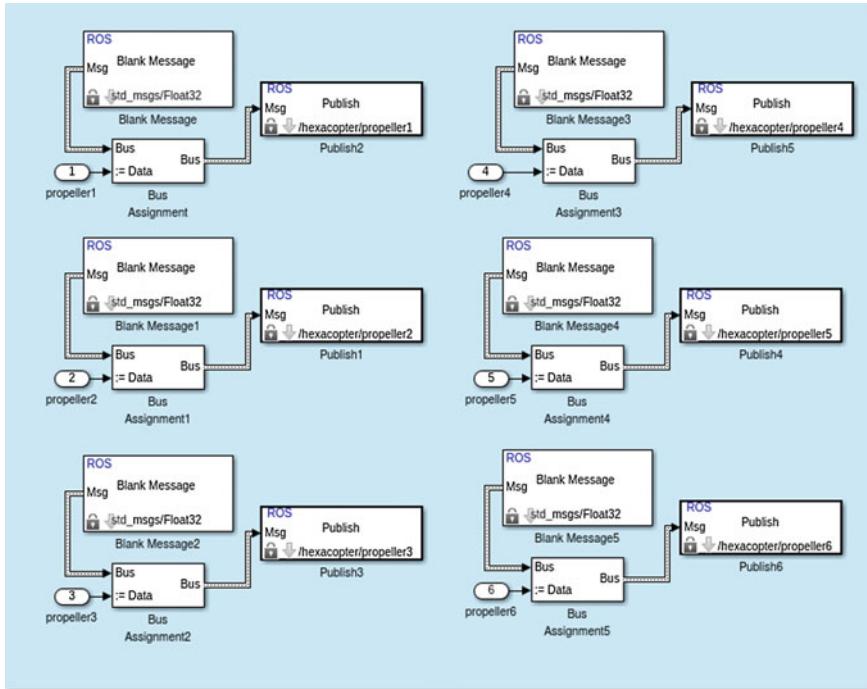


Fig. 30 Matlab interface for ROS publishers and subscribers - 2

To perform the hexacopter movements, another MATLAB function was written, using the *MATLABfunction*. This is necessary to send the speed signals that increase or decrease each propeller velocity depending on the desired motion. The motion code for this example is shown below.

MATLAB script for Hexarotor

```

function [prop1,prop2,prop3,prop4,prop5,prop6] = hexarotor(roll,
pitch, yaw, thrust)

hovering = 0.42;

thrust = thrust + hovering;

prop1 = thrust +0 -(pitch/2) +yaw;
prop2 = thrust +0 -(pitch/2) -yaw;
prop3 = thrust -roll +0 +yaw;
prop4 = thrust +0 +(pitch/2) -yaw;
prop5 = thrust +0 +(pitch/2) +yaw;
prop6 = thrust +roll +0 -yaw;

prop1=single(prop1);
prop2=single(prop2);
prop3=single(prop3);
prop4=single(prop4);
prop5=single(prop5);
prop6=single(prop6);

end

```

With the motion blocks generated, is next necessary to create a control script mechanism to correct the motion stabilization of the aircraft model. For this example, a classical PID controller was created in MATLAB. The objective of this example is not to discuss the control algorithm project, but show how to create a virtual aircraft in V-REP and perform flight control operations on it using MATLAB via ROS as a basis for more complex experimentation. Hence, the principal task of this controller is to set a fixed spatial position for the hexacopter without significant variation. This is achieved by measuring the roll, pitch, yaw, and height and performing corrections on the propellers to reduce the position and inclination error to zero. For the proposed example, a height of 1 m at the ($x = 0$, $y = 0$) position was set as the target. The result of the control algorithm is shown in Fig. 31.

In the ROS context, the inclusion of all these nodes and topics results in the logical structure shown in Fig. 32.

The velocity of the propellers is controlled in the MATLAB Simulink Toolbox. One way to change this simulation example and make the hexacopter move from a fixed point is by changing the code in this block. It is a good initial task to better understand all the simulation parts and prepare for creating other kinds of applications.

6 Final Considerations

This chapter was meant to be a starting point for more complex applications. The objective was to provide the minimal background necessary to enable the development of a simulated simple multirotor virtual aircraft control, making it possible for interested users to work with the tools without spending a long time learning them. Because of this, all the examples were simplified, and the implementation of real applications based on these tools will demand some additional work. All the scenes and scripts created can be download in <https://sourceforge.net/projects/rosbook-2016-chapter-4/files/>. The V-REP simulation software is a powerful tool for developing several robot and automation simulations. The authors strongly recommend the study of V-REP's manual, which is cited in the bibliography, and also following the software tutorials before starting to work with more complex simulations. The models, scripts, code, and other materials of interest used in this chapter are accessible online in the book's digital repository. Additional questions will be accepted by the authors at any time.

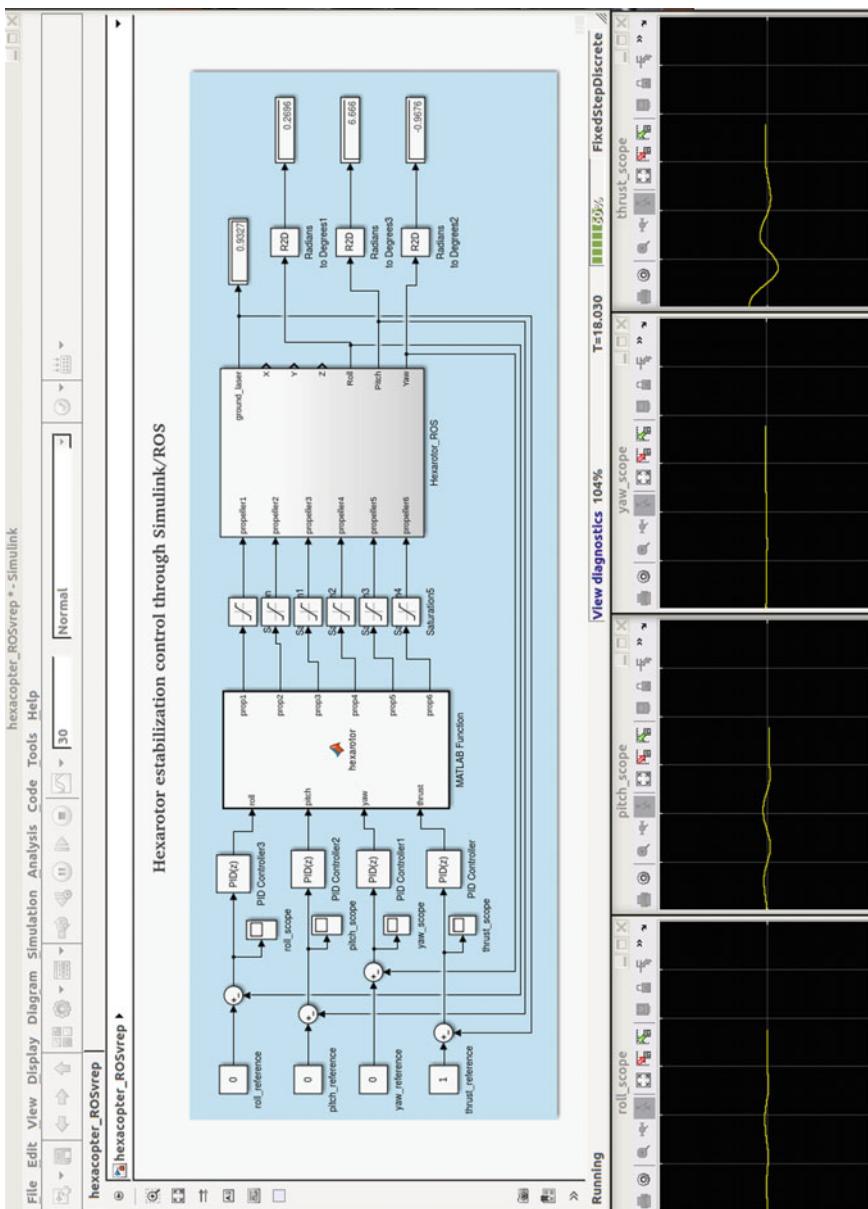


Fig. 31 Hexacopter controller in Simulink

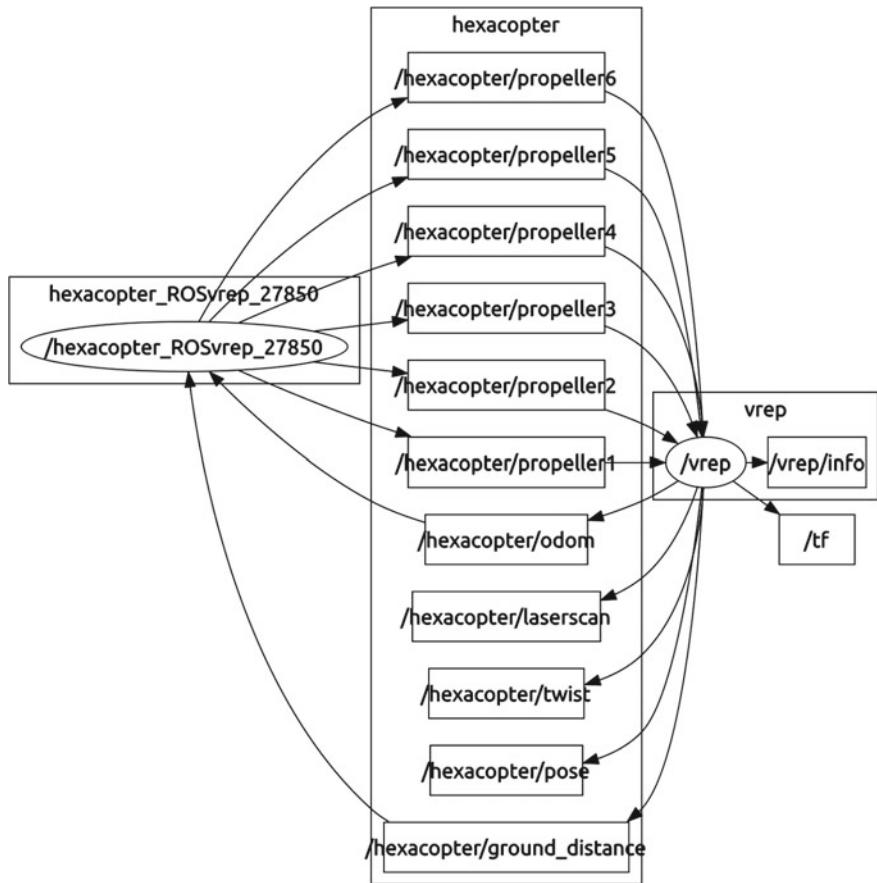


Fig. 32 ROS nodes for Simulink control of Hexarotor

References

1. Coppelia and Robotics. 2015. Virtual robot experimentation platform user manual version 3.3.0. Technical report, Coppelia Robotics. <http://www.coppeliarobotics.com/helpFiles/>
2. GrabCAD. 2016. GrabCAD Community Web Site GrabCAD. <https://grabcad.com/library>
3. Coppelia and Robotics. 2015. V-rep rosplugin publishers. Technical report, Coppelia Robotics. <http://www.coppeliarobotics.com/helpFiles/en/rosPublishers.htm>
4. Coppelia and Robotics. 2015. V-rep rosplugin subscribers. Technical report, Coppelia Robotics. <http://www.coppeliarobotics.com/helpFiles/en/rosSubscribers.htm>
5. Meeussen, W. 2015. Coordinate Frames for Mobile Platforms Ros rep-105 coordinate frames for mobile platforms. Technical report, Ros.org. <http://www.ros.org/reps/rep-0105.html>
6. Mathworks. 2016. Get Started with ROS in Simulink. Matlab Documentation, Mathworks. <http://www.mathworks.com/help/robotics/examples/get-started-with-ros-in-simulink.html>
7. Corke, P. 2015. Integrating ros and matlab [ros topics]. *IEEE Robotics Automation Magazine* 22 (2): 18–20.

8. Mathworks. 2016. Get started with ros. Technical report, Mathworks. <https://www.mathworks.com/help/robotics/examples/get-started-with-ros.html>

Author Biographies

Alvaro Rogério Cantieri has been an Associate Professor at the Federal Institute of Paraná (IFPR) since 2010. He obtained his undergraduate degree and Master's degree in electronic engineering at the Federal University of Paraná 1994 and 2000, respectively. He is currently studying for his Ph.D. in electrical engineering and industrial informatics at the Federal University of Technology - Paraná (Brazil). He started his teaching career in 1998 in the basic technical formation course of the Politecnical Institute of Parana (Parana, Brazil) and worked as a Commercial Director of the RovTec Engineering Company, focusing on electronic systems development. His research interests include autonomous multirotor aircraft, image processing, and communications systems.

André Schneider Oliveira obtained his undergraduate degree in computing engineering at the Universidade of Itajaí Valley (2004), Master's degree in mechanical engineering from Federal de Santa Catarina University (2007), and Ph.D. degree in Automation Systems from Santa Catarina University (2011). He works as an Associate Professor at the Federal University of Technology - Paraná (Brazil). His research interests include robotics, automation, and mechatronics, mainly navigation systems, control systems, and autonomous systems.

Marco Aurélio Wehrmeister received his Ph.D. degree in computer science from the Federal University of Rio Grande do Sul (Brazil) and the University of Paderborn (Germany) in 2009 (double-degree). In 2009, he worked as a Lecturer and Postdoctoral Researcher for the Federal University of Santa Catarina (Brazil). From 2010 to 2013, he worked as tenure-track Professor with the Department of Computer Science of the Santa Catarina State University (Brazil). Since 2013, he has worked as a tenure-track Professor in the Department of Informatics of the Federal University of Technology - Paraná (UTFPR, Brazil). From 2014 to 2016, he was Head of the MSc course on Applied Computing at UTFPR.

João Alberto Fabro is an Associate Professor at the Federal University of Technology - Parana (UTFPR), where he has worked since 2008. From 1998 to 2007, he was with the State University of West-Parana. He has an undergraduate degree in informatics from the Federal University of Paraná (1994), a Master's degree in computing and electrical engineering from Campinas State University (1996), a Ph.D. degree in electrical engineering and industrial informatics from UTFPR (2003) and recently became a Postdoctoral Researcher at the Faculty of Engineering, University of Porto, Portugal (2014). Has experience in computer science, especially computational intelligence, and is actively researching the following subjects: computational intelligence (neural networks, evolutionary computing, and fuzzy systems) and autonomous mobile robotics. Since 2009, has participated in several robotics competitions in Brazil, Latin America, and the World Robocup with both soccer robots and service robots.

Marlon de Oliveira Vaz obtained his undergraduate degree in computer science from the Pontifical Catholic University (PUCPR -1998) and Master's degree in mechanical engineering from PUCPR (2003). He is now a Teacher at the Federal Institute of Parana and pursuing a Ph.D. in electrical and computer engineering at the Federal University of Technology – Parana. He works mainly in the following research areas: graphical computing, image processing, and educational robotics.

Building Software System and Simulation Environment for RoboCup MSL Soccer Robots Based on ROS and Gazebo

**Junhao Xiao, Dan Xiong, Weijia Yao, Qinghua Yu,
Huimin Lu and Zhiqiang Zheng**

Abstract This chapter presents the lesson learned during constructing the software system and simulation environment for our RoboCup Middle Size League (MSL) robots. The software is built based on ROS, thus the advantages of ROS such as modularity, portability and expansibility are inherited. The tools provided by ROS, such as RVIZ, rosbag, rqt_graph just to name a few, can improve the efficiency of development. Furthermore, the standard communication mechanism (topic and service) and software organization method (package and meta-package) introduces the opportunity for sharing codes among the RoboCup MSL community, which is a fundamental issue to forming hybrid teams. As known, to evaluate new algorithms for multi-robot collaboration on real robots is expensive, which can be done in a proper simulation environment. Particularly, it would be nice if the ROS based software can also be applied to control the simulated robots. As a result, the open source simulator Gazebo is selected, which offers a convenient interface with ROS. In this case, a Gazebo based simulation environment is constructed to visualize the robots and simulate their motions. Furthermore, the simulation has also been used to evaluate new multi-robot collaboration algorithms for our NuBot RoboCup MSL robot team.

Keywords Robot soccer · Gazebo · ROS · Multi-robot collaboration · Simulation

J. Xiao · D. Xiong · W. Yao · Q. Yu · H. Lu (✉) · Z. Zheng

College of Mechatronics and Automation, National University
of Defense Technology, Changsha 410073, China
e-mail: lhnew@nudt.edu.cn

J. Xiao

e-mail: junhao.xiao@ieee.org

D. Xiong

e-mail: xiongdan@nudt.edu.cn

W. Yao

e-mail: weijia.yao.nudt@gmail.com

Q. Yu

e-mail: yuqinghua163@163.com

Z. Zheng

e-mail: zqzheng@nudt.edu.cn

1 Introduction

RoboCup,¹ short for Robot World Cup is an international initiative to foster the research in artificial intelligence (AI) and mobile robotics, by offering a publicly appealing, but formidable challenge. In other words, it is a perfect combination of sport and technology, thus has attracted many researchers and students. Since founded in 1997, it has promoted the research field for almost two decades [1, 2]. The final goal of RoboCup is that a team of fully autonomous humanoid soccer robots will beat the human World Cup champion team by 2050 [3].

Besides soccer games, other competition stages have been introduced into RoboCup along with its growth. As a result, the contest currently has six major competition domains, namely RoboCup Soccer, RoboCup Rescue, RoboCup@Home, RoboCup@Work, RoboCup Logistics League and RoboCupJunior, and each domain has several leagues and sub-leagues. More information can be found in the Robot World Cup book series published by Springer [4, 5].

For RoboCup middle size league (MSL), the robots can be designed freely as long as they stay below a maximum size and a maximum weight. The game is played on a carpet field at the size of $18\text{ m} \times 12\text{ m}$, with white lines and circles as landmarks for localization. In the competition, all the robots are completely distributed and autonomous, which means they must use their own on-board sensors to perceive the environment and make decisions. According to the rules, wireless communication is allowed to share information among a team of robots, which can help the cooperation and coordination. Therefore, RoboCup MSL is a standard and challenging real-world test bed for multi-robot control, robot vision and other relative research subjects.

As one league with the longest history among RoboCup, lots of scientific and technical progresses have been achieved in RoboCup MSL, an overview can be found in [6]. Its games are also becoming more and more fluent and fierce. For example, the robots can actively handle the ball for stepping forwards, turning and stepping backwards, can make dynamic long passes, and their velocity can reach about 5 m/s , etc. Therefore, in recent years, the RoboCup MSL final has been serving as the grand finale of RoboCup, which gives the opportunity to all audiences and participants to enjoy the game together. A typical competition scenario has been drawn in Fig. 1. However, this brings lots of difficulties for new teams to catch-up, because it is not easy and very time-consuming to design and implement a team of RoboCup MSL soccer robots from the very beginning.

Since its birth in 2010, as an open source software, ROS has attracted a huge number of robotic researchers and hobbyist, which have been serving as an active and high-productive community to boost the development of ROS. The community not only optimizes the code but also stands in the front of robotic research, i.e., implementations of state-of-the-art algorithms can be found in ROS. Therefore, ROS is becoming the *de facto* standard for robotic software. Built under ROS, the robot software components can be well and easily organized. Strictly speaking, the code

¹<http://www.robocup.org/>.



Fig. 1 A typical scenario of the RoboCup MSL competition: a match between TU/e and NuBot in RoboCup 2014 João Pessoa, Brazil

implementation can achieve high modularity and re-usability. Meanwhile, lots of useful tools have been provided by ROS, which can ease data logging and sharing, and code sharing among RoboCup MSL teams.

In April 2014, a question is raised among our team, i.e., whether ROS is suitable for RoboCup MSL? As a result, we decided to drop our self-developed software framework (more than 10 years development) and build the software system for our soccer robot based on ROS. Then in July, we participated RoboCup 2014 with robots with new “soul”. The competition brought two positive remarks. First, the software was more robust than ever before. Second, the work was acknowledged by other teams. In this paper, we will detail the ROS based software, and hope to provide a valuable reference for building ROS based software for distributed multi-robot systems.

In addition, to evaluate new algorithms for multi-robot collaboration using real robots is very time consuming, which demands a high-fidelity simulation environment. Particularly, it would be of efficiency if the simulated robot has the same control interface as the real robot. In fact, there are many robotic simulators either commercially available or open source, such as V-REP [7], Gazebo [8], Webots [9], LpzRobots [10], just to name a few. Detailed introduction and comparison of the state-of-the-art robotics simulators can be found at [11, 12]. Among the simulators, Gazebo offers a convenient interface, i.e., from the software interface point of view, there is no difference between controlling a real robot and its Gazebo dummy. In other words, the algorithms evaluated using the simulation environment can be applied to the real robots without change. Therefore, we choose Gazebo to construct the simulation environment.

This Chapter will cover the design and implementation of the software system, simulation environment, and their interfaces for our RoboCup MSL robots. It is based on our previous work of [13, 14]. The code has been made open source, with the ROS based software can be accessed at https://github.com/nubot-nudt/nubot_ws, while the simulator can be accessed at https://github.com/nubot-nudt/gazebo_visual. A video showing a simulated match using our multi-robot simulator can be found at <https://youtu.be/rMuAZGf65AE>. The remainder of the chapter consists of the following topics:

- First, the background is introduced in Sect. 2.
- Second, a brief introduction of our NuBot multi-robot system is given in Sect. 3, including the mechanical structure, the perception sensors and the electrical system.
- Third, the ROS-based software is detailed in Sect. 4.
- Fourth, the Gazebo-based simulation environment is drawn in Sect. 5, with the focus on how we designed the same control interface for the real robots and simulated robots.
- Fifth, two short tutorials are given for single robot simulation and multi-robot simulation in Sects. 6 and 7, respectively.
- A short conclusion is given in Sect. 8.

2 Background

It is not trivial to design robots for highly competitive and dynamic environments like the RoboCup MSL, thus many hardware designs and software algorithms have been proposed, see [6] for an overview. In this section, we try to give a brief introduction from which the readers can acquire more details about the achievements in RoboCup MSL, and the efforts which have been done to cut down the difficulty in developing RoboCup MSL robots. Our previous works will also be introduced in this section.

RoboCup MSL has achieved scientific results in robust design of mechanical systems, sensor-fusion, tracking, world modelling and distributed multi-robot coordination. A special issue named “Advances in intelligent robot design for the Robocup MSL” was published in 2011 [15]. In this issue, the state-of-the-art research about mechatronics and embedded robot design, vision and world modelling algorithms, team coordination and strategy was presented. The paper [6] overviews the history and the current state of the RoboCup MSL competition, which also presents a plan to further boost scientific progress and to attract new teams to the league. Surveys about team strategies, vision systems and visual perception algorithms in RoboCup MSL can be found in [16–18].

Recently, some RoboCup trustees reported the history and state-of-the-art of RoboCup soccer leagues, in which they had very positive comments on the RoboCup MSL [19], e.g., “This Middle Size League has had major achievements during the last few years. Middle Size League teams have developed software that allows amazing

forms of cooperation between robots. The passes are very accurate and some complex, cooperatively made goals are scored after passing the ball, rather than just out-dribbling an opponent and playing individually". However, this brings lots of difficulties for new teams to catch-up. As a result, the number of RoboCup MSL teams has not rise for years. How to draw more teams to participate RoboCup MSL and then make contributes to RoboCup MSL is becoming a major problem. Facing this reality, the RoboCup MSL community has made efforts to reduce the difficulty in implementing RoboCup MSL teams. For example:

- The launch of ROP (Robotic Open Platform) [20], it facilitates the release of hardware designs of robots and modules under an open hardware license. In the repository, the robots named Turtle of team Tech United have been fully released.
- Another remarkable propose is to design and implement an affordable platform for RoboCup MSL, thus providing an easier starting point for any new team, i.e., the Turtle-5k project.² With the support from the Tech United team, the TURTLE-5k platform is developed based on the 2012 TURTLE robots, which has won the RoboCup MSL world Champion. The Value Engineering method has been employed to find out some the most cost part, where the cost could be reduced.
- Real-time efficient communication among robots is of key important for cooperation, the CAMBADA team [21] proposed a TDMA (Time Division Multiple Address) based communication protocol, which is designed for real time data sharing in a local network. CAMBADA also implemented the communication protocol, which is named real-time database tool (RTDB) [22, 23]. Furthermore, RTDB has been made open source, and several teams are using it for communication.

Although RoboCup MSL teams have made significant achievements, there are still some open problems and challenges in constructing a RoboCup MSL robot team to play with human beings:

- The robot platform should have good performance in critical aspects such as top speed and top accelerations, and be able to handle impacts. It should be easy to assemble and maintain.
- It is necessary to improve the stability of the electrical system, and the extension of sensors should be better supported.
- The robustness of the vision system should be improved to make it work reliably in both indoor and outdoor environments with highly dynamic lighting conditions.
- The software framework should support code reusing and data sharing as much as possible.

²<http://www.turtle5k.org/>.

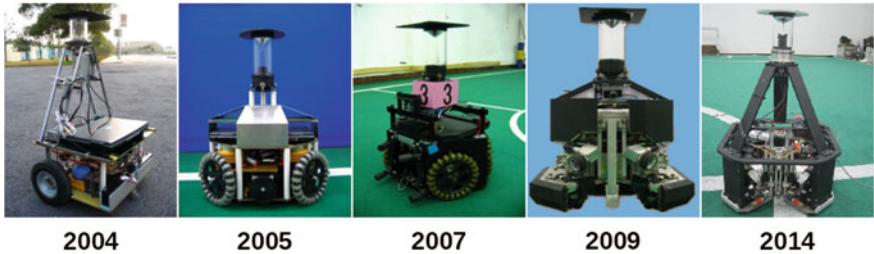


Fig. 2 The 5 generation NuBot robots

3 The NuBot Multi-robot System

Our NuBot team³ was founded in 2004. As shown in the Fig. 2, from the very beginning, 5 generations robots have been created. It can be seen that, the NuBot robots have been always using omni-directional vision system [24, 25], and have been equipped with omni-directional chassis since the second generation [26]. We had participated in RoboCup simulation and small size league (SSL) at first. Since 2006, we have been participating RoboCup MSL actively, e.g., we have been to Bremen, Germany (2006), Atlanta, USA (2007), Suzhou, China (2008), Graz, Austria (2009), Singapore (2010), Eindhoven, Netherlands (2013), Joao Pessoa, Brazil (2014), and Hefei, China (2015) [27]. We have been also participating RoboCup ChinaOpen since it was launched in 2006. This chapter will present the software system and simulation environment for the last generation robot.

3.1 Mechanical Platform

This section draws the mechanical platform of our NuBot soccer robots. When designing the robot platform, there are several criteria to be considered. First, it should comply with the rules and regulations of RoboCup MSL, namely, its size, weight and safety concerns. Second, it should have good maneuverability in order to play against others. Lastly, because malfunctions or failures are unavoidable during the intensive and fierce RoboCup MSL games, the mechanical parts should embrace high modularity such that they are easy to assemble and maintain. To fulfil these requirements, The NuBot robots have been designed modularly as shown in Fig. 3.

Currently, the regular robot and the goalie robot are heterogeneous due to their different tasks. For the regular robot, it should be able to do the same things as a human soccer player, such as moving, dribbling, passing and shooting. Therefore, the mechanical platform is subdivided into three main modules as illustrated in Fig. 3a.

³<http://nubot.trustie.net>.

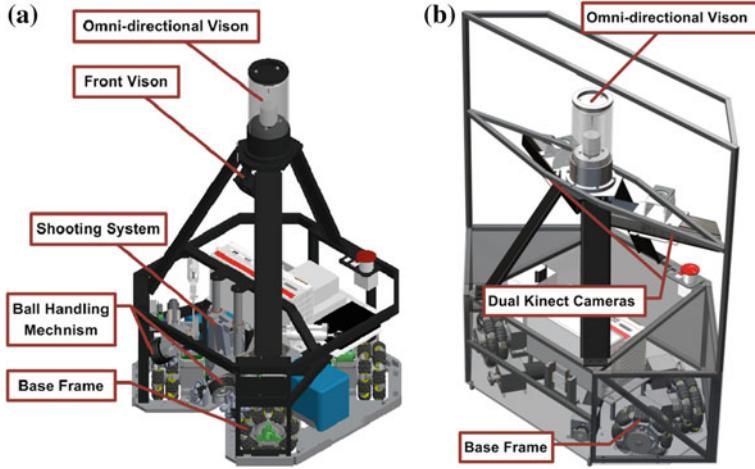


Fig. 3 **a** The regular robot. **b** The goalie robot

- the base frame;
- the ball-handling mechanism;
- the electromagnet shooting system;

For the goalie robot, the ball-handling mechanism, the electromagnet shooting device and the front vision system are removed, instead two RGB-D cameras are integrated as shown in Fig. 3b. Furthermore, to increase the side acceleration, the configuration of omni-directional wheels has also been modified.

In the below, we give a brief introduction to each part.

Base Frame The holonomic-wheeled platform, which is capable of carrying out rotation and translation simultaneously and independently, has been used by most RoboCup MSL teams [21, 28]. In the NuBot platform, our custom-designed omni-directional wheel has been utilized, as illustrated in Fig. 4a. Four such omni-directional wheels are uniformly arranged on the base as shown in Fig. 4. Despite the added costs of extra weight and extra power consumption, the four-wheel-configuration platform can generate more traction force than a normal three-wheel-configuration one, thus can improve the maneuverability. For the goalie robot, its main motion is side moving to defend coming balls. In this case, the configuration of omni-directional wheels have been modified as shown in Fig. 4c.

Ball-Handling Mechanism The ball-handling mechanism enables the robot to catch and dribble the football. As illustrated in Fig. 5, there are two symmetrical assemblies, each contains a wheel, a DC motor, a set of transmission bevel-gears, a linear displacement transducer and a support mechanism. According to the gravity and pressure from the support mechanism, the wheels are stuck to the ball when the ball is in. Therefore, they can generate various friction forces to the ball, and make it

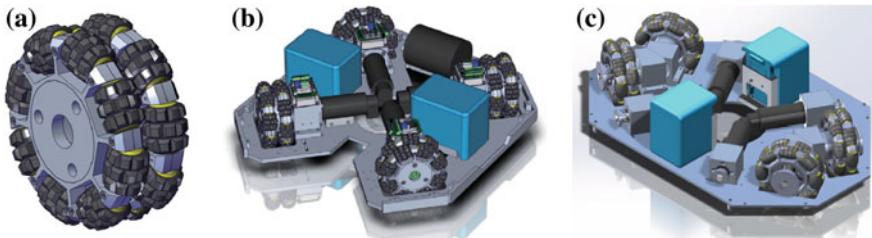
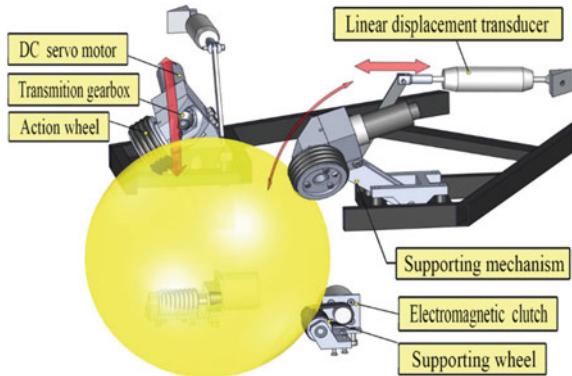


Fig. 4 **a** The custom designed omni-directional wheel. **b** Base for regular robots. **c** Base for the goalie robot

Fig. 5 The ball-handing mechanism of the NuBot



rotate in desired directions and speeds together with the soccer robot. During dribbling, the robot constantly adjusts the speed of the wheels, in order to maintain a proper distance between the ball and the robot using a closed-loop control system. This control system takes the distance between the ball and the robot as the feedback signal, which is measured by the linear displacement transducers. As the ball moves closer to the robot, the supporting mechanism will raise, then compress the transducer, otherwise the support mechanism will fall and stretch the transducer. This system effectively solves the ball-handling control problem.

Electromagnet Shooting System The shooting system enables the robot to pass and score, currently there are three ways to construct a shooting actuator, i.e., spring mechanisms, pneumatic systems and solenoids electromagnet, an overview can be seen in [29]. When using spring mechanisms, the shooting power is quite hard to control. The pneumatic systems usually need a large gas tank to generate high pressure to realize strong shooting, and the number of shots generally depends on the size of the gas tank. As a result, most RoboCup MSL teams choose to use solenoid electromagnet, whose shooting force can be high and is easier to control. Our custom-designed solenoid electromagnet has been depicted in Fig. 6, it consists of a solenoid, an electromagnet core, a shooting rod, and two linear actuators with potentiometer. The shooting rod can be adjusted in height to allow for different shooting modes,

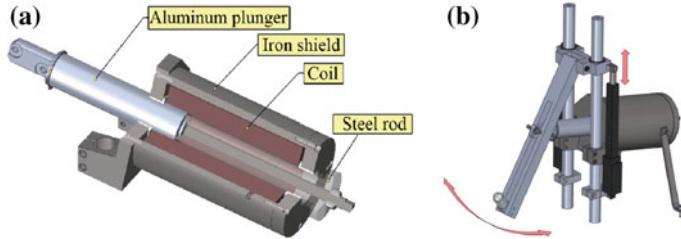


Fig. 6 **a** The solenoids electromagnet. **b** Mechanism of the shooting system

namely, flat shots for passing and lob shots for scoring. Two modes are realized using two linear actuators to move the hinge of the shooting rod to different positions. For more detail, please refer to [13].

3.2 Visual Perception System

For RoboCup MSL robots, the visual perception system is of great importance as they should be fully autonomous. There are three kind of visual perception sensors in our system, namely an omni-directional vision system, a front vision system and a RGB-D vision system. Among them, each robot has an omni-directional vision, and each regular robot has an additional front vision, while the goalie robot has dual RGB-D cameras, as shown in Fig. 3. Below a short introduction to the vision sensors is given.

Omni-Directional Vision System Almost all RoboCup MSL teams are using omni-directional vision systems, which is composed of a convex mirror and a camera pointing upward towards the mirror. The panoramic mirror plays the most important impact on the imaging quality, especially on the distortion of the panoramic image. Currently we are using the mirror designed by the team named Tech United Eindhoven [30], which has a relative simple profile, at the same time is easy to calibrate.

Front Vision The front vision system is an auxiliary sensors for the regular robots, which is a low-cost USB camera and facing down upon the ground, as shown in Fig. 3. With it, the robot can recognize and localize the ball with high accuracy when the ball is close to the robot. The position of the ball is estimated based on the pinhole camera projection model. It is of great significance for accurate ball catching and dribbling.

Dual RGB-D Cameras In the current RoboCup MSL games, most of the goals are achieved by lob shooting, so accurate estimation of the shooting touchdown-point of the ball is fundamental for the goalie robot to defend these shoots. Although the object’s 3D information can be acquired using the omni-directional vision system and a front vision system together, the accuracy cannot be guaranteed because the imaging resolution of the omni-directional vision is relative low due to its large field of view (FoV). The Kinect 2 RGB-D camera can stream out color and depth information simultaneously at the frame rate of 30 fps, whose sensing range is up to 8 m. Therefore, it is an ideal sensor to obtain the 3D ball information for the goalie robot. Thus, our goalie robot is equipped with dual RGB-D cameras, as demonstrated in Fig. 3, to recognize and localize the ball, estimate its moving trace and predict the touchdown-point in 3D space.

3.3 Industrial Electrical System

As the RoboCup MSL game becomes more and more competitive and fierce, the requirements on the robustness and reliability of the electronic system are also increasing. To improve the robustness of our robot control system, the electrical system of NuBot robots is designed based on the so called PC-based control technology, whose block diagram has been drawn in Fig. 7. As can be seen, the system uses an Ethernet-based field-bus system named EtherCAT [31, 32] to realize high-speed communication between the industrial PC and the connected modules. All vision

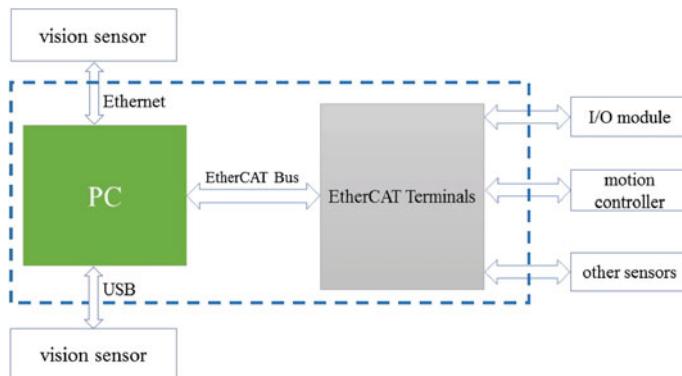


Fig. 7 The electrical system based on PC control technology. The *blue-dashed box* represents industrial PC and Ethernet-based field-bus, which are the core module of the PC-based control technology

and control algorithms are processed on the industrial PC. The industrial electrical system has been used through 2014 Brazil and 2015 China international RoboCup competitions, 2014 China RoboCup competition.

4 ROS-Based Software for NuBot Robots

The recent achievements in robotics make autonomous mobile robot play an increasingly important role in daily life. However, it is difficult to develop a generic software for different robots, e.g. it is usually difficult to reuse others' code implements. A solution named Robot Operating System (ROS), launched by Willow Garage company, provides a set of software libraries and tools for building robot applications across multiple computing platforms. ROS has many advantages: ease of use, high-efficiency, cross-platform, supporting multi-programming languages, distributed computing, code reusability, and is completely open source (BSD) and free for others to use. We also use ROS to build our NuBot software. Furthermore, our software is developed on Ubuntu, and it is also open source. For the current version, the operating system is Ubuntu 14.04, and the ROS version is indigo.

The software framework, as shown in Fig. 8, is divided into 5 main parts:

1. the Prosilica Camera node and the OmniVision node;
2. the UVC Camera node and the FrontVision node;
3. the NuBot Control node;
4. the NuBot HWControl node;
5. the RTDB and the WorldModel node.

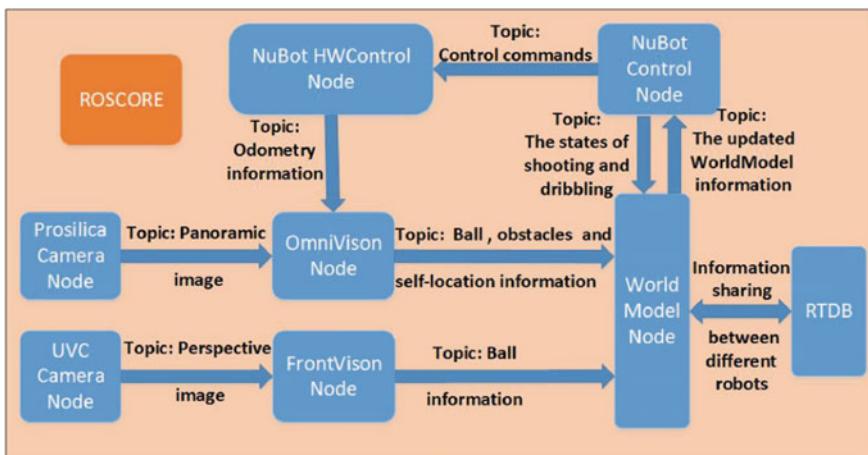


Fig. 8 The software framework based on ROS

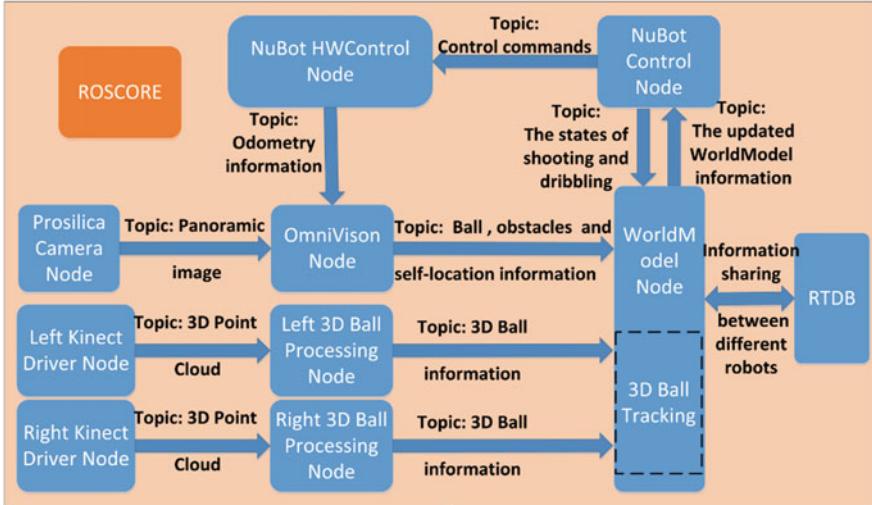


Fig. 9 The goalie software framework based on ROS

For the goalie robot, four Kinect related nodes will replace the FrontVision node and the UVC Camera node, i.e., a driver node and a 3D ball tracking node is required for each Kinect, as shown in Fig. 9. These nodes will be described in the following sub-sections.

4.1 The OmniVision Node

Perception is the base to realise the autonomous ability for mobile robots, such as path planning, motion control, self localization, action decision and cooperation. Omni-directional vision is one of the most important sensors for RoboCup MSL soccer robots. The image is captured and published by the Prosilica Camera node.⁴ It takes less than 30 ms to perform the computation of below algorithms, so the OmniVision node can be run in real-time.

Colour Segmentation and White Line-Points Detection The color lookup table is calibrated off-line. Because of its simplicity and low computational requirements, it is used to realize color segmentation. A typical panoramic image captured by our omni-directional vision system is shown in Fig. 10a in a RoboCup MSL standard field, the corresponding segmentation result is shown in Fig. 10b. As can be seen, this method can be used to distinguish the ball, green field, black obstacles and white

⁴http://wiki.ros.org/prosilica_camera.

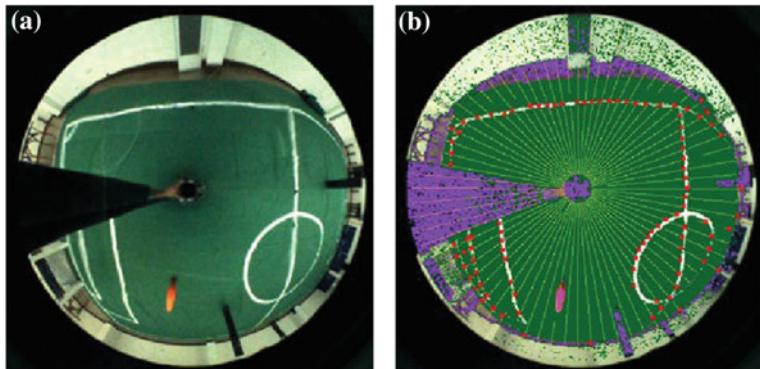
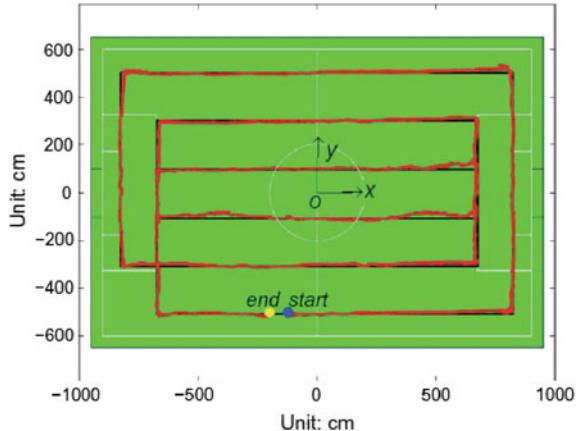


Fig. 10 **a** The image captured by our omni-directional vision system. **b** The result of color segmentation for image in **a**, for the visualization purpose, the ball has been colored with *pink* and obstacles have been colored with *purple*

line in the color-coded environment. To detect white line in the panoramic image, we search for significant color variations along some scan lines because of the different color values between the white lines and the green field. As shown in Fig. 10b, these scan lines are radially arranged around the image center, and the red points represent the resulted white line-points.

Self-localization To localize an autonomous mobile robot under a highly dynamic structured environment is still a challenge. A matching optimization algorithm has been employed to realize global localization and pose tracking for our soccer robots accurately in real-time. A brief introduction is given below, see [33] for more detail. As an off-line preprocessing step, 315 samples are acquired as the robot's candidate positions which are located uniformly in the field. Then, for the real-time global localization, the orientation is obtained by an Motion Trackers instrument (MTi). Afterwards, the match optimization localization algorithm is used to determine the real pose among the samples. Once global localized, pose tracking phase is started, where the encoders based odometry is used to obtain its coarse pose, and a Kalman Filter is employed to fuse the odometry with the match optimization result. A typical localization result is illustrated in Fig. 11, during the experiment, the robot was manually pushed to follow straight lines on the field, which is shown as black lines in the figure. The red traces depict the localization result. The mean position error is less than 6 cm.

Fig. 11 The robot's self-localization results



4.2 The FrontVision Node and the Kinect Node

The FrontVision node processes the perspective image captured and published by the UVC Camera node,⁵ and provides a more accurate ball position information when the ball is in the near front of a regular robot. The node detects the ball using a color segmentation algorithm and region growing algorithm similar to the OmniVision node. Then we can estimate the ball position based on the following assumptions. First, the ball is located on the ground. Second, the pinhole camera model is adopted to calibrate camera interior and exterior parameters off-line. Lastly, the height of the camera to the ground and its view direction is known.

3D information of the ball is of great significance for the goalie robot to intercept lob shot balls. However, using the front vision system and the omni-directional vision system to interpret depth information is difficult. Therefore, a dual RGB-D cameras setup is employed to recognize and localize the ball, estimating its moving trace in 3D space. The OpenNI RGB-D camera driver, which has been integrated into ROS, is employed for obtaining point clouds data in the Kinect driver node. Basic point cloud processing, such as noise filtering and segmentation, is based on algorithms of the Point Cloud Library (PCL) [34].

As shown in Fig. 12, in the 3D ball processing node, the same color segmentation algorithm as that in the OmniVision node is used to obtain some candidate ball regions. Then, the random sample consensus algorithm (RANSAC) [35] is used to fit a spherical model to the shape of the 3D candidate ball regions. With the proposed method, only a little number of candidate ball regions need to be fitted. Lastly, to intercept the ball for the goalie, the 3D trajectory of the ball regarded as a parabola curve is estimated and the touchdown-point in 3D space is also predicted in the 3D ball processing node, using a similar algorithm as in [36]. In total, the node takes

⁵http://wiki.ros.org/uvc_camera.

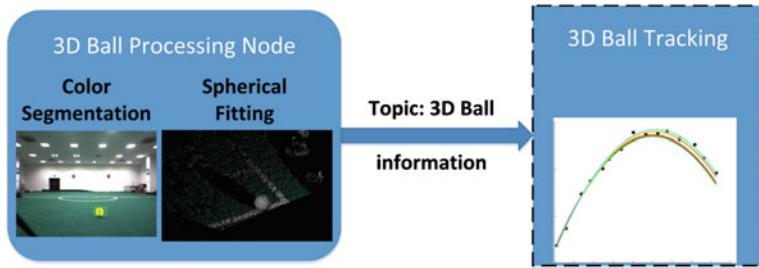


Fig. 12 The 3D ball processing data flow

about 30–40 ms to process a frame of RGB-D data, therefore can meet the real-time requirement of highly dynamic RoboCup MSL games.

4.3 The NuBot Control Node

On top level of the controllers, the NuBot soccer robots typically adopt a three-layer hierarchical structure. To be specific, the NuBot control node basically contains strategy, path planning and trajectory tracking.

The design of soccer robots aims to fulfil all the tasks completely autonomously and cooperatively. Therefore, multi-robot cooperation plays a central role in RoboCup MSL. To allocate the roles of the robots and initiate the cooperation, a group intelligence scheme is proposed to imitate the captain or the decision-maker in the competition, see [37] for detail. In our scheme, a hybrid distributed role allocation method is employed, including role evaluation, role assignment and dynamic reassignment. The soccer robot can select a proper role among the following set: attacker, defender and others. While the roles are determined, each robot is motivated to perform the corresponding tasks individually and autonomously, such as moving, defending, passing, catching and dribbling.

Path planning and obstacle avoidance is still quite a challenge under highly dynamic competition environments. To deal with it, an online path planning method based on the subtargets method and B-spline curve is proposed [38]. Benefiting from the proposed method, we can obtain a smooth path and realize real-time obstacle avoidance at a relative high speed. The method can be summarized as follows:

- generating some via-points using the subtargets algorithm iteratively;
- obtaining a smooth path by using B-spline curve between via-points; and
- optimizing the planning path via actual constraints such as the maximal size of an obstacle and the robot velocity and so on.

To track the planned path at a high speed with a quick dynamic response and low tracking error, Model Predictive Control (MPC) is utilized, as MPC can easily take into account the constraints and use the future information to optimize the current

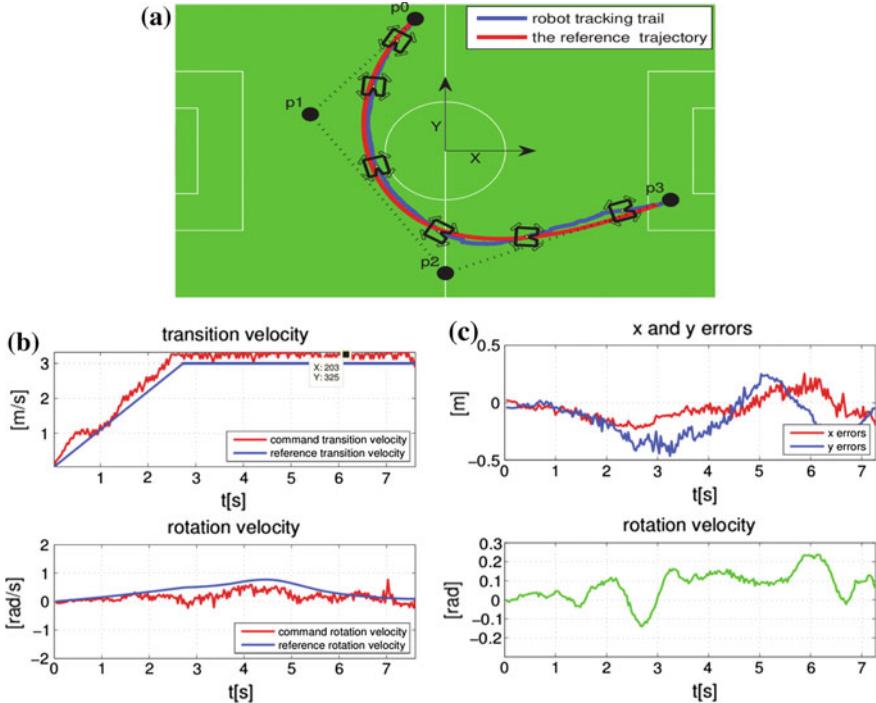


Fig. 13 A typical path tracking result of the proposed controller. **a** The robot starts at control point p_0 to track the given trajectory, and finally stops at point p_3 . The reference trajectory and the real trajectory is shown in a red curve and a blue curve, respectively. **b** The speed during the tracking, which is bounded at 3.25 m/s. **c** The tracking errors

output [26]. Firstly, a linear full-dynamic error model based on the kinematics model of the soccer robot is obtained. Then, MPC is used to design the control law to satisfy both the kinematics constraints and kinetics constraints. Meantime, Laguerre Networks is used to design the MPC controller, in order to reduce the computational time for the online application. As illustrated in Fig. 13, the robot can track the path with a quick dynamic response and low tracking errors by our proposed MPC control law.

4.4 The NuBot HWControl Node

On bottom level of the controllers, the NuBot HWControl node performs four main tasks:

1. controlling the four motors of the base frame;
2. obtaining odometry information;

3. controlling the ball-handling system; and
4. actuating the shooting system.

The ROS EtherCAT library for our robots is developed to exchange information between the industrial PC and the actuators and sensors, e.g., AD module, I/O module, Elmo controller, motor encoder, linear displacement sensor. The speed control commands calculated in the NuBot Control node are sent to four Elmo motor controllers of the base frame at 33 Hz for realizing robot motion control. Meanwhile, the motor encoder data are used to calculate odometry information, which are published to the OmniVision node. For the third task, high control accuracy and high-stability performance are achieved by feedback plus feedforward PD control for the active ball-handling system. The relative distance between the robot and the ball measured with two linear displacement sensors is regarded as feedback signal, and the robot velocity is used as the feedforward signal. For the last task, the shooting system first needs to be calibrated off-line. During competitions, the node adjusts the hinge of the shooting rod to different heights according to the received commands: flat-shooting or lob-shooting from the NuBot Control node. Furthermore, it can determine the shooting strength according to the calibration results and kicks the ball out.

4.5 *The WorldModel Node*

The real-time database tool (RTDB) [22, 23] developed by the CAMBADA team is used to realize the robot-to-robot communication. The information of the ball, the obstacles and the robot itself provided by the OmniVision node, the Kinect node and the FrontVision node are combined with the data communicated from teammates to acquire a unified world representation in the WorldModel node. The information from its own sensors and other robots is of great significance for single-robot motion and multi-robot cooperation. For example, every robot fuses all obtained ball information, and only the robot with the shortest distance to ball should catch it and others should move to appropriate positions; each robot achieves accurate positions of the obstacles and obtains the positions of its teammates by communication, thus it can realize accurate teammate and opponent identification, which is important for our robots to perform man-to-man defense.

5 Gazebo Based Simulation System

In this paper, the open source simulator Gazebo [8] is employed to simulate the motions of our soccer robots. The main reason to use Gazebo is that it offers a convenient interface with ROS, which has been used to construct software for our real robots, see Sect. 4 for detail. In addition, Gazebo also features 3D simulation, multiple physics engines, high fidelity models, huge user base and etc. Therefore,

Table 1 Properties of the robot model

Property	Value
Mass	31 kg
Moment of inertia	$I_{zz} = 2.86 \text{ kg} \cdot \text{m}^2$ $I_{xx} = I_{yy} = I_{xy} = I_{xz} = I_{yz} = 0$
Friction coefficient	0.1
Velocity decay	Linear: 0. Angular: 0
Model plugin	nubot_gazebo

the simulation system based on ROS and Gazebo can take advantage of many state-of-the-art robotics algorithms and useful debugging tools built in ROS. It can also benefit from or contribute to the active development communities of ROS and Gazebo in terms of code reuse and project co-development.

The remainder of this section is organized as follows. Section 5.1 introduces the creation of simulation models and a simulation world. Section 5.2 presents the realization of a single robot's basic motions by a Gazebo model plugin. Furthermore, in Sect. 5.3, the model plugin is integrated with the real robot code so that several robot models are able to reproduce real robots' behavior. Finally, in Sect. 5.4, three tests are conducted to validate the effectiveness of the simulation system.

5.1 Simulation Models and a Simulation World

Gazebo models, which consist of links, joints (optional), plugins (optional) and etc., are specified by SDF (Simulator Description Format)⁶ files. Besides, a simulation world, which determines lighting, simulation step size, simulation frequency and other simulation properties, is specified by a world file.

Simulation Models Models used in this simulation system include the NuBot robot model, the soccer field model and the soccer ball model.

- **Robot model:** It is composed of a chassis link without any joint. Table 1 lists some important properties specified in the robot model SDF file. Besides, another two important properties, mesh and collision that are used for visualization and collision detection respectively, are illustrated in Fig. 14. They are drawn by the open source 3D drawing tool SketchUp.⁷ Note that the collision element is not a duplicate of the model's exterior but a simplified cylinder with the same base shape and height as the model exterior. Furthermore, we do not model the real robot's physical mechanisms, such as omni-directional wheels, ball-dribbling, ball-kicking and omni-vision camera mechanisms. Therefore, this model does

⁶<http://sdformat.org/>.

⁷<http://www.sketchup.com/>.

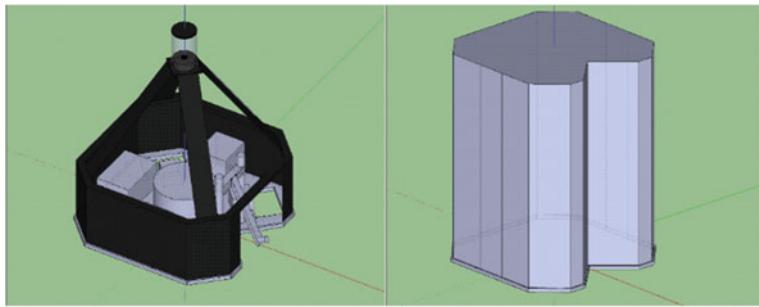


Fig. 14 Mesh and collision properties of the robot model. *Left* Mesh property; *Right* collision property

Table 2 Properties of the simulation world

Property	Value
Physics engine	Open dynamics engine
Max step size	0.007 s
Gravity	-9.8 m/s ²

not require any joints. The simplification is reasonable according to the simulation purpose: to test multi-robot collaboration strategies and algorithms. Therefore the emphasis of the simulation system is on the final effect of robot basic motions but not the complicated physical processes involved. The physical mechanisms capabilities are realized by a Gazebo model plugin that will be discussed in Sect. 5.2.

- **Soccer field model:** Images of the goal net, field ground and field lines, together with OGRE material scripts⁸ are used to construct the field model. The field is then scaled according to the 2015 RoboCup MSL rules. The collision elements are composed of each parts' corresponding geometry.
- **Soccer ball model:** The soccer ball model is built with the same attributes of a defined FIFA (Fdration Internationale de Football Association) standard size 5 soccer ball that is played in RoboCup MSL. The pressure inside the model is neglected and the collision element is a sphere of the same size of the soccer ball.

The Simulation World The world file specifies the simulation background, lighting, camera pose, physics engines, simulation step size and etc. Some important properties of the simulation world are listed in Table 2. Finally, a simulation world with three robots and a soccer ball is created, see Fig. 15.

⁸<http://www.ogre3d.org/>.

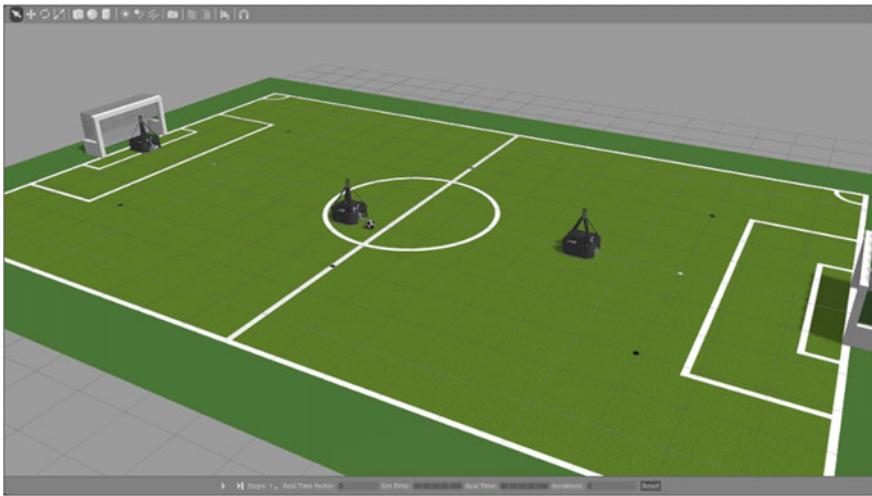


Fig. 15 The simulation world, with three robots playing a ball

5.2 Basic Motions Realization

To realize a single robot's basic motions, a Gazebo model plugin named “nubot_gazebo” is written. A model plugin is a shared library that attached to a specific model and inserted into the simulation. It can obtain and modify the states of all the models in a simulation world.

Overview of the “nubot_gazebo” Plugin When “nubot_gazebo” plugin is loaded at the beginning of a simulation process, its tasks include:

- Obtaining parameters of the soccer ball model's name, ball-dribbling distance threshold, ball-dribbling angle threshold and etc. from the ROS parameter server.
- Setting up ROS publishers, subscribers, service servers and a dynamic reconfigure server.
- Binding model plugin update function that runs in every simulation iteration.

The model plugin starts running automatically when a robot model is spawned. For example, when the robot model “bot1” is spawned, a computation graph shown in Fig. 16 is created, which is visualized by the ROS tool rqt_graph. As can be seen, there is only one node called “/gazebo”, which publishes (represented by an arrow pointing outward) and subscribes (represented by an arrow pointing inward) several topics enclosed by small rectangles. The topics inside the “gazebo” namespace are created by a ROS package called gazebo_ros_pkgs, which provides wrappers around the stand-alone Gazebo and thus enables Gazebo to make full use of ROS messages, services and dynamic reconfigure. Those inside the “bot1” namespace are created by the model plugin. All the topic names are self-explanatory. For instance, messages on the /bot1/nubotcontrol/velcmd topic are used to control the robot model's velocity.

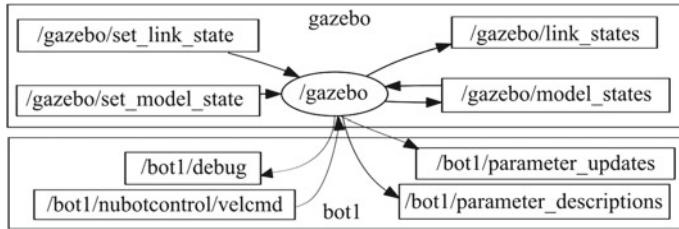


Fig. 16 The computation graph of the model plugin

Although the physical mechanism of the omni-vision camera is not simulated, the robot model is still able to obtain information of other models' positions and velocities by subscribing to the topic `/gazebo/model_states`. In addition, ball-dribbling and ball-kicking are realized by calling corresponding ROS services. They will be discussed in the following part.

Motion Realization A single robot's basic motions include omni-directional locomotion, ball-dribbling and ball-kicking.

- **Omni-directional locomotion:** Gazebo's built-in functions `SetLinearVel` and `SetAngularVel` are used to make the robot model move in any direction given any translation vector and rotation vector respectively.
- **Ball-dribbling:** If the distance between the robot and the soccer ball is within a distance threshold and the angle from the robot front direction to the ball viewing direction is also within an angle threshold, then the dribble condition is satisfied and the robot is able to dribble the ball. Under this condition, to realize ball-dribbling, the soccer ball's pose is directly and continuously set by Gazebo's built-in function to continually satisfy the dribble condition.
- **Ball-kicking:** Similarly, ball-kicking is realized by giving the soccer ball a specific velocity at the start of the kicking process. There are two ways of kicking, e.g., the ground pass and the lob shot. For the ground pass, the soccer ball does not lose contact with the ground so its initial velocity vector is calculated in the field plane. As for the lob shot, the soccer ball is kicked into the air so its speed in the up-direction should also be taken into account. Since the air resistance is trivial compared with the gravity effect, it is reasonable to assume that the ball's flight path is a parabola.

Single Robot Motions Test To test single robot's basic motions, four behavior states are defined as follows: `CHASE_BALL`, `DRIBBLE_BALL` (including two sub-states `MOVE_BALL` and `ROTATE_BALL`), `KICK_BALL`, and `RESET`. The robot model performs these motions following the behavior states transfer graph as shown in Fig. 17. The test results, as shown in Fig. 18 prove that the “`nubot_gazebo`” model plugin realizes basic motions successfully.

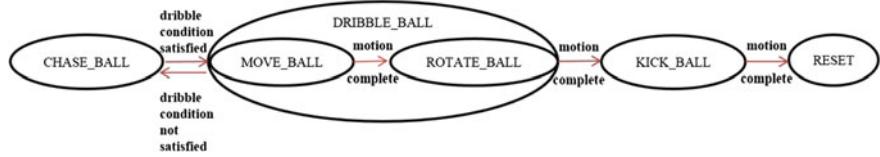


Fig. 17 Single robot behavior states transfer graph

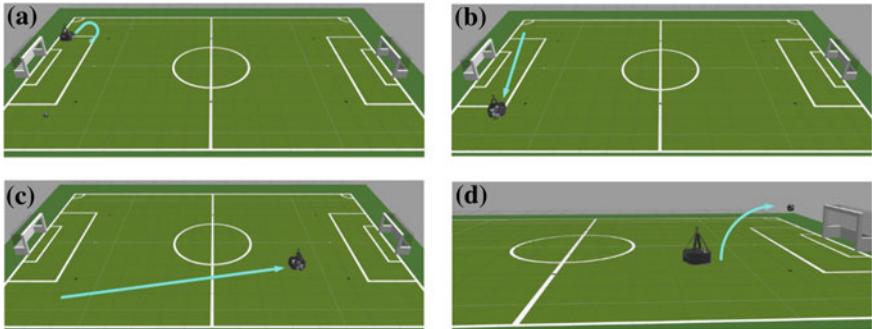


Fig. 18 Single robot simulation result. **a** Initial state; **b** CHASE_BALL state; **c** DRIBBLE_BALL state; **d** KICK_BALL state

5.3 Model Plugin and Real Robot Code Integration

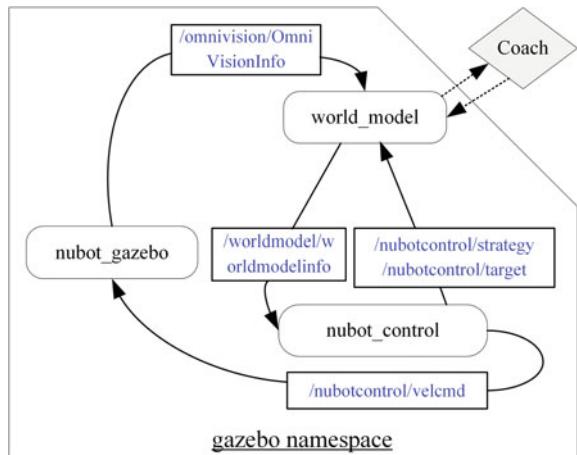
It would be better to use the same interface to control the real robots and the simulated robots. In this case, the multi-robot collaboration algorithms could be evaluated using the simulation system. Furthermore, the implementation can be directly applied to the real robots without any modification. In other words, it is significant to integrate the model plugin with the real robot code.

In the real robot code, there are eight nodes in total (see Fig. 8). Among them, “world_model” and “nubot_control” are close related to multi-robot collaboration and cooperation. In addition, there is a coach program which receives and visualizes information from each robot and sends basic commands such as game-start, game-stop, kick-off and corner-ball via RTDB.

To integrate the real robot code with the model plugin, the left five nodes which are related to hardware should be replaced by the model plugin. This successful replacement requires an appropriate interface, in other words, correct ROS messages-passing and services-calling between them. Finally, the data flow of the integration of the real robot code and the model plugin is shown in Fig. 19. There are three noticeable changes described as follows.

- Coach communicates with each robot’s “world_model” node via ROS messages: for convenience and reliability, the communication between Coach and “world_model” no longer requires RTDB in the simulation scenario. Instead, they are able to send and receive ROS messages in one local computer. In particular,

Fig. 19 The data flow graph of the integration of the real robot code and the model plugin



each robot receives messages about game status from the Coach. However, the Coach only receives the world model information from one selected robot. This is because all the robot's world model information is accurate and shared in the simulation environment, there is no need for the Coach to obtain other robots' world model information.

- An intermediary node (simulation interface) for communication among robots: in the real world scenario, robots share their own strategies information with their teammates by RTDB. However, as for simulation, it is neither practical nor necessary to use RTDB as a communication measure since all robots are simulated in one computer. Therefore, an intermediary node (simulation interface) subscribes to messages on collaboration strategies from all robots and in return, publishes new messages containing all the strategies information. So all the robots are able to share the information without the use of RTDB network communication.

In addition, topic-name-prefixing is employed for simulation to distinguish different robots. Because all the robot models use the same model plugins and are created into one simulation world, they cannot distinguish their own messages and services from others. In this case, it is necessary for each model's name to be used as a prefix to their own topic names or service names. Therefore, the robots can subscribe to their own topics or respond to their own services. These prefixes, i.e., the model names, are obtained by a bash script to guarantee that each name is mapped to the appropriate robot models as shown in Fig. 20. The bash scripts also start the simulation interface node and spawn models for Gazebo. It works as a mapping mechanism and a bridge between different separate components. This helps isolate the real robot code from the simulation components so as to improve the adaptability of the simulation system. In other words, different robot code can be easily tested in this environment since it does not depend on the simulation.

- Gaussian noise: Gaussian noise is added to the position and velocity information obtained by the robot model to mimic the real world situation.

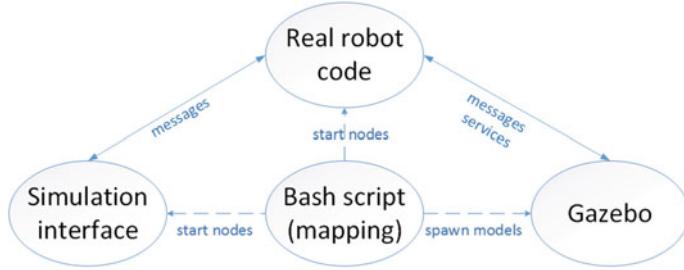


Fig. 20 The functions of the bash script

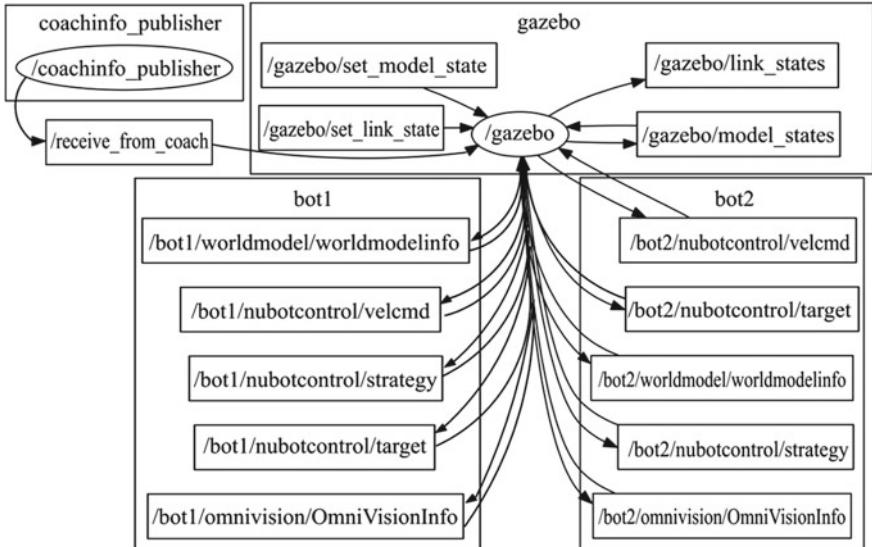


Fig. 21 The computation graph of the simulation with two robot models

Finally, two robot models bot1 and bot2 are spawned into a simulation world and the corresponding computation graph is shown in Fig. 21. Note that all the model plugins are embedded in the /gazebo node and the topic names are all prefixed by corresponding model names due to the mapping function of the bash script discussed before.

5.4 Simulation of a Match

It is also possible to simulate a match of two simulated teams, which could be used to evaluate new collaboration algorithms. Furthermore, machine learning algorithms could be used to train the simulated robots during the simulated match, and the trained

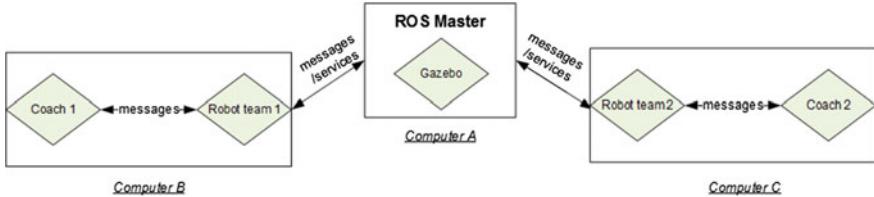


Fig. 22 The overall structure of the configuration of two simulation teams



Fig. 23 The simulation of a soccer match by two robot teams

results can be then applied to the corresponding real robots. Figure 22 shows the overall structure of the setup. There are totally three computers to simulate a soccer match between two robot teams. One of the computer is for Gazebo visualization with model plugins to simulate the motions of each robot. The other computers are used for running the real robot codes and their corresponding Coach programs. The total computation involved has been distributed to three computers and therefore, the simulation speed is fast enough to test the multi-robot coordination strategies in real time. In addition, there is only one ROS master in computer A, which registers nodes, services, topics and other ROS resources from all the three computers. Finally, the simulation of a match (without goalie) is shown in Fig. 23.

6 Single Robot Simulation Tutorial

Note that the `single_nubot_gazebo` package can simulate only ONE robot soccer player for RoboCup MSL. It is designed for demonstration of how the simulation system works. However, it can be adapted for other purposes. If you want to test multi-robot cooperation strategies, please refer to the `gazebo_visual` package, while the compilation in this tutorial is still useful. For further information, please refer to our previous paper [14].

6.1 Get the Package

If you have git installed, you could use the below command to download the package:

```
$ git clone git@github.com:nubot-nudt/gazebo_visual.git
```

As an alternation, you could also go to https://github.com/nubot-nudt/single_nubot_gazebo and download the package in zip format and extract it in your computer.

6.2 Environment Configuration

The recommended Operating Environment is Ubuntu 14.04 and ROS Jade with Gazebo included. For more operating environment, please refer to the readme file at https://github.com/nubot-nudt/single_nubot_gazebo.

ROS Jade has `gazebo_ros_pkgs` with it, so you don't have to install the package again. However, the following steps should be done to fix a bug in ROS Jade related to Gazebo:

```
$ sudo gedit /opt/ros/jade/lib/gazebo_ros/gazebo
```

In this file, go to line 24 and delete the last '/', i.e.,

```
setup_path=$(pkg-config --variable=prefix gazebo)/share/gazebo/
```

is replaced with

```
setup_path=$(pkg-config --variable=prefix gazebo)/share/gazebo
```

After these steps, try to run the command below to check if it is successful.

```
$ rosrun gazebo_ros gazebo
```

or

```
$ roslaunch gazebo_ros empty_world.launch
```

If either one is successful running, then you are ready for the following steps.

6.3 Package Compiling

(1) Go to the package root directory (`single_nubot_gazebo`), e.g.

```
$ cd ~/single_nubot_gazebo
```

(2) If you already have `CMakeLists.txt` in the `src` folder, then you can skip this step. If not, execute the commands below:

```
$ cd src
$ catkin_init_workspace
$ cd ..
```

(3) Configure the package using the command below. In this step, you may encounter some errors related to Git. However, if you did not use Git, just ignore them.

```
$ ./configure
```

(4) Compiling the package, the simulation system is ready if the compiling is completed done.

```
$ catkin_make
```

6.4 Package Overview

The robot movement is realized by a Gazebo model plugin called NubotGazebo generated by source files nubot_gazebo.cc and nubot_gazebo.hh. Most importantly, the essential part of the plugin is realizing three motions: omnidirectional locomotion, ball-dribbling and ball-kicking.

Basically, this plugin subscribes to topic /nubotcontrol/velcmd for omnidirectional movement, and subscribes to services /BallHandle and /Shoot for ball-dribbling and ball-kicking, respectively. You can customize this code for your robot based on these messages and services as a convenient interface. The types and definitions of the topics and services are listed in Table 3.

For the definition of /BallHandle service, when enable equals to a non-zero number, a dribble request would be sent. If the robot meets the conditions to dribble the ball, the service response BallIsHolding is true. For the definition of /Shoot service, when ShootPos equals to -1, this is a ground pass. In this case, strength is the initial speed you would like the soccer ball to have. When ShootPos equals to 1, this is a lob shot. In this case, strength is useless since the strength is calculated by the Gazebo plugin automatically and the soccer ball would follow a parabola path to enter the goal area(only if the robot heads towards

Table 3 Topics and services

Topic/Service	Type	Definition
/nubotcontrol/velcmd	nubot_common/VelCmd	<pre>float32 Vx float32 Vy float32 w</pre>
/BallHandle	nubot_common/BallHandle	<pre>int64 enable --- int64 BallIsHolding</pre>
/Shoot	nubot_common/Shoot	<pre>int64 strength int64 ShootPos --- int64 ShootIsDone</pre>

the goal area). If the robot successfully kicks the ball even if it failed to goal, the service response `ShootIsDone` is true.

There are three ways for a robot to dribble a ball, e.g.,

- (a) Setting ball pose continually: this is the most accurate one; nubot would hardly lose control of the ball, but the visual effect is not very good (the ball does not rotate).
- (b) Setting ball secant velocity: this is less accurate than method (a) but more accurate than method (c).
- (c) Setting ball tangential velocity: this is the least accurate. If the robot moves fast, such as 3 m/s, it would probably lose control of the ball. However, this method achieves the best visual effect under low-speed condition.

For package `single_nubot_gazebo`, it uses method (c) for better visualization effect. However, for package `nubot_gazbeo`, it uses method (a) for better control of the soccer ball.

6.5 Single Robot Automatic Movement

The robot will do motions according to the state transfer graph shown in Fig. 17, following the below steps:

- (1) Go to the package root directory.
- (2) source the `setup.bash` file:

```
$ source devel/setup.bash
```

- (3) Using `roslaunch` to load the simulation world

```
$ roslaunch nubot_gazebo sdf_nubot.launch
```

Note: Step 2 should be performed every time to open a new terminal. Alternatively, this command can be wrote into the `~/.bashrc` file so that step 2 is not required when opening new terminal.

Finally, the robot will rotate and translate with a given trajectory, i.e., it accelerates at a constant acceleration and stays at a constant speed after reaching the maximum velocity.

You could click the `Edit->Reset World` from the menu (or press `ctrl-shift-r`) to reset the simulation world so the robot would do the basic motions again.

When the robot reaches its final state (HOME), its motion can be controlled using keyboard under `$ rosrun nubot_gazebo nubot_teleop_keyboard`. You could also run `$ rqt_graph` to see the data flow chart of messages/topics.

6.6 *NuBotGazebo API*

For the detailed information and usage of the `NuBotGazebo` class, please refer to the `doc/` folder.

6.7 *How You Could Use It to Do More Stuff*

The main purpose of the simulation system is to test multi-robot collaboration algorithms. As a precondition, the users have to know how to control the movement of each robot in the simulation. The topic publishing and service calling could be inferred by reading the source of keyboard controlling. In a word, to control the movement of the robots requires publishing velocity commands on the topic `/nubotcontrol/velcmd`. If the robot is close enough to the ball, dribble the ball by calling the ROS service named `/BallHandle` and kick the ball by calling the service named `/Shoot`. The types and definitions of these topics and services are presented in Table 3.

7 Multi Robot Simulation Tutorial

7.1 *Package Overview*

The following three packages should be used together to simulate multi-robots together. The `nubot_ws` and the `coach4sim` package can be downloaded at https://github.com/nubot-nudt/nubot_ws and <https://github.com/nubot-nudt/coach4sim> respectively.

package	description
<code>gazebo_visual</code>	For robot simulation and visualization
<code>nubot_ws</code>	For robot controlling
<code>coach4sim</code>	Game command sending

Qt has to be installed in order to use `coach4sim`. However, for those who do not want to install Qt, a solution is to use ROS command line tools for sending game commands:

```
$ rostopic pub -r 1 /nubot/receive_from_coach
nubot_common/CoachInfo "
MatchMode: 10
MatchType: 0 "
```

In the command, MatchMode is the current game command, MatchType is the previous game command. The coding of the game commands is in core.hpp. For quick reference:

```
enum MatchMode {
    STOPROBOT = 0,
    OUR_KICKOFF = 1,
    OPP_KICKOFF = 2,
    OUR_THROWIN = 3,
    OPP_THROWIN = 4,
    OUR_PENALTY = 5,
    OPP_PENALTY = 6,
    OUR_GOALKICK = 7,
    OPP_GOALKICK = 8,
    OUR_CORNERKICK = 9,
    OPP_CORNERKICK = 10,
    OUR_FREEKICK = 11,
    OPP_FREEKICK = 12,
    DROPBALL = 13,
    STARTROBOT = 15,
    PARKINGROBOT = 25,
    TEST = 27
};
```

The robot movement is realized by a Gazebo model plugin which is called NubotGazebo generated by source files nubot_gazebo.cc and nubot_gazebo.hh. Basically the essential part of the plugin is realizing basic motions: omni-directional locomotion, ball-dribbling and ball-kicking.

The plugin single_nubot_gazebo is similar to that in package single_nubot_gazebo, i.e., it subscribes to the topic nubotcontrol/velcmd for omnidirectional movement, and subscribes to the service BallHandle and Shoot for ball-dribbling and ball-kicking, respectively. For package gazebo_visual, there is a new topic named omnivision/OmniVisionInfo which contains messages about the soccer ball and all the robots' information such as position, velocity and etc. Since there may be multiple robots, the name of those topics and services should be prefixed with the robot model names in order to be distinguished with each other. For example, if a robot model's name is nubot1, then the topic names are /nubot1/nubotcontrol/velcmd and /nubot1/omnivision/OmniVisionInfo and the service names would be /nubot1/BallHandle and /nubot1/Shoot accordingly. The types and definitions of the topic nubot1/omnivision/OmniVisionInfo is as:

```
Header header
BallInfo ballinfo
ObstaclesInfo obstacleinfo
RobotInfo[] robotinfo
```

As shown above, there are three new message types in the definition of the omnivision/OmniVisionInfo topic, i.e., BallInfo, ObstaclesInfo and RobotInfo. The field robotinfo is a vector. Before introducing the format

of these messages, three other underlying message types Point2d, Ppoint and Angle are listed below.

```

# Point2d.msg, representing a 2-D point.
float32 x                      # x component
float32 y                      # y component

# PPoint.msg, representing a 2-D point in polar coordinates.
float32 angle                  # angle against polar axis
float32 radius                 # distance from the origin

# Angle.msg, representing the angle
float32 theta                  # angle of rotation

# BallInfo.msg, representing the information about the ball
Header header                  # ROS header defined in std_msgs
int32 ballinfostate            # the state of the ball information
Point2d pos                     # position in the global reference
                                # frame
PPoint real_pos                # relative position in the robot
                                # body frame
Point2d velocity               # velocity in the global reference
                                # frame
bool pos_known                 # ball position is known(1) or not(0)
bool velocity_known            # ball velocity is known(1) or not(0)

# ObstaclesInfo.msg, representing the obstacles information
Header header                  # ROS header defined in std_msgs
Point2d[] pos                   # position in the global reference
                                # frame
PPoint[] polar_pos              # position in the polar frame, whose
                                # origin is the center of the robot
                                # and the polar axis
                                # is along the kicking mechanism

# RobotInfo.msg, representing teammates' information
Header header                  # ROS header defined in std_msgs
int32 AgentID                  # ID of the robot
int32 targetNum1                # robot ID to be assigned for target
                                # position 1
int32 targetNum2                # robot ID to be assigned for target
                                # position 2
int32 targetNum3                # robot ID to be assigned for target
                                # position 3
int32 targetNum4                # robot ID to be assigned for target
                                # position 4
int32 staticpassNum             # in static pass, the passer's ID
int32 staticcatchNum            # in static pass, the catcher's ID
Point2d pos                     # robot position in global coordinate
                                # system
Angle heading                  # robot heading in global coordinate
                                # system
float32 vrot                    # rotational velocity in the global
                                # coordinate system
Point2d vtrans                  # linear velocity in the global

```

```

        # coordinate system
bool iskick          # robot kicks the ball(1) or not(0)
bool isvalid          # robot is valid(1) or not(0)
bool isstuck          # robot is stuck(1) or not(0)
bool isdribble        # robot dribbles the ball(1) or not(0)
char current_role    # the current role
float32 role_time   # time duration that the robot keeps
                     # the role unchanged
Point2d target       # target position

```

7.2 Configuration of Computer A and Computer B

The recommended way to run simulation is with two computers to run `nubot_ws` and `gazebo_visual` separately, i.e., computer A runs `gazebo_visual` to display the movement of robots, while computer B runs `nubot_ws` to control the virtual robots. In addition, computer B should also run the `coach` program for sending game commands. Communication between computer A and computer B is via ROS topics and services.

Following is an configuration example:

- Adding each other's IP address in the `/etc/hosts` file;
- Run `gazebo_visual` in computer A;
- In computer B, export `ROS_MASTER_URI` and then run `nubot_ws`;
- In computer B, run the `coach` and send game command.

8 Conclusion

In summary, we have presented the ROS based software and Gazebo based simulation for our RoboCup MSL robots. ROS based software makes it easier to share data and code among RoboCup MSL teams, and construct hybrid teams. Further, we have also detailed the design of the interface between the robot software and simulation, which brings the possibility to evaluate multi-robot collaboration algorithms using the simulation.

We expect this work to be of value in the RoboCup MSL community. On the one hand, the researchers can refer to our method to design both software and simulation for RoboCup MSL robots, or even general robots. On the other hand, the NuBot simulation software can be used to simulate RoboCup MSL matches, which enables the state-of-the-art machine learning algorithms to be used for multi-robot collaboration training.

Lastly, the presented ROS based software and Gazebo based simulation can also be employed for multi-robot collaboration researches more than RoboCup with little modification.

Acknowledgements Our work is supported by National Science Foundation of China (NO. 61403409 and NO. 61503401), China Postdoctoral Science Foundation (NO. 2014M562648), and graduate school of National University of Defense Technology. All members of the NuBot research group are gratefully acknowledged.

References

1. Kitano, H., M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa. 1997. Robocup: The robot world cup initiative. In *Proceedings of the first international conference on Autonomous agents*, 340–347. ACM.
2. Kitano, H., M. Asada, Y. Kuniyoshi, I. Noda, E. Osawa, and H. Matsubara. 1997. Robocup: A challenge problem for AI. *AI Magazine* 18 (1): 73.
3. Kitano, H., M. Asada, I. Noda, and H. Matsubara. 1998. Robocup: robot world cup. *IEEE Robotics Automation Magazine* 5: 30–36.
4. Almeida, L., J. Ji, G. Steinbauer, and S. Luke. 2016. *RoboCup 2015: Robot World Cup XIX*, vol. 9513. Heidelberg: Springer.
5. Bianchi, R.A., H.L. Akin, S. Ramamoorthy, and K. Sugiura. 2015. *RoboCup 2014: Robot World Cup XVIII*, vol. 8992. Heidelberg: Springer.
6. Soetens, R., R. van de Molengraft, and B. Cunha. 2014. Robocup msl-history, accomplishments, current status and challenges ahead. In *RoboCup 2014: Robot World Cup XVIII*, ed. R.A.C. Bianchi, H.L. Akin, S. Ramamoorthy, and K. Sugiura, 624–635. Heidelberg: Springer.
7. Rohmer, E., S.P.N. Singh, and M. Freese. 2013. V-rep: A versatile and scalable robot simulation framework. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1321–1326.
8. Koenig, N., and A. Howard. 2004. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings*, vol. 3, 2149–2154. IEEE.
9. Michel, O. 1998. Webots: Symbiosis between virtual and real mobile robots. In *the First International Conference on Virtual Worlds*, (London, UK), 254–263. Springer.
10. Der, R., and G. Martius. 2012. The LpzRobots Simulator. In *The Playful Machine Ralf*, ed. R. Der, and G. Martius. Heidelberg: Springer.
11. Harris, A., and J.M. Conrad. 2011. Survey of popular robotics simulators, frameworks, and toolkits. In *2011 Proceedings of IEEE Southeastcon*, 243–249.
12. Castillo-Pizarro, P., T.V. Arredondo, and M. Torres-Torriti. 2010. Introductory survey to open-source mobile robot simulation software. In *Robotics Symposium and Intelligent Robotic Meeting (LARS), 2010 Latin American*, 150–155.
13. Xiong, D., J. Xiao, H. Lu, Z. Zeng, Q. Yu, K. Huang, X. Yi, Z. Zheng, C. Loughlin, and C. Loughlin. 2016. The design of an intelligent soccer-playing robot. *Industrial Robot: An International Journal* 43 (1).
14. Yao, W., W. Dai, J. Xiao, H. Lu, and Z. Zheng. 2015. A simulation system based on ros and gazebo for robocup middle size league. In *2015 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 54–59. IEEE.
15. Van De Molengraft, M., and O. Zweigle. 2011. Advances in intelligent robot design for the robocup middle size league. *Mechatronics* 21 (2): 365.
16. Nadarajah, S., and K. Sundaraj. 2013. A survey on team strategies in robot soccer: team strategies and role description. *Artificial Intelligence Review* 40 (3): 271–304.
17. Nadarajah, S., and K. Sundaraj. 2013. Vision in robot soccer: a review. *Artificial Intelligence Review* 1–23.
18. Li, X., H. Lu, D. Xiong, H. Zhang, and Z. Zheng. 2013. A survey on visual perception for RoboCup MSL soccer robots. *International Journal of Advanced Robotic Systems* 10 (110).

19. Nardi, D., I. Noda, F. Ribeiro, P. Stone, O. von Stryk, and M. Veloso. 2014. Robocup soccer leagues. *AI Magazine* 35 (3): 77–85.
20. Lunenburg, J., R. Soetens, F. Schoenmakers, P. Metsemakers, R. van de Molengraft, and M. Steinbuch. 2013. Sharing open hardware through rob, the robotic open platform. In *Proceedings of 17th annual RoboCup International Symposium*.
21. Neves, A.J., A.J. Pinho, A. Pereira, B. Cunha, D.A. Martins, F. Santos, G. Corrente, J. Rodrigues, J. Silva, J.L. Azevedo, et al. 2010. *CAMBADA soccer team: from robot architecture to multiagent coordination*. INTECH Open Access Publisher.
22. Santos, F., L. Almeida, P. Pedreiras, and L.S. Lopes. 2009. A real-time distributed software infrastructure for cooperating mobile autonomous robots. In *International Conference on Advanced Robotics, 2009. ICAR 2009*, 1–6. IEEE.
23. Santos, F., L. Almeida, and L.S. Lopes. 2008. Self-configuration of an adaptive TDMA wireless communication protocol for teams of mobile robots. In *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008*, 1197–1204. IEEE.
24. Lu, H., S. Yang, H. Zhang, and Z. Zheng. 2011. A robust omnidirectional vision sensor for soccer robots. *Mechatronics* 21 (2): 373–389.
25. Lu, H., H. Zhang, J. Xiao, F. Liu, and Z. Zheng. 2008. Arbitrary ball recognition based on omni-directional vision for soccer robots. In *RoboCup 2008: Robot Soccer World Cup XII*, ed. L. Iocchi, H. Matsubara, A. Weitzenfeld, and C. Zhou, 133–144. Heidelberg: Springer.
26. Zeng, Z., H. Lu, and Z. Zheng. 2013. High-speed trajectory tracking based on model predictive control for omni-directional mobile robots. In *2013 25th Chinese Control and Decision Conference (CCDC)*, 3179–3184. IEEE.
27. Xiao, J., H. Lu, Z. Zeng, D. Xiong, Q. Yu, K. Huang, S. Cheng, X. Yang, W. Dai, J. Ren, et al. 2015. Nubot team description paper 2015. In *Proceedings of RoboCup 2015, Hefei, China*.
28. Rajaie, H., O. Zweigle, K. Häussermann, U.-P. Käppeler, A. Tamke, and P. Levi. 2011. Hardware design and distributed embedded control architecture of a mobile soccer robot. *Mechatronics* 21 (2): 455–468.
29. Zandsteeg, C. 2005. Design of a robocup shooting mechanism. University of Technology Eindhoven.
30. Martinez, C.L., F. Schoenmakers, G. Naus, K. Meessen, Y. Douven, H. van de Loo, D. Bruijnen, W. Aangenent, J. Groenen, B. van Ninhuijs, et al. 2014. Tech united eindhoven, winner robocup 2014 msl. In *Robot Soccer World Cup*, 60–69. Springer.
31. Jansen, D., and H. Buttner. 2004. Real-time ethernet: the ethercat solution. *Computing and Control Engineering* 15 (1): 16–21.
32. Prytz, G. 2008. A performance analysis of EtherCAT and PROFINET IRT. In *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008*, 408–415. IEEE.
33. Xiong, D., H. Lu, and Z. Zheng. 2012. A self-localization method based on omnidirectional vision and mti for soccer robots. In *2012 10th World Congress on Intelligent Control and Automation (WCICA)*, 3731–3736. IEEE.
34. Rusu, R.B., and S. Cousins. 2011. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, (Shanghai, China), 9–13 May 2011.
35. Schnabel, R., R. Wahl, and R. Klein. 2007. Efficient ransac for point-cloud shape detection. *Computer Graphics Forum* 26 (2): 214–226.
36. Lu, H., Q. Yu, D. Xiong, J. Xiao, and Z. Zheng. 2014. Object motion estimation based on hybrid vision for soccer robots in 3d space. In *Proceedings of RoboCup Symposium 2014*, (Joao Pessoa, Brazil).
37. Wang, X., H. Zhang, H. Lu, and Z. Zheng. 2010. A new triple-based multi-robot system architecture and application in soccer robots. In *Intelligent Robotics and Applications*, ed. H. Liu, H. Ding, Z. Xiong, and X. Zhu, 105–115. Heidelberg: Springer.
38. Cheng, S., J. Xiao, and H. Lu. 2014. Real-time obstacle avoidance using subtargets and Cubic B-spline for mobile robots. In *Proceedings of the IEEE International Conference on Information and Automation (ICIA 2014)*, 634–639. IEEE.

Author Biographies

Junhao Xiao (M'12) received his Bachelor of Engineering (2007) from National University of Defense Technology (NUDT), Changsha, China, and his Ph.D. (2013) at the Institute of Technical Aspects of Multimodal Systems (TAMS), Department Informatics, University of Hamburg, Hamburg, Germany. Then he joined the Department of Automatic Control, NUDT (2013) where he is an assistant professor on Robotics and Cybernetics. The focus of his research lies on mobile robotics, especially on RoboCup Soccer robots, RoboCup Rescue robots, localization, and mapping.

Dan Xiong is a Ph.D. student at National University of Defense Technology (NUDT). He received his Bachelor of Engineering and Master of Engineering both from NUDT in 2010 and 2013, respectively. His research focuses on image processing and RoboCup soccer robots.

Weijia Yao is a Master student at National University of Defense Technology (NUDT). He received his Bachelor of Engineering from NUDT in 2015. He currently focuses on multi-robot coordination and collaboration.

Qinghua Yu is a Ph.D. student at National University of Defense Technology (NUDT). He received his Bachelor of Engineering and Master of Engineering both from NUDT in 2011 and 2014, respectively. His research focuses on robot vision and RoboCup soccer robots.

Huimin Lu received his Bachelor of Engineering (2003), Master of Engineering (2006) and Ph.D. (2010) from National University of Defense Technology (NUDT), Changsha, China. Then he joined the Department of Automatic Control, NUDT (2010) where he is an associate professor on Robotics and Cybernetics. The focus of his research lies on mobile robotics, especially on RoboCup Soccer robots, RoboCup Rescue robots, omni-directional vision, and visual SLAM.

Zhiqiang Zheng received his Ph.D. (1994) from University of Liege, Liege, Belgium. Then he joined the Department of Automatic Control, National University of Defense Technology where he is a full professor on Robotics and Cybernetics. The focus of his research lies on mobile robotics, especially on multi-robot coordination and collaboration.

VIKI—More Than a GUI for ROS

**Robin Hoogervorst, Cees Trouwborst, Alex Kamphuis
and Matteo Fumagalli**

Abstract This chapter introduces the open-source software VIKI. VIKI is a software package that eases the configuration of complex robotic systems and behavior by providing an easy way to collect existing ROS packages and nodes into modules that provide coherent functionalities. This abstraction layer allows users to develop behaviors in the form of a collection of interconnected modules. A GUI allows the user to develop ROS-based software architectures by simple drag-and-drop of VIKI modules, thus providing a visual overview of the setup as well as ease of reconfiguration. When a setup has been created, VIKI generates a roslaunch file by using the information of this configuration, as well as the information from the module definitions, which is then launched automatically. Distributed capabilities are also guaranteed as VIKI enables the explicit configuration of roslaunch features in its interface. In order to show the potential of VIKI, the chapter is organised in the form of a tutorial which provides a technical overview of the software, installation instructions as well as three use-cases with increased difficulty. VIKI functions alongside your ROS installation, and only uses ROS as a runtime dependency.

Keywords ROS · GUI · Abstraction layer · Modularity · Educational · Software architecture

R. Hoogervorst · C. Trouwborst · A. Kamphuis
University of Twente, Drienerlolaan 5, 7522 NB Enschede, Netherlands
e-mail: r.w.p.hoogervorst@student.utwente.nl

C. Trouwborst
e-mail: ceestrouwborst@gmail.com

A. Kamphuis
e-mail: kamphuis.alex@gmail.com

M. Fumagalli (✉)
Aalborg University, A.C. Meyers Vænge 15, 2450 Copenhagen, Denmark
e-mail: M_fumagalli@m-tech.aau.dk

1 Introduction

One of the major advantages of using the Robot Operating System (ROS) [1] is the possibility to exploit the modular features that it provides, thus allowing the user to develop packages that are easy to install and highly reusable. These functionalities allow the new generation of robot developers to build complex distributed systems based on open source ROS packages in an easy and reliable manner. However, even though single packages are usually easily installed and used, the development of complex systems that use multiple existing packages remains a task that requires experience and prior knowledge of the tools necessary to properly set up and configure the environment. This means that novice developers need to first grasp different concepts, such as nodes, topics, topic types, and many more, before being able to properly configure their system. Even for experienced users, it is often hard to reuse packages efficiently. Reusing a package in a different experiment or setup requires adding references to the package in a ROS launch file, adding lines for a large variety of settings available to the packages used, while keeping overview of the different communication channels used. It is very easy to lose a clear perspective through several development iterations. To minimise this problem, different tools have been developed and are used by ROS users for configuration and debugging as well as for monitoring. These tools typically allow the visualization of the running setup. This is helpful for the developers and users developing and testing combinations of different ROS packages, but may yield complicated graphs. Additionally, these tools do not provide the ability to make changes inside their graphical representation.

In order to reduce the complexity of creating and configuring large runtime systems with ROS, while still providing the functionality of reusability and modularity, a new software package, namely VIKI (see Fig. 1), is proposed and presented in

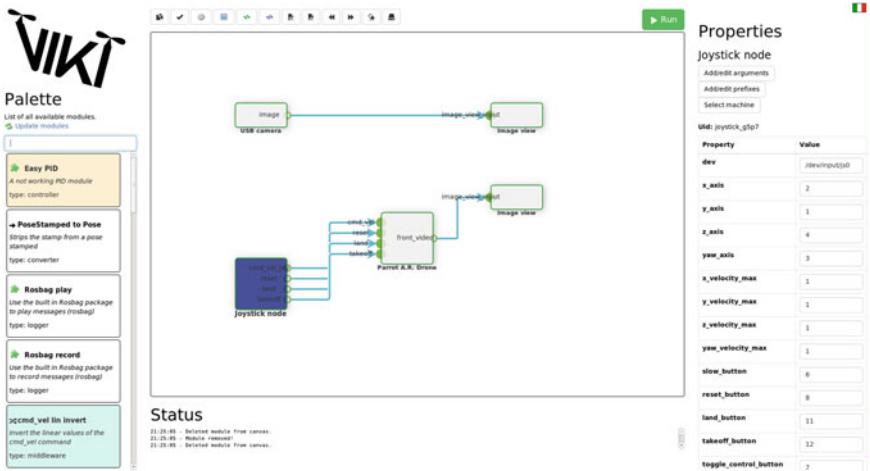


Fig. 1 A screenshot of VIKI



Fig. 2 The intention of the abstraction layers within VIKI is to build a configuration using modules, while ROS is being used for runtime. The gray sections are within the abstraction layer of VIKI, the white ones are within ROS

this chapter. VIKI aims at improving the ease of configuring the environment, thus minimising the problems of complexity and loss of overview for the user. In order to do this, VIKI adds a *layer of abstraction* which presents simplified information to the end-users and developers and a *Graphical User Interface* (GUI) that allows clear and intuitive visualization of the interconnection and communication throughout the setup. By visualizing the modules, rather than all nodes, the visual overview can be greatly simplified, while still viewing the information that is important. More precisely, the abstraction layer within VIKI utilizes meta-information about packages, and allows for the combination of (parts of) packages. This is implemented in *modules* and can be visualized as seen in Fig. 2. The modules represent building blocks with coherent functionality that can be combined more easily to ultimately define the overall system behaviour. While VIKI abstracts the ROS packages inside the modules, the running environment is still entirely based on these packages, keeping the entire set of ROS functionalities unaltered. As a consequence, the user can reason about the packages on a higher level of abstraction, allowing the user to focus on the system architecture while using VIKI. In order to allow an easy and intuitive use of its modules, VIKI employs extra metadata about the inputs, outputs and the packages in order to provide visual connection of modules.

Besides that, VIKI provides a Graphical User Interface that allows to arrange modules by *dragging and dropping*. This creates a visual overview of the setup, which is easily adaptable for implementing and testing new control schemes. VIKI gives the ability to adapt the arrangements easily and run them again. Because the details of the package implementation are already dealt with in the abstraction, the user can use this GUI also to reason at a higher level about the software.

Since the module description files provide extra information about the nodes, VIKI can use this information to aid the user. Types of topics are specified inside these module files and this prohibits the user from connecting wrong topic types to each other. Besides that, the GUI can provide an instant overview of the inputs and outputs that a module exposes, without consulting further documentation. Especially for the starting ROS developer, this can help avoid confusion.

VIKI is opensource and released under an MIT license. The full code repository can be found at <https://www.github.com/UT-RAM/viki>. Full documentation can be found at <http://viki.readthedocs.io>.

The rest of this chapter is structured as follows. Section 2 gives an overview of the existing available applications that provide similar functionalities of VIKI. Sections 3 and 4 aid the reader in setting up VIKI and running some examples, respectively. Section 5 finally provides a more detailed technical overview of the internals of VIKI.

2 Background

2.1 *Existing Software*

Using ROS might involve using many different tools aiding in running an environment such as editors, compilers, make systems and package specific tools. Many of these tools are console applications due to the fact that console applications are in general easier to create and are sufficient to provide the experienced user with enough power. Inexperienced users however may be daunted by console based applications and therefore require a more visual experience.

Several existing ROS tools and packages provide such a visual experience for very specific cases. An interesting and fairly complete example is rxDeveloper [2]. This software aims at a visual interface for building ROS launch files. Although the package provides a very promising list of features, including the generation of template files for both C++ and Python, it appears to have not received the attention it deserved. The project was last updated over 4 years ago for a currently deprecated ROS version. Using unsupported software is not recommended as it usually leads to unmaintainable setups for the research itself. Furthermore, for new ROS user it can still be challenging to use and start with, since rxDeveloper relies on roslaunch files and lacks tools for easy configuration of these packages.

Another package that comes close to the functionality of VIKI is BRIDE [3]. BRIDE looks similar to rxDeveloper, in the sense that it visualizes ROS nodes and makes it easy to connect them. The tool seems to aim at a workflow that lets the user design visually, and generate package code based on this visual design. Similar to rxDeveloper, BRIDE does not benefit from active development or an active community. A major difference with respect to VIKI is that BRIDE places emphasis on making it easier to create your own packages, while VIKI places emphasis on reuse of already available modules and integration. In fact, BRIDE allows the generation of ROS packages based on the design, while VIKI generates visual design based on the available modules.

Another package worth mentioning is FKIE Node Manager [4], which is intended to be a GUI for managing ROS nodes, topics, services, parameters and launch files present on multiple systems. It provides the user with a clear overview of running nodes and can support the user in the design phase with its launch file editor. This text editor has syntax highlighting and allows you to insert lines from templates. This is the first big difference with VIKI, that allows connecting ROS packages through a Drag-and-Drop workflow. Another difference is that VIKI allows for an additional

layer of abstraction through the use of VIKI modules. In short: where FKIE node manager can be used to get information on all the parts in a running system, VIKI excels in designing system architecture by focusing on the connections between functional blocks.

Besides these complete packages, there is a big number of visual tools available for ROS. A few examples are presented in [5] and others can be found on ROS wiki pages and scattered around the web [6, 7]. Most of the examples are robot specific and therefore offer no advantage outside the use of that specific robot except possibly for the reuse of code and structure. More general tools are often built in the ROS GUI [8, 9] and focus on the visualization or control of a robot at runtime. It is worth to note that for very specific use cases interesting packages are available, such as the ROS GUI for Matlab (proprietary software) [10] that provides a way of connecting to the ROS master through a Matlab GUI, and Linkbot Labs (proprietary software) which uses small linkable robots to teach students how to program [11].

Many of these tools provide functionalities that are compatible with VIKI. This can be of great advantage when the user wants to use specific tools for different utilities. One could, for example, load the roslaunch file that VIKI generates into FKIE node manager, and use FKIE's functionality for a runtime lower level overview and manage launching nodes using that. This allows the user to design his or her environment using VIKI on a high abstraction level, while using a low abstraction level during runtime. This low-level overview could be useful for, for example, specific debugging cases. Similarly, rqt can provide visualizations on a low-level scale, while VIKI is used to design the top-level functionality. The user is free to switch between these environments, and using VIKI does not prohibit using the lower level tools at runtime.

3 ROS Environment Configuration

VIKI is a standalone application alongside the installation of ROS. The prerequisites are that ROS is installed and a catkin workspace is available. Furthermore, it is assumed that *git* and *python 2* are installed. Further dependencies are installed automatically by the provided self-configuring tool, making VIKI easy to set up.¹

Further configuration depends on what modules are loaded into VIKI. As stated earlier, a module uses (a set of) ROS packages. Some ROS packages require additional configuration, which is needed for VIKI modules that use these packages as well. VIKI ensures that the modules provided in the core are either automatically set up, or easy to set up by following the documentation. For more information visit <http://wiki.readthedocs.io> or follow the steps below.

The installation of VIKI is a process of two steps²:

¹This configuration is based on version *0.2-Alice*, released on 9 May 2016.

²Installation instructions might change in future releases. For the most recent installation instructions on VIKI, read the instructions located in the github repository.

1. Clone the repository located at <https://www.github.com/UT-RAM/viki> into a dedicated directory. The authors suggest to install VIKI inside the home folder, although the user is free to choose any other preferred location.
2. Navigate to the installation folder in the command line and run:

```
./viki configure
```

This command starts a program that guide the user through the installation of VIKI. The user will be asked to provide relevant information necessary for the proper installation of VIKI, such as the installed ROS version and the location of the catkin workspace. When in doubt regarding the entries, use the default value.

After the installation is completed, a desktop entry will be added such that it is possible to launch VIKI using the Unity Dash. The options provided by the user at the moment of the installation can be post-edited by editing *config.json*. The user is suggested to visit the documentation [12] for further info and troubleshooting.

After completing the installation of VIKI, modules need to be added. By running *./viki add-module-repository core*, VIKI will install the core module repository inside the root directory specified during installation. This is the location where future VIKI modules will be added. When the command *add-module-repository core* is ran, modules are installed by pulling the git repository *github.com/UT-RAM/viki-modules* into the aforementioned directory. At completion of this step, VIKI will be able to automatically find the modules and use the module files that are available in there.

Verification of the installation is done by launching VIKI, as explained in the next section. If any problem is encountered during installation, usually a quick search on google may solve these issues. If you are running into any specific issues with VIKI, do not hesitate to contact the developers on github or create an issue at the github repository.

4 Testdriving VIKI

This section is a hands-on tutorial that will allow the reader to get acquainted with VIKI and its functionalities by describing three test cases with increasing difficulty. The first tutorial is based on the use of the well-known turtlesim. This is a 2D simulation for a turtle robot. Here, the reader is instructed on how to launch and play with the turtle from within VIKI. After that, a more realistic system is set-up using VIKI, and the user is guided through launching an UAV (the Parrot A.R. Drone is used as UAV) and controlling this with a controller or joystick. This specific example is of course difficult to carry out without the proper hardware. However, it proves how VIKI can handle this sort of task very well. The last section will guide the reader in setting up networking capabilities for ROS by exploiting the embedded functionalities of VIKI and its modules, which allow intuitive and easy configuration of the networked software architecture. In this third tutorial, the same

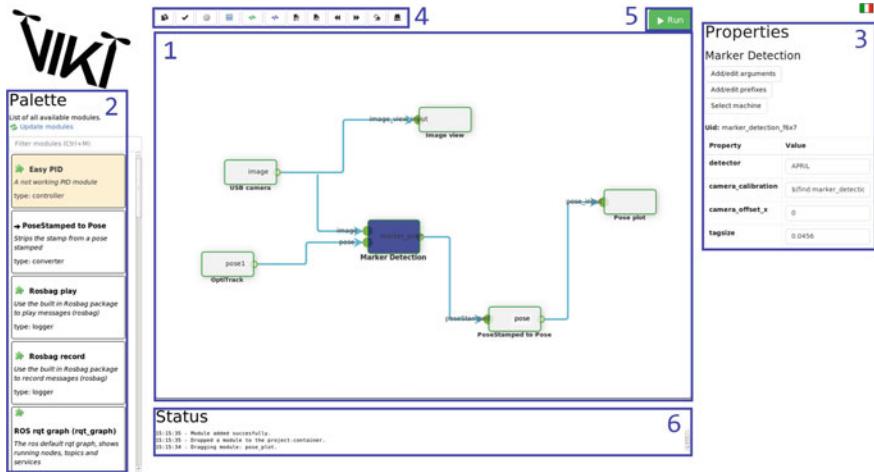


Fig. 3 Screenshot of VIKI’s interface. 1 The canvas: a container where you can build your project. 2 Module Palette: list of all available modules with descriptions. 3 Module specifics: it shows information on the currently selected module. 4 Toolbar: buttons with program specific actions such as save and open. 5 Run button: builds and launches your project. 6 Status pane: it shows debug information on internal actions performed by VIKI

UAV is launched from one computer, while the controls are launched from another device.

4.1 Turtlesim

This first tutorial is meant to guide the reader through launching the first project with VIKI using the turtlesim. As a first step, VIKI has to be started. There are two methods to do this:

- in a terminal, navigate to the installation folder of viki (by default /home/[user]/viki) and execute:

```
./viki run
```

- This launches the graphical interface of VIKI, which is illustrated in Fig. 3.
- launch VIKI from the Unity Dash, which is commonly used in Ubuntu to launch application. During the installation of VIKI an entry is created in Unity Dash. By clicking the dash icon in the upper-left corner of Ubuntu, it is possible to search for VIKI. Note that Ubuntu may not refresh its dash immediately. Therefore, if the VIKI icon is not present in the dash after installation, it may be required to log off first and or to reboot.

Fig. 4 The palette inside the interface of VIKI after entering ‘turtle’ in the search bar

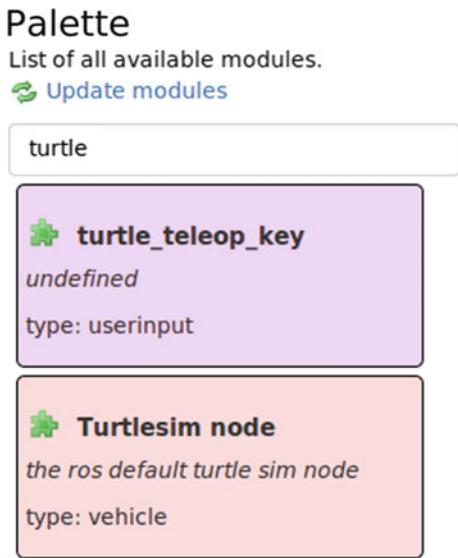


Fig. 5 The canvas in VIKI after adding two loose modules to it



For this tutorial, two modules are needed, namely the *turtle simulator* and the module that interprets the keyboard input and sends it to the simulator. In order to find an existing module, it is possible to make a search inside the palette (indicated by number 2 in Fig. 3) by clicking in the textbox and entering e.g. “turtle”. By doing this, two modules will be displayed, as shown in Fig. 4, namely:

- *turtle_teleop_key*
- *turtlesim node*

Now these modules can be clicked and dragged on to the canvas (indicated by 1 in Fig. 3). The result should look similar to Fig. 5.

Note that the two modules are still loose, meaning that no connection has yet been made among the modules. In order to connect the modules, it is possible to drag the output of one module, to the input of another. In the specific case addressed by this tutorial, it is necessary to connect the output of *turtle_teleop_key* to the input of *turtlesim node*, which can be done by dragging the teleop’s output node to the turtle’s input node. An arrow will appear that indicates the direction of information. It must be noted that it is not possible to start dragging from the turtle’s input node, as VIKI is constructed in a way that the direction of information should be followed.

After completing these steps, the setup is now ready. The user can now hit the green *run* button on top of the screen, indicated by number 5 in Fig. 3. This will open a new terminal providing some text feedback to the user, regarding the status of the setup. More importantly in this tutorial, a window with a turtle in it should also appear. By clicking at any point of the text window, it becomes possible to use the arrow keys to control the turtle in the other window.

In order to close the *turtlesim* application, it is possible to select the terminal window and press Ctrl + C, which will kill its processes. After gracefully shutting down, the terminal window will disappear, and the user is free to run again the canvas setup.

It is important to point out that after pressing the launch button, VIKI will actually launch a separate process in a new terminal window. In case something goes wrong during launch, it is by default impossible to see what actually went wrong, since the terminal closes automatically, due to the process finishing. This is due to the default setting in Ubuntu. In order to avoid this, thus giving the user the possibility to check the output of the processes that are displayed in the VIKI terminal, the following procedure needs to be completed:

1. Open a terminal window
2. In the menubar, click on *edit* → *Profile preferences*. A configuration window should now open up.
3. Click on the tab *Title and command*
4. The last option *When command exists:*, choose *Hold the terminal open*, instead of *Exit the terminal*, as shown in Fig. 6

4.2 Flying the Parrot A.R. Drone

After the demonstration of the basic functionalities of VIKI in the previous section, the reader is guided through a more advanced tutorial, showing the possibilities of using of VIKI modules and system setup. The system that is chosen for this tutorial is a Parrot A.R. Drone, as it is a commonly used platform for experimentation with drones, as well as a commercially affordable system for educational purposes. In this tutorial, the reader will be shown how to launch this drone and fly it with a joystick. In the actual implementation of VIKI, out of the box packages are provided in the form of existing VIKI modules. Whereas the desired module is not available yet in the VIKI module repository, the user is invited to follow the documentation available on <http://viki.readthedocs.io>.

In order to complete this part of the tutorial, the hardware necessary to launch the system needs to be available and ready to be used. The Parrot A.R. Drone and the joystick are the elements that need to be set up. Important information for a proper configuration of the modules are the IP address of the drone, as well as the device location of the joystick. In order to set up the drone, it must be turned on and the computer running VIKI should be connected to its wireless network. This network

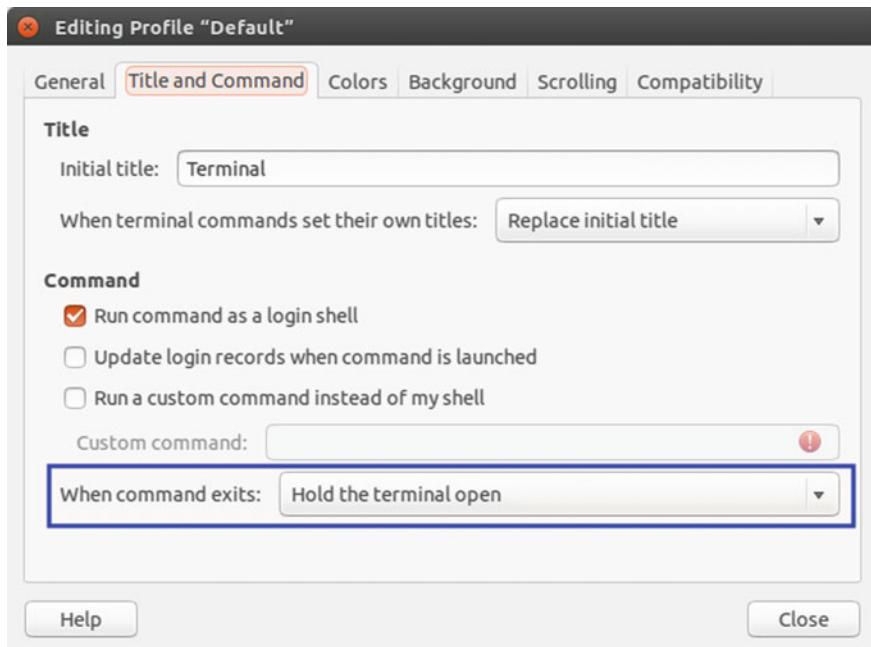


Fig. 6 Set the terminal window to hold open after launch to be able to read the errors that ROS may raise during runtime

is usually called *ardrone_xxxx*, with xxxx being an identifier. If all is correct, the default IP for it is *192.168.1.1*. More complex network set ups are possible, but this is out of the scope of this tutorial and will therefore not be covered hereafter.

The setup of the joystick requires the knowledge of the device name. This can be discovered by opening a terminal and typing the command:

ls -al /dev/input |grep js.

This will provide a list of all the devices that are recognized as joystick by the machine. Most likely, the joystick device will be listed as */dev/input/js0*. In case multiple joysticks are found, it is possible to test and find the correct joystick location by using a program called *jstest*.

After setting up the hardware for this tutorial, the software architecture can be designed in a similar manner as in the previous tutorial. The steps that are necessary to do so are:

- launch VIKI
- drag the modules called *Joystick node*, *Parrot A.R. Drone* and *Image view* to the canvas (Fig. 7)
- setup the desired connections as illustrated in Fig. 8: drag from the *cmd_vel_joy* to the input *cmd_vel* of the Parrot A.R. Drone module.



Fig. 7 The modules that are needed for launching the Parrot A.R. Drone

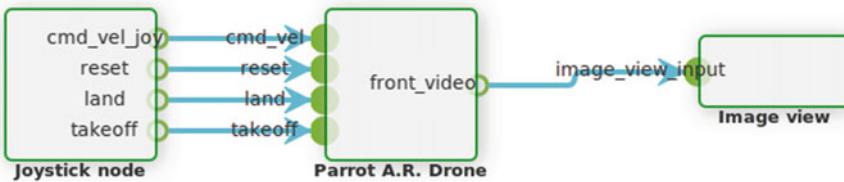


Fig. 8 Modules on the canvas connected in the right way

The Joystick node will read from the joystick and publishes *cmd_vel* messages on the *cmd_vel_joy* topic. The Parrot A.R. Drone and Image view modules are wrappers for existing ROS packages, providing functionality to launch the parrot and visualize image topics respectively.

Note that while dragging, it is possible to see that this input of the Parrot A.R. Drone turns green, while the reset, land and takeoff entry points turn red. VIKI colours the in- and outputs while dragging based on topic type. This prohibits linking wrong topics to each other. You should now also be able to drag the right topics from the joystick module to the Parrot module. The Parrot module also provides an video stream from its front camera. By linking this to the Image view module, VIKI will show you this video stream to a window on your screen.

By click on the *Joystick* module, it is possible to configure the joystick parameters (Fig. 9). On the right-hand side, in the properties panel (as indicated by 4 in Fig. 3), you should see a set of possible parameters, mostly for configuring the buttons that the user may like to use. The most important setting is the *dev* setting, which configures the joystick device to be used. These parameters are for reference shown in Fig. 9. Enter the address of the device previously discovered here.

The Parrot should be configured correctly by default. If this is not the case, it will be necessary to reconfigure the software in order for the Parrot to respond to any command. The ardrone autonomy package uses arguments to configure the drone. These arguments can be set using VIKI as well. In order to do so, it is necessary to select the Parrot module and click on *add/edit arguments* on the right column. A window should open up, similar to the one of Fig. 10. Change the text field to “-ip [ip]”, where the user needs to substitute the IP address of the ardrone in this field.

Fig. 9 The properties panel for the joystick, where **dev** can be set to `/dev/input/js0`

Properties

Joystick node

- Add/edit arguments
- Add/edit prefixes
- Select machine

Uid: joystick_lu7q

Property	Value
dev	<code>/dev/input/js0</code>
x_axis	2
y_axis	1
z_axis	4
yaw_axis	3
x velocity max	

Fig. 10 The arguments panel in which extra launch arguments for VIKI can be added



Following these instructions should achieve proper setup of the necessary hardware and VIKI software architecture. By pressing the big green RUN button it is possible to launch the set up that has just been created. By doing so, a video stream should pop-up displaying the videotostream of the front camera of the A.R. Drone, thus enabling an *on-eye* viewpoint when piloting. The user can finally save the configuration.

4.3 SSH

The previous two sections have guided the reader through setting up the VIKI canvas and module configurations to launch an UAV using VIKI. In practical situations, these UAVs are often controlled on a distributed system (e.g. an onboard computer connected to a ground station). ROS provides the functionality to delegate launches to other machines, using SSH. The GUI of VIKI has support to configure this and automatically generate it as well, by exploiting the capabilities of the roslaunch runtime layer.

This section is based on the documentation on distributed systems, that can be found in the main documentation of VIKI. This section is split in two parts. The first part covers the network configuration that needs to be applied to run a distributed system. This makes sure that the computers can reach each other using SSH. The second part shows how to launch the complete software architecture from the centralised VIKI canvas.

Network configuration: this section introduces the inexperienced user to configure two computers on the same local network. In case a different setup has to be configured, the user should make sure to access the PCs by hostname. More precisely, this section guides the user through the setup of one *master* computer, which will run VIKI, and a *slave* which will launch ROS nodes.

First of all, make sure the following prerequisites are satisfied:

- the computers used are connected on the same network where a wireless networking adapter is also available.
- there is no firewall between the computers that may block the connections between them. When on a local network, this is usually the case by default
- VIKI is installed on the *master* computer
- ROS is installed on both computers.³

For the computers to find each other, usually a DNS server is used. Since the discussed setup does not use it, the hostnames have to be added by hand to the hosts file of the computer, which takes care of resolving hostnames as well. To do so, it is necessary to open a terminal and run these two separate commands on each computer to retrieve the local IP address and hostname:

- *ifconfig*: this will show, among other information, the IP address of the machine. This is located at the *inet addr* field, under the adapter that you are using.
- *hostname*: this will print out a single line with the default hostname of your computer.

This information needs to be known to each of the computers and can be added to the */etc/hosts* file. To do this, open up this file in an editor by, e.g.

³Note that ROS versions do not need to be the same, however VIKI follows the ROS updates and this might change in the future.

```
sudo gedit /etc/hosts.
```

For each external host that needs to be reached from this computer, a line for that computer has to be added.

If this is done correctly, it should now be possible to reach each other computer from this one. This can be tested by typing *ping <hostname>*.

Once the hostnames are set up correctly, we can use specific capabilities of this in VIKI. In case of any issue on this system configuration, the reader is invited to make a quick search on the ROS documentation on distributed systems, or at the VIKI documentation itself.

Launching distributed systems inside VIKI: in this last part of the tutorial, the Parrot AR. Drone will be launched from inside VIKI, which is ran on the *master* computer, while the parrot is connected to the *slave* PC.

In order to prepare the hardware to allow this tutorial to be executed, it is necessary to make sure that the *slave* PC has both an ethernet adapter. If this is the case, perform the following steps:

- connect the slave PC to your local network using the ethernet adapter
- connect the Parrot A.R. Drone to the slave PC using the wireless adapter.

At completion of the hardware connections, perform the following steps:

1. launch VIKI on the *master*
2. open the configuration from the previous section, including the modules for the joystick node, parrot and image view.
3. Click the harddrive icon in the toolbar, *Open Machine list*, to show the list with machines. A panel as shown in Fig. 11 should open up.
4. change the hostname to the hostname of your master PC by pressing the edit sign in the corner to the right.
5. click on plus sign to add a machine, which should show a panel as in Fig. 12.
6. the name is used to reference this machine later on in VIKI. For hostname, username and password, enter the necessary values to be able to connect to the slave PC and click *Save* twice to go back to the main canvas.
7. click on the Parrot Module
8. on the right bar, click on *Select Machine*. This opens up a panel as in Fig. 13.
9. select the machine you just created
10. click *Save changes*.

The full configuration is now completed. Pressing the *RUN* button launches this setup. The outcome of this tutorial should behave similarly as in the previous section, with the difference that modules are now running on two different machines.

We understand that this specific example is difficult to copy without having the proper hardware available, but to our belief it is necessary to include, as it proves that VIKI is very capable of handling these kind of tasks.

Fig. 11 The overview of machines within VIKI. Note that the *ROSMASTERURI* can also be set here, as well as viewing/editing the list of remote machines

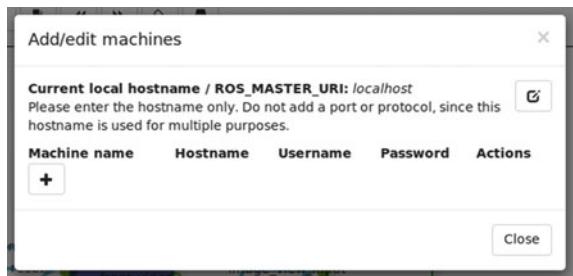


Fig. 12 The screen to add a machine within VIKI

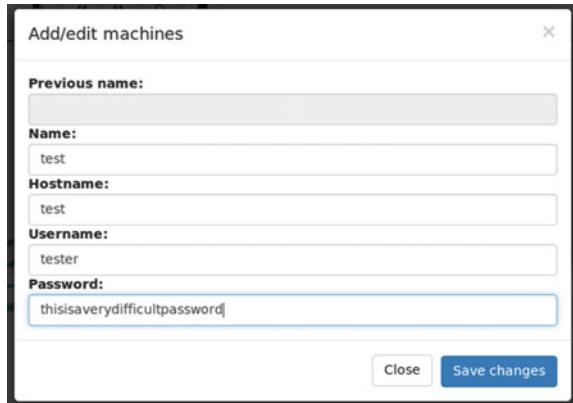
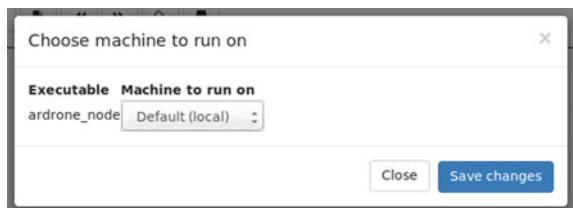


Fig. 13 The panel in which a machine for a specific module can be chosen



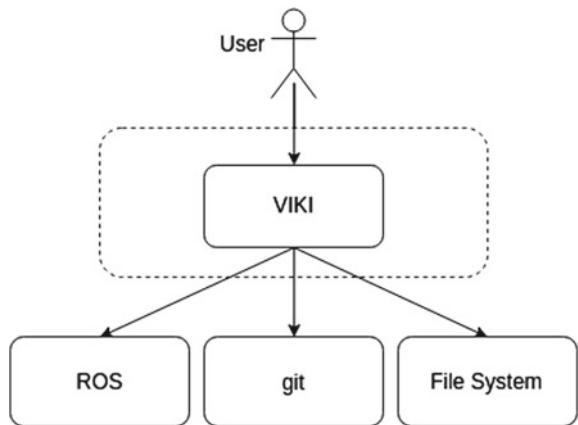
5 Technical Overview

This section provides a technical overview of VIKI and the components it is build of, and it relates VIKI to the other tools putting VIKI in perspective with most of the tools that are originally available by the ROS environment.

5.1 VIKI Architecture

From a broad perspective, VIKI can be considered as a tool that provides an interface between the user and low-level software. It assists the user in interacting with ROS,

Fig. 14 Overview of the structure of VIKI in the environment. The user interacts with VIKI, while VIKI interacts with ROS, git and the File System to provide the functionality



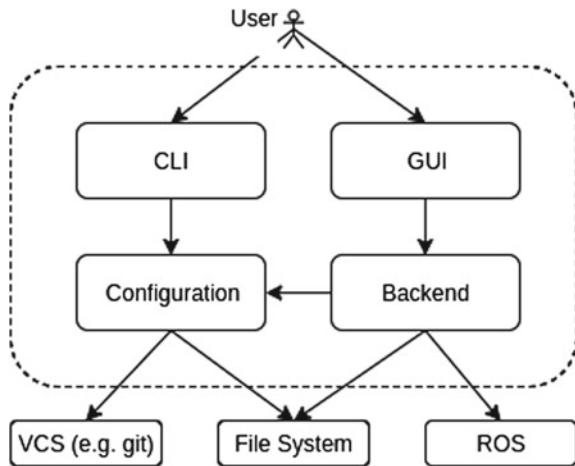
version control systems like *git* and *mercurial*, and the file system, as shown in Fig. 14. This allows users to use VIKI as an interface to configure, create environments and start their projects, without requiring them to re implement or make explicit use of these specific details and tools. In this context, ROS is used to launch the setup and execute the software of the end user. Version Control is used for updating and creating VIKI modules, and the file system is used for storing configurations and finding available modules.

In this sense, VIKI aims to enhance, rather than replace other ROS tools. More precisely, the authors aim at providing a tool that can be used by both new and experienced ROS users, without affecting the options to use existing runtime tools, such that the user can still take advantage of these.⁴ As Fig. 14 shows, VIKI is a level of abstraction between the user and ROS, and not between ROS and the environment. After the user presses the RUN button present inside the VIKI GUI interface, ROS is launched using an automatically generated launch file. Note that this does not prevent other applications to interact with the ROS-based software, thus making it possible to use the developers' preferred debugging and monitoring tools.

From a lower level, architectural point of view, the structure of VIKI is defined by 4 distinct components; being the Command line interface (CLI), Graphical User Interface (GUI), Configuration Component and the Backend Component. The architecture of these 4 components is shown in Fig. 15. The Backend handles all main functionalities of VIKI. The user can access these functionalities by launching the GUI. The GUI only provides an interface and does not handle any logic specific to VIKI. In order to support the GUI, an CLI is provided to aid in small configuration tasks. More precisely, the CLI is used for configuration, installation of module repositories and other support features. When the CLI has created the configuration,

⁴Note that VIKI is still under strong development after the first release. Some of the advanced ROS tools, such as the multi-master, could not be yet fully integrated at the moment of reading this chapter. For an update on the current status of the installation, the interested user may refer to the online documentation.

Fig. 15 Overview of the internal structure of VIKI. VIKI provides both a CLI and GUI as interface for the user. The backend handles functionality using ROS and the File System. The GUI provides the interface to this backend for the user. The CLI is used for configuration, to be used by the backend



the GUI runs using this configuration. When modifications need to be done by the user, the CLI provides a quick and solid way to change allow these changes. The configuration component is separate and is used for internal configuration purposes. It only stores information coming from the CLI and Backend, but does not provide additional functionality.

5.2 Modules

One of the core concepts in VIKI are modules. For a user familiar with ROS it may be difficult at first to differentiate between packages or nodes and VIKI's modules. This may be very well due to the deliberate design decision that a VIKI module could act as a single node, or as a package. The goal of a module is to provide coherent functionality for a specific use case.

VIKI handles abstraction of ROS packages by providing these as modules, as shown in Fig. 16. The first level shows available ROS packages. Using module description files, the ROS packages are predefined for the end-user as modules. Using VIKI, these modules can be arranged in a configuration file, which can be processed to a ROS launch file, which will be launched using ROS. By predefining essential information of these packages in the module description files, users can use these modules directly.

Module description files are always named *viuki.xml* and put into separate directories inside the ROS workspace. These description files are of the type XML.

A default module existing in VIKI for instance called *twist_from_joy* consists of two nodes:

- the *joy_node* node from the *joy* package
- a custom node *fly_from_joy* that translates joy messages into sensible twist messages for the operation of, e.g., a multicopter.

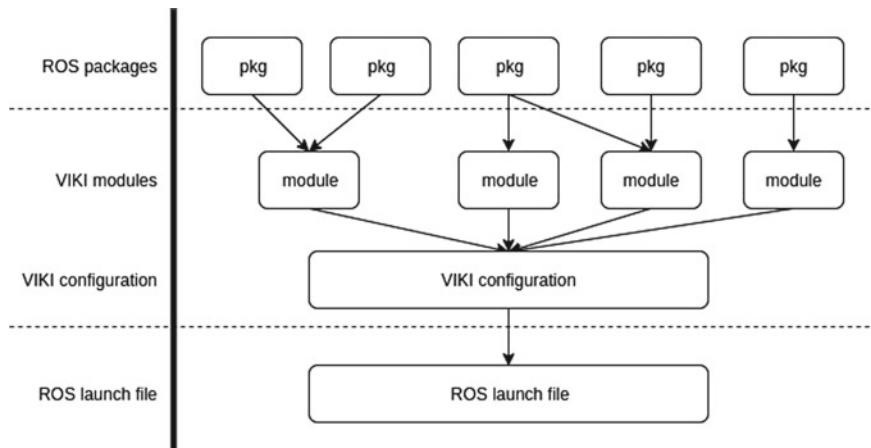


Fig. 16 Visual overview of the abstraction layer in VIKI. VIKI modules are build on top of ROS packages, embedded with extra information. With these modules, a configuration file is built, which is converted to a ROS launch file. This launch file is used by VIKI during runtime

Note that these two nodes are not in the same package, but are in the same module.

The user will thus, on first sight, only see one VIKI module. This module provides a single output which in the back end corresponds with a ROS topic on which the twist messages are published. This is an important aspect of the back end: it leaves the regular ROS structure completely intact. If one were to run a tool to visualize the running nodes and active topics (e.g. *rqt_graph*) both nodes mentioned above will be active and connected through a uniquely named relay node. This provides experienced users with the possibility to leave their existing ROS work flow unaltered.

Even though modules may be a simplification of a sub-system within a desired configuration of ROS nodes, VIKI allows to fully customize how these nodes are launched by passing parameters, command line arguments or launch prefixes similar to how ROS launch does. In light of the desire to run certain nodes on other machines, support for adding SSH required tags to a launch file has been implemented.

For most of the VIKI users, a complete combination of all possible functionalities will not be provided in terms of available modules belonging to the VIKI module library. Therefore, this requires the user to define custom modules where it is considered necessary. Creating a module is done by creating an XML file describing the properties of the module. This XML file contains:

- Meta information, such as title and description.
- Inputs and outputs of the module. These provide the interface of the module and are linked to the in- and outputs of executables.
- Executables, which are ROS nodes. These contain information about the node such as the inputs, outputs and a set of default parameters to be used.
- Extra configuration options, which can be used to link executables internally or add extra options.

This information is stored in a *viki.xml* file using the XML format, and is placed in the ROS workspace. An example of such a file is shown in listing 1. The next section elaborates on the organisation of modules and how to increase the reusability. More information about the internals of the XML file can be found in the documentation online.

```
<!-- VIKI_MODULE -->
<module type="controller" id="simple_PID">
  <meta>
    <name>PID controller</name>
    <description>
      An example node with a PID controller
    </description>
  </meta>

  <!-- The in- and outputs of the module as a whole. They are linked to
  -->
  <inputs>
    <input type="ros_topic" name="pose" link="control_node/pose"
    <message_type="geometry_msgs/Pose" required="true" />
  </inputs>

  <outputs>
    <output type="ros_topic" name="command" link="control_node/setpoint"
    <message_type="geometry_msgs/Twist" required="false" />
  </outputs>

  <executable id="control_node" pkg="PID" exec="node">
    <inputs>
      <input type="ros_topic" name="pose"
    <message_type="geometry_msgs/Pose" required="true" />
    </inputs>
    <outputs>
      <output type="ros_topic" name="setpoint"
    <message_type="geometry_msgs/Pose" required="false" />
    </outputs>
    <params>
      <param name="gain" default="2" type="realnumber" />
    </params>
  </executable>

  <!-- Configuration of the module: a method to connect executables
  -->
  <configuration>
  </configuration>
</module>
```

Listing 1: Possible contents of a *viki.xml* file that holds the information about the ROS packages in the module.

5.3 Modularity and Reusability

One of the main requirements in the design of VIKI is that the additional abstraction layer should not conflict with the modularity that ROS packages and nodes offer. It in fact stimulates the user to leverage the powerful ROS structure that is built around reliable communication between independent nodes. Thanks to the ease of adding modules to the system and connecting them to other modules by simply dragging-and-dropping, VIKI promotes having modules that have a small, but well-defined functionality (as the example module *twist_from_joy* demonstrates).

VIKI relies on two principles to maintain full modularity and promote code reusability:

1. It uses existing communication infrastructure, thus relying on ROS topics, topic name spaces and the ROS *topic_tools* to abstract the process of connecting nodes.
2. It requires no change to existing node logic. Combining existing nodes and packages into a module will never require developers to change the code of their nodes. An additional XML-formatted file instructs VIKI on how to combine existing nodes.

These principles not only guarantee that VIKI will always be compatible with existing ROS infrastructure, but it greatly promotes abstracting projects in small, reusable parts.

VIKI scans the specified 'root module directory' recursively for *viki.xml* files on startup. This means all module definition files live in subfolders inside this root folder. This gives the user flexibility on how to organise his or her personal modules. In order to keep overview, VIKI encourages two separate methods of locating module files.

To allow for modules to be shared and used between different developers, VIKI introduces the concept of *module repositories*. These are git repositories that include a set of module description files. The second option is to include a *viki.xml* file directly in the repository of a ROS package. Parts of the modules used in projects can originate from the original, open-source VIKI module repository, while other parts can be located in a private repository for the project team. These built-in functionalities prove that VIKI is designed with sharing well-defined, reusable modules in mind.

Thus, there are two different methods for sharing module description files, each with their own goal.

- Using module repositories. VIKI makes it possible to add different module repositories and manage these using version control. When a new module is required, the user can create a new folder inside this repository and add a new *viki.xml* file to this folder. This module file can use packages that are available on the system (e.g. using apt-get), or include code for a small package itself inside a subfolder. This approach is preferred when a binary installation of the ROS packages requires to be added to a VIKI module, or when it is desired to combine nodes of several packages into one module. When it is required to include a large package inside a repository, it is encouraged to put this in a separate repository and include this as

a dependency in the description file, as this keeps module repositories small and easy to use.

- Add *viki.xml* file to the ROS package. This is the preferred method when using the ROS package directly from source when designing a VIKI module **specifically for that package**. Note that in this case, it is important that the ‘root module directory’ is specified as the root of the catkin workspace (in the *config.json* file), such that all directories in the workspace are scanned.

All modules for the first method live within the *viki_modules* directory inside the catkin workspace. A good use-case for these repositories would be a project inside a research group. For this project, a new repository could be created which includes all these module files. The users can easily pull these repositories from remote storage locations and use them directly within VIKI to browse the different modules that are available. This gives users inside this project a quick overview of what is already available and the components that can be used directly.

5.4 GUI

Many tools within ROS are aimed at providing overview after the software has launched (see Sect. 2.1). The VIKI GUI, on the other hand, aims at creating a visual building space to compose projects and complex software architectures, while providing the user with a direct overview. This is done at the abstraction level of VIKI, providing overview at a higher level. When detailed overview at ROS level is needed, *rqt_graph* can still be used. While *rqt_graph* provides a graphical overview of all nodes and topics after launch, VIKI provides this overview between a set of modules with a subset of topics.

From within the GUI, all available modules are listed in a *palette* which can be searched through quickly based on module name or description. From there modules can easily be dragged onto the canvas and connected to other modules. The canvas then provides a visual representation of the architecture. Modules can be connected via dragging and dropping arrows representing data flow. The GUI will provide visual feedback on which topic types match. For every module it is possible to edit settings on the executable (ROS node) level.

The GUI provides an all-in-one run button, which starts the created project. Behind the scenes, VIKI generates a ROS launch file which is launched within a separate thread. While the steps from GUI to launch are abstracted from the end-user, they are easy to run independently. It is possible to generate a launch file using VIKI, copy it to another computer and launch from there, provided that the ROS packages are available.

5.5 Future Goals

VIKI is under heavy development. At the time of writing, the latest version is 0.2, which is the version this chapter is based on. The goal of VIKI is to reduce time researchers, students, as well as software integrators spend on setting up a robotic experiment. The vision behind the development of VIKI is to let users use it as a main design tool, while still allowing ease of access and use of the most important tools provided by ROS (and compatible with it), in order to boost the time of development of complex behaviors for robots. For this reason, VIKI has been designed to be open-source, and it is licenced⁵ under an MIT license. This has been chosen in order to allow building a community around VIKI. To reach this goal, focus has to be put on integrating VIKI, as well as enhancing the VIKI modules' repository, with existing ROS tools and packages that will guarantee better usability for the end user and more functionality within VIKI.

Development of VIKI's features will be based on the community and the feedback that is provided. The authors find it important to interact with the users and focus on building features that are requested most. Users can therefore obtain the required information for development by getting into contact with the authors through github.

Among the major functionalities that are still under development, is the possibility to use multi-master tools that allow full distribution of the ROS executables, while minimising the use of the communication bandwidth. It has been mentioned that VIKI has support for launching nodes using the distributed capabilities of ROS itself, but the correct functioning at the current version requires the startup of one single ROS core. Future goals on the short term include incorporating at least one method of running multiple masters.

Besides that, future goals will also aim at better integration with the existing ROS environment and improving the workflow during package and module development, to ensure users are not bound to only use VIKI. Features that support this might include automatic or interactive module generation and generating VIKI configuration or modules based on launch files. Specific decisions on the implementation of this will be discussed with the community and tailored to their needs, as already mentioned before.

References

1. Quigley, Morgan et al. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.
2. Muellers, Filip. 2012. rxDeveloper 1.3b with sourcecode generators. <http://www.ros.org/news/2012/04/rxdeveloper-13b-with-sourcecode-generators.html> (visited on 17 April 2016).
3. BRIDE. BRICS Integrated Development Environment. 2014. <http://www.best-of-robotics.org/bride/>.

⁵At the moment of writing of this chapter.

4. Fraunhofer FKIE. Node Manager FKIE. 2016. https://fkie.github.io/multimaster_fkie/node_manager.html (visited on 13 May 2016).
5. Robotnik. ROS graphic user interfaces. 2013. <http://www.robotnik.eu/ros-graphic-user-interfaces/> (visited on 17 April 2016).
6. Price, John H. Creating a Graphical user Interface for Joint Position Control in Controllit!. https://robotcontrolit.com/documentation/gui_jpos (visited on 17 April 2016).
7. Stumm, Elena. 2010. ROS/Web based Grahical User Interface for the sFly Project. Semester-Thesis. ETH Zurich. http://students.asl.ethz.ch/upl_pdf/289-report.pdf?aslsid=c472f08de49967cf2e11840561d8175a.
8. Willow Garage. ROS GUI. 2012. <http://www.willowgarage.com/blog/2012/10/21/ros-gui> (visited on 17 April 2016).
9. Kaestner, Ralf. 2016. Plugins Related to ROS TF Frames. <https://github.com/ethz-asl/ros-tf-plugins> (visited on 17 April 2016).
10. James (Techsource Systems). ROS GUI. 2015. <https://de.mathworks.com/matlabcentral/fileexchange/50640-ros-gui> (visited on 17 April 2016).
11. Linkbot. Linkbot Labs. 2016. <http://www.barobo.com/downloads/> (visited on 17 April 2016).
12. Hoogervorst, Robin, Alex Kamphuis, and Cees Trouwborst. VIKI documentation. 2016. <http://viki.readthedocs.io> (visited on 09 Sep 2016).

Author Biographies

Robin Hoogervorst is a Master student Computer Science at the University of Twente. He has successfully completed the Bachelor Advanced Technology. Based on the research from his Bachelor Assignment, a paper called ‘*Vision-IMU based collaborative control of a blind UAV*’ has been published on RED-UAS Mexico. His main interests are in the field of Software Engineering, more specifically at creating solid and dynamic software which people love.

Cees Trouwborst is a Master student Systems and Control at the University of Twente, specializing in Robotics and Mechatronics. Before this, he finished the Bachelor Advanced Technology with a bachelor thesis on “Control of Quadcopters for Collaborative Interaction”. His areas of interest include autonomous systems, machine learning, Internet-of-Things and software architecture.

Alex Kamphuis is a Mechanical Engineering student at the University of Twente. He was awarded a bachelor of science degree in Advanced Technology after the completion of his thesis on the ‘implementation of the velocity obstacle principle for 3D dynamic obstacle avoidance in quadcopter waypoint navigation’.

Since then he pursues a master of science degree at the multiscale mechanics group. Part of his current research on sound propagation through granular media is conducted at the German Aerospace center in Cologne. It entails cooperation with experienced researchers on topics such as stress birefringence and zero gravity environments. As such he has performed experiments in over 60 zero-g parabolas.

His other interests are running, reading and music.

Matteo Fumagalli is Assistant Professor in Mechatronics within the Department of Mechanical and Manufacturing Engineering at Aalborg University. He received his M.Sc. in 2006 from Politecnico di Milano, and his PhD at University of Genoa, where he worked in collaboration with the IIT - Istituto Italiano di Tecnologia. He has been post-doc at the Robotics and Mechatronics group of the University of Twente, where he carried out research on advanced robotic system design and control.