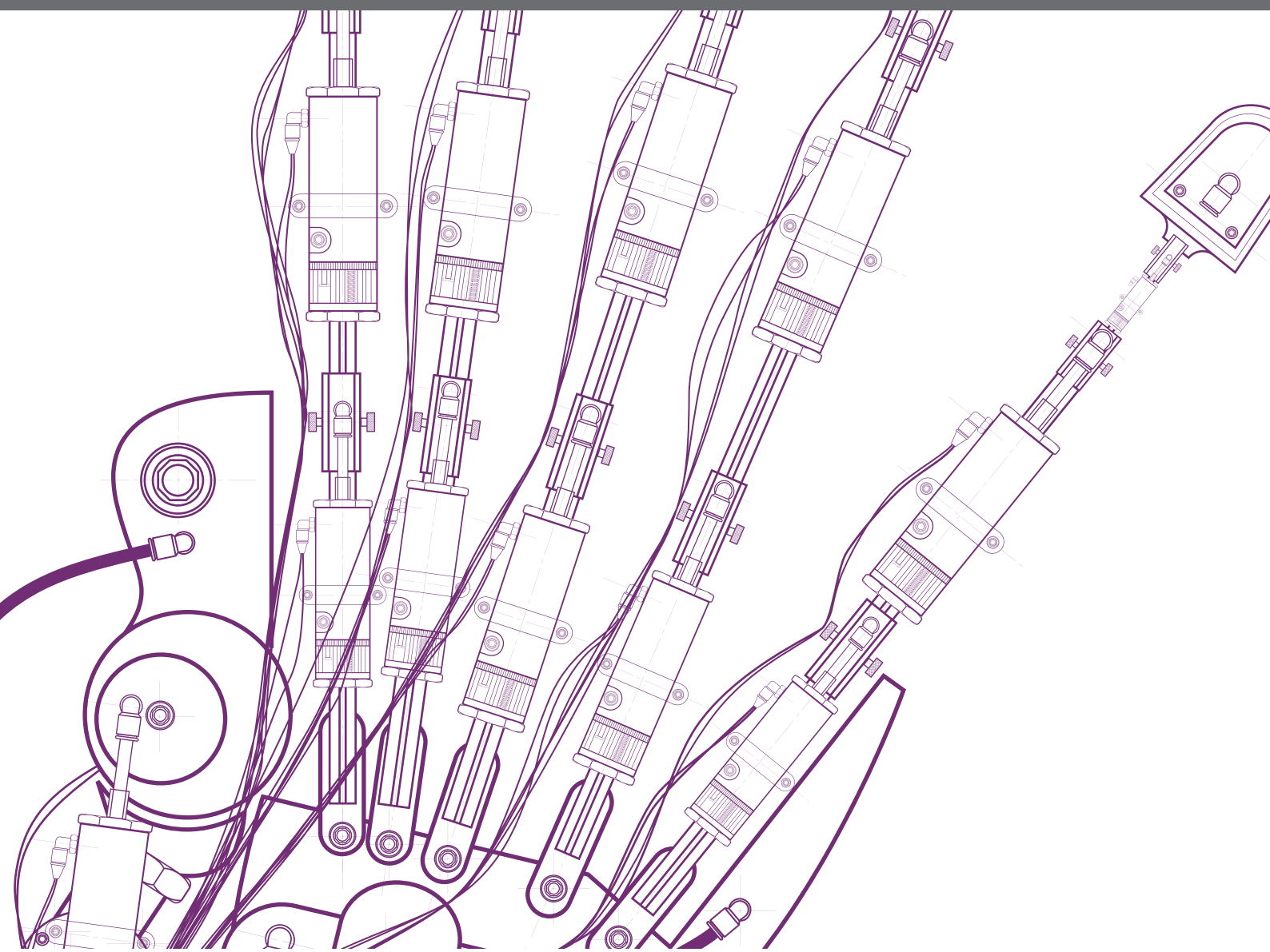
BUILDING THE ICUB MINDWARE: OPEN-SOURCE SOFTWARE FOR ROBOT INTELLIGENCE AND AUTONOMY

EDITED BY: Daniele Pucci, Vadim Tikhanoff, Ugo Pattacini, Maxime Petit and Lorenzo Jamone

PUBLISHED IN: Frontiers in Robotics and AI



frontiers Research Topics

frontiers

ISSN 1664-8714
ISBN 978-2-88963-541-2
DOI 10.3389/978-2-88963-541-2

About Frontiers

Frontiers is more than just an open-access publisher of scholarly articles: it is a pioneering approach to the world of academia, radically improving the way scholarly research is managed. The grand vision of Frontiers is a world where all people have an equal opportunity to seek, share and generate knowledge. Frontiers provides immediate and permanent online open access to all its publications, but this alone is not enough to realize our grand goals.

Frontiers Journal Series

The Frontiers Journal Series is a multi-tier and interdisciplinary set of open-access, online journals, promising a paradigm shift from the current review, selection and dissemination processes in academic publishing. All Frontiers journals are driven by researchers for researchers; therefore, they constitute a service to the scholarly community. At the same time, the Frontiers Journal Series operates on a revolutionary invention, the tiered publishing system, initially addressing specific communities of scholars, and gradually climbing up to broader public understanding, thus serving the interests of the lay society, too.

Dedication to Quality

Each Frontiers article is a landmark of the highest quality, thanks to genuinely collaborative interactions between authors and review editors, who include some of the world's best academicians. Research must be certified by peers before entering a stream of knowledge that may eventually reach the public - and shape society; therefore, Frontiers only applies the most rigorous and unbiased reviews.
Frontiers revolutionizes research publishing by freely delivering the most outstanding research, evaluated with no bias from both the academic and social point of view. By applying the most advanced information technologies, Frontiers is catapulting scholarly publishing into a new generation.

What are Frontiers Research Topics?

Frontiers Research Topics are very popular trademarks of the Frontiers Journals Series: they are collections of at least ten articles, all centered on a particular subject. With their unique mix of varied contributions from Original Research to Review Articles, Frontiers Research Topics unify the most influential researchers, the latest key findings and historical advances in a hot research area! Find out more on how to host your own Frontiers Research Topic or contribute to one as an author by contacting the Frontiers Editorial Office: researchtopics@frontiersin.org

BUILDING THE ICUB MINDWARE: OPEN-SOURCE SOFTWARE FOR ROBOT INTELLIGENCE AND AUTONOMY

Topic Editors:
Daniele Pucci, Italian Institute of Technology, Italy
Vadim Tikhanoff, Italian Institute of Technology, Italy
Ugo Pattacini, Italian Institute of Technology, Italy
Maxime Petit, Ecole Centrale de Lyon, France
Lorenzo Jamone, Queen Mary University of London, United Kingdom

Intelligence and autonomy are among the most extraordinary capacities blossomed by human evolution. Yet, endowing humanoid robots with these two crucial capabilities is still one of the biggest problems for the robotics community, despite decades of research. On the software side, algorithms for artificial intelligence are still at an embryonic stage. On the hardware side, robotic actuators are a far cry from the muscular human system in terms of flexibility and adaptability, which in turn reduces autonomy and robustness. Underneath the nature of algorithms for intelligence and technology for autonomy, the importance of efficient, scalable implementations of robust software goes without saying.

Among the large variety of humanoid robots, the iCub has emerged as one of the most diffused research platforms. It has been developed as part of the RobotCub EU project and subsequently adopted by more than 35 laboratories worldwide. Collaborations across laboratories are encouraged by writing code and libraries openly available. As a consequence, iCub is considered to be the ideal platform for experimenting and advancing open-source software for research in several domains, ranging from motor control to cognitive systems.

Citation: Pucci, D., Tikhanoff, V., Pattacini, U., Petit, M., Jamone, L., eds. (2020). Building the iCub Mindware: Open-source Software for Robot Intelligence and Autonomy. Lausanne: Frontiers Media SA. doi: 10.3389/978-2-88963-541-2

Table of Contents

frontiers in Robotics and AI

ORIGINAL RESEARCH
published: 05 October 2017
doi: 10.3389/frobt.2017.00045

Prediction of Intention during Interaction with iCub with Probabilistic Movement Primitives

Oriane Dermy[1,2,3], Alexandros Paraschos[4,5], Marco Ewerton[4], Jan Peters[4,6], François Charpillet[1,2,3] and Serena Ivaldi[1,2,3]*

[1]Inria, Villers-lés-Nancy, France, [2]Université de Lorraine, Loria, UMR7503, Vandoeuvre, France, [3]CNRS, Loria, UMR7503, Vandoeuvre, France, [4]TU Darmstadt, Darmstadt, Germany, [5]Data Lab, Volkswagen Group, Munich, Germany, [6]Max Planck Institute for Intelligent Systems, Tubingen, Germany

This article describes our open-source software for predicting the intention of a user physically interacting with the humanoid robot iCub. Our goal is to allow the robot to infer the intention of the human partner during collaboration, by predicting the future intended trajectory: this capability is critical to design anticipatory behaviors that are crucial in human–robot collaborative scenarios, such as in co-manipulation, cooperative assembly, or transportation. We propose an approach to endow the iCub with basic capabilities of intention recognition, based on Probabilistic Movement Primitives (ProMPs), a versatile method for representing, generalizing, and reproducing complex motor skills. The robot learns a set of motion primitives from several demonstrations, provided by the human via physical interaction. During training, we model the collaborative scenario using human demonstrations. During the reproduction of the collaborative task, we use the acquired knowledge to recognize the intention of the human partner. Using a few early observations of the state of the robot, we can not only infer the intention of the partner but also complete the movement, even if the user breaks the physical interaction with the robot. We evaluate our approach in simulation and on the real iCub. In simulation, the iCub is driven by the user using the Geomagic Touch haptic device. In the real robot experiment, we directly interact with the iCub by grabbing and manually guiding the robot's arm. We realize two experiments on the real robot: one with simple reaching trajectories, and one inspired by collaborative object sorting. The software implementing our approach is open source and available on the GitHub platform. In addition, we provide tutorials and videos.

Keywords: robot, prediction, intention, interaction, probabilistic models

OPEN ACCESS

Edited by:
Lorenzo Jamone, Queen Mary University of London, United Kingdom

Reviewed by:
Rodolphe Gelin, Aldebaran Robotics, France
Francesco Rea, Fondazione Istituto Italiano di Tecnologia, Italy
Mirko Wächter, Karlsruhe Institute of Technology, Germany

****Correspondence:***
Oriane Dermy
oriane.dermy@inria.fr

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *31 May 2017*
Accepted: *21 August 2017*
Published: *05 October 2017*

Citation:
Dermy O, Paraschos A, Ewerton M, Peters J, Charpillet F and Ivaldi S (2017) Prediction of Intention during Interaction with iCub with Probabilistic Movement Primitives. Front. Robot. AI 4:45. doi: 10.3389/frobt.2017.00045

1. INTRODUCTION

A critical ability for robots to collaborate with humans is to predict the intention of the partner. For example, a robot could help a human fold sheets, move furniture in a room, lift heavy objects, or place wind shields on a car frame. In all these cases, the human could begin the collaborative movement by guiding the robot, or by leading the movement in the case that both human and robot hold the object. It would be beneficial for the performance of the task if the robot could infer the intention of the human as soon as possible and collaborate to complete the task without requiring any further assistance. This scenario is particularly relevant for manufacturing (Dumora et al., 2013), where robots could help human partners in carrying a heavy or unwieldy object, while humans could

guide the robot without effort in executing the correct trajectory for positioning the object at the right location.[1] For example, the human could start moving the robot's end effector toward the goal location and release the grasp on the robot when the robot shows that it is capable of reaching the desired goal location without human intervention. Service and manufacturing scenarios offer a wide set of examples where collaborative actions can be initiated by the human and finished by the robot: assembling objects parts, sorting items in the correct bins or trays, welding, moving objects together, etc. In all these cases, the robot should be able to predict the goal of each action and the trajectory that the human partner wants to do for each action. To make this prediction, the robot should use all available information coming from sensor readings, past experiences (prior), human imitation and previous teaching sessions, or collaborations. Understanding and modeling the human behavior, exploiting all the available information, is the key to tackle this problem (Sato et al., 1994).

To predict the human intention, the robot must identify the current task, predict the user's goal, and predict the trajectory to achieve this goal. In the human–robot interaction literature, many keywords are associated with this prediction ability: inference, goal estimation, legibility, intention recognition, and anticipation.

Anticipation is the ability of the robot to choose the right thing to do in a current situation (Hoffman, 2010). To achieve this goal, the robot must predict the effect of their action, as studied with the concept of affordances (Sahin et al., 2007; Ivaldi et al., 2014b; Jamone et al., 2017). It also must predict the human intention, which means estimating the partner's goal (Wang et al., 2013; Thill and Ziemke, 2017). Finally, it must be able to predict the future events or states, e.g., being able to simulate the evolution of the coupled human–robot system, as it is frequently done in model predictive control (Ivaldi et al., 2010; Zube et al., 2016) or in human-aware planning (Alami et al., 2006; Shah et al., 2011).

It has been posited that having legible motions (Dragan and Srinivasa, 2013; Busch et al., 2017) helps the interacting partners in increasing the mutual estimation of the partner's intention, increasing the efficiency of the collaboration.

Anticipation requires thus the ability to visualize or predict the future desired state, e.g., where the human intends to go to. Predicting the user intention is often formulated as predicting the target of the human action, meaning that the robot must be able to predict at least the goal of the human when the two partners engage in a joint reaching action. To make such prediction, a common approach is to consider each movement as an instance of a particular skill or goal-directed movement primitive.

In the past decade, several frameworks have been proposed to represent movements primitives, frequently called *skills*, the most notable being Gaussian Mixture Models (GMM) (Khansari-Zadeh and Billard, 2011; Calinon et al., 2014), Dynamic Movement Primitives (DMP) (Ijspeert et al., 2013), Probabilistic Dynamic Movement Primitive (PDMP) (Meier and Schaal, 2016), and Probabilistic Movement Primitives (ProMP) (Paraschos et al., 2013a). For a thorough review of the literature, we refer the interested reader to Peters et al. (2016). Skill learning techniques have been applied to several learning scenarios, such as playing table tennis, writing digits, and avoiding obstacles during pick and place motions. In all these scenarios, the humans are classically providing the demonstrations (i.e., realizations of the task trajectories) by either manually driving the robot or through teleoperation, following the classical paradigm of imitation learning. Some of them have been also applied to the iCub humanoid robot: for example, Stulp et al. (2013) used DMPs to adapt a reaching motion online to the variable obstacles encountered by the robot arm, while Paraschos et al. (2015) used ProMPs to learn how to tilt a grate including torque information.

Among the aforementioned techniques, ProMPs stand out as one of the most promising techniques for realizing intention recognition and anticipatory movements for human–robot collaboration. They have the advantage, with respect to the other methods, of capturing by design the variability of the human demonstrations. They also have useful structural properties, as described by Paraschos et al. (2013a), such as co-activation, coupling, and temporal scaling. ProMPs have already been used in human–robot coordination for generating appropriate robot trajectories in response to initiated human trajectories (Maeda et al., 2016). Differently from DMPs, ProMPs do not need the information about the final goal of the trajectory, which is something that DMPs use to set an attractor that guarantees convergence to the final goal.[2] Also, they perform better in presence of noisy measurements or sparse measurements, as discussed in Maeda et al. (2014).[3] In a recent paper, Meier and Schaal (2016) proposed a method called PDMP (Probabilistic Dynamic Movement Primitive). This method improves DMP with probabilistic properties to measure the likelihood that the movement primitive is executed correctly and to perform inference on sensor measurement. However, The PDMPs do not have a data-driven generalization and can deviate arbitrarily from the demonstrations. These last differences can be critical for our humanoid robot (for example, if it collides with something during the movement, or if during the movement it holds something that can fall down due to a bad trajectory, etc.). Thus, the ProMPs method is more suitable for our applications.

In this article, we present our approach to the problem of predicting the intention during human–robot physical interaction and collaboration, based on Probabilistic Movement Primitives (ProMPs) (Paraschos et al., 2013a), and we present the associated open-source software code that implements the method for the iCub.

To illustrate the technique, the exemplifying problem we tackle in this article is to allow the robot to finish a movement initiated by the user that physically guides the robot arm. From the first observations of the joint movement, supposedly belonging to a

[1] Currently, this scenario is frequently addressed in manufacturing by robots and lifters; in the future, we imagine that humanoid robots could also be used for such task, for assisting workers in environments where robots cannot be installed on a fixed base, such as in some aircraft manufacturing operations (Caron and Kheddar, 2016).

[2] There may be applications where converging to a unique and precise goal could be a desirable property of the robot's movement. However, it is an assumption that prevents us to generalize the method for different actions, and this is another reason why we prefer ProMPs.

[3] We refer the interested reader to Maeda et al. (2014) for a thorough comparison between DMPs and ProMPs to be used for interaction primitives and prediction.

movement primitive of some task, the robot must recognize which kind of task the human is doing, predict the "future" trajectory, and complete the movement autonomously when the human releases the grasp on the robot.[4]

To achieve this goal, the robot first learns the movement primitives associated with the different actions/tasks. We choose to describe these primitives with ProMPs, as they are able to capture the distribution of demonstrations in a probabilistic model, rather than with a unique "average" trajectory. During interaction, the human starts physically driving the robot to perform the desired task. At the same time, the robot collects observations of the task. It then uses the prior information from the ProMP to compute a prediction of the desired goal together with the "future" trajectory that allows it to reach the goal.

A conceptual representation of the problem is shown in **Figure 1**. In the upper part of this figure, we represent the training step for one movement primitive: the robot is guided by the human partner to perform a certain task, and several entire demonstrations of the movement that realizes the task are collected. Both kinematics (e.g., Cartesian positions) and dynamics (e.g., wrenches) information are collected. The N trajectories constitute the base for learning the primitive, that is learning the parameters $\boldsymbol{\omega}$ of the trajectory distribution. We call this learned distribution the *prior distribution*. If multiple tasks are to be considered, then the process is replicated such that we have one ProMP for every task. The bottom of the figure represents the inference step. From the *early observations*[5] of a movement initiated by the human partner, the robot first recognizes which ProMP best matches the early observations (i.e., it recognizes the primitives that the human is executing, among the set of known primitives). Then, it estimates the future trajectory, given the early observations (e.g., first portion of a movement) and the prior distribution, computing the parameters $\boldsymbol{\omega}^*$ of the *posterior distribution*. The corresponding trajectory can be used by the robot to autonomously finish the movement, without relying on the human.

In this article, we describe both the theoretical framework and the software that is used to perform this prediction. The software is currently implemented in Matlab and C++; it is open source, available on `github`:

https://github.com/inria-larsen/icubLearningTrajectories

and it has been tested both with a simulated iCub in Gazebo and the real iCub. In simulation, physical guidance is provided by the Geomagic Touch[6]; on the real robot, the human operator simply grabs the robot's forearm.

We also provide a practical example of the software that realizes the exemplifying problems. In the example, the recorded trajectory is composed of both the Cartesian position and the forces at the end effector. Notably, in previous studies (Paraschos et al., 2015), ProMPs were used to learn movement primitives using joint positions. Here, we use Cartesian positions instead of joints positions to exploit the redundancy of the robotic arm in performing the desired task in the 3D space. At the control level of the iCub, this choice requires the iCub to control its lower-level (joint torque) movement with the Cartesian controller (Pattacini et al., 2010) instead of using the direct control at joint level. As for the forces, we rely on a model-based dynamics estimation that exploits the 6 axis force/torque sensors (Ivaldi et al., 2011; Fumagalli et al., 2012). All details for the experiments are presented in the article and the software tutorial.

[4] To avoid disambiguation, in our method, tasks are encoded by primitives that are made of trajectories: this is a very classical approach for robot learning techniques and in general techniques based on primitives. Of course this is a simplification, but it allows representing a number of different tasks: pointing, reaching, grasping, gazing, etc.

[5] In this article, we denote by *early observations* the first portion of a movement observed by the robot, i.e., from $t = 0$ to acurrent t.

[6] The Geomagic Touch is a haptic device, capable of providing force feedback from the simulation to the operator. In our experiments with the simulated iCub, we did not use this feature. We used the Geomagic Touch to steer the arm of the simulated robot. In that sense, we used it more as a joystick for moving the left arm.

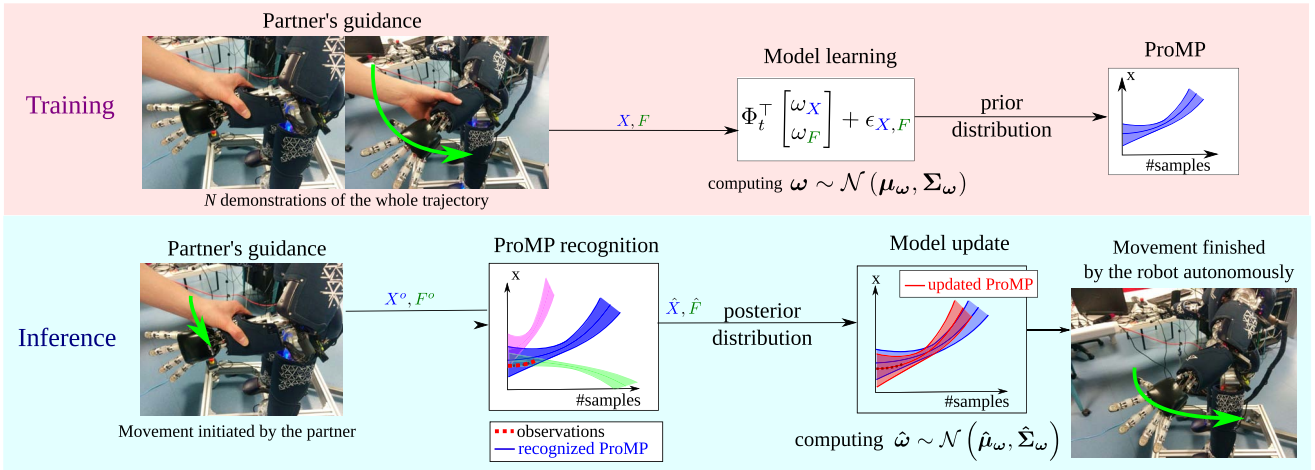


FIGURE 1 | Conceptual use of the ProMP for predicting the desired trajectory to be performed by the robot in a collaborative task. Top: training phase, where ProMPs are learned from several human demonstrations. Bottom: inference phase (online), where from early observations, the robot recognizes the current (among the known) ProMP and predicts the human intention, i.e., the future evolution of the initiated trajectory.

To summarize, the contributions of this article are as follows:

- the description of a theoretical framework based on ProMPs for predicting the human desired trajectory and goal during physical human–robot interaction, providing the following features: recognition of the current task, estimation of the task duration, and prediction of the future trajectory;
- an experimental study about how multimodal information can be used to improve the estimation of the duration/speed of an initiated trajectory;
- the open-source software to realize an intention recognition application with the iCub robot, both in simulation and on the real robot.

The article is organized as follows. In Section 2, we review the literature about intentions in Human–Robot Interaction (HRI), probabilistic models for motion primitives, and their related software. In Section 3, we describe the theoretical tools that we use to formalize the problem of predicting the intention of the human during interaction. Particularly, we describe the ProMPs and their use for predicting the evolution of a trajectory given early observations. In Section 4, we overview the software organization and the interconnection between our software and the iCub's main software, both for the real and simulated robot. The following sections are devoted to presenting our software and its use for predicting intention. We choose to present three examples of increasing complexity, with the simulated and real robot. We provide and explain in detail a software example for a 1-DOF trajectory in Section 5. In Sections 6 and 7, we present the intention recognition application with the simulated and real iCub, respectively. In the first examples with the robot, the "tasks" are exemplified by simple reaching movements, to provide simple and clear trajectories that help the reader understand the method, whereas the last experiment with the robot is a collaborative object sorting task. Section 8 provides the links to the videos showing how to use the software in simulation and on the iCub. Finally, in Section 10, we discuss our approach and its limitations and outline our future developments.

2. RELATED WORK

In this article, we propose a method to recognize the intention of the human partner collaborating with the robot, formalized as the target and the "future" trajectory associated with a skill, modeled by a goal-directed Probabilistic Movement Primitive. In this section, we briefly overview the literature about intention recognition in human–robot interaction and motion primitives for learning of goal-directed robotic skills.

2.1. Intention during Human–Robot Interaction

When humans and robots collaborate, mutual understanding is paramount for the success of any shared task. Mutual understanding means that the human is aware of the robot's current task, status, goal, available information, that he/she can reasonably predict or expect what it will do next, and *vice versa*. Recognizing the intention is only one piece of the problem but still plays a crucial part for providing anticipatory capabilities.

Formalizing intention can be a daunting task, as one may find it difficult to provide a unique representation that explains the intention for very low-level goal-directed tasks (e.g., reaching a target object and grasping it) and for very high-level, complex, abstract or cognitive tasks (e.g., change a light bulb on the ceiling—by building a stair composed of many parts, climbing it and reaching the light bulb on the ceiling, etc.). Demiris (2007) reviews different approaches of action recognition and intention prediction.

From the human's point of view, understanding the robot's intention means that the human should find intuitive and non-ambiguous every goal-directed robot movement or actions, and it should be clear what the robot is doing or going to do (Kim et al., 2017). Dragan and Srinivasa (2014) formalized the difference between *predictability* and *legibility*: a motion is legible if an observer can quickly infer its goal, while a motion is predictable when it matches the expectations of the observer given its goal.

The problem of generating *legible* motions for robots has been addressed in many recent works. For example, Dragan and Srinivasa (2014) use optimization techniques to generate movements that are predictable and legible. Huang et al. (2017) apply an Inverse Reinforcement Learning method on autonomous cars to select the robot movements that are maximally informative for the humans and that will facilitate their inference of the robot's objectives.

From the robot's point of view, understanding the human's intention means that the robot should be able to decipher the ensemble of verbal and non-verbal cues that the human naturally generates with his/her behavior, to identify, for a current task and context, what is the human intention. The more information (e.g., measurable signals from the human and the environment) is used, the better and more complex the estimation can be.

The simplest form of intention recognition is to estimate the goal of the current action, under the implicit assumption that each action is a goal-directed movement.

Sciutti et al. (2013) showed that humans implicitly attribute intentions in form of goals to robot motions, proving that humans exhibit anticipatory gaze toward the intended goal. Gaze was also used by Ivaldi et al. (2014a) in a human–robot interaction game with iCub, where the robot (human) was tracking the human (robot) gaze to identify the target object. Ferrer and Sanfeliu (2014) proposed the Bayesian Human Motion Intentionality Prediction algorithm, to geometrically compute the most likely target of the human motion, using Expectation–Maximization and a simple Bayesian classifier. In Wang et al. (2012), a method called Intention-Driven Dynamics model, based on Gaussian Process Dynamical Models (GPDM) (Wang et al., 2005), is used to infer the intention of the robot's partner during a ping-pong match, represented by the target of the ball, by analyzing the entire human movement before the human hits the ball. More generally, modeling and descriptive approaches can be used to match predefined labels with measured data (Csibra and Gergely, 2007).

A more complex form of intention recognition is to estimate the future trajectory from the past observations. In a sense, to estimate $[x_{t+1}, \ldots, x_{t+T_{future}}] = f(x_t, x_{t-1}, \ldots, x_{t-T_{past}})$. This problem, very similar to the estimate of the forward dynamics model of a system,

is frequently addressed by researchers in model predictive control, where being able to "play" the system evolving in time is the basis for computing appropriate robot controls. When a trajectory can be predicted by an observer from early observations of it, we can say that the trajectory is not only *legible*, but *predictable*. A systematic approach for predicting a trajectory is to reason in terms of movement primitives, in such a way that the sequence of points of the trajectory can be generated by a parametrized time model or a parametrized dynamical system. For example, Palinko et al. (2014) plan reaching trajectories for object carrying that are able to convey information about the weight of the transported object. More generally, in generative approaches (Buxton, 2003), latent variables are used to learn models for the primitives, both to generate and infer actions. The next subsection will provide more detail about the state-of-the-art techniques for generating movement primitives.

In Amor et al. (2014), the robot first learns Interaction Primitives by watching two humans performing an interactive task, using motion capture. The Interaction Primitive encapsulates the dependencies between the two human movements. Then, the robot uses the Interaction Primitive to adapt its behavior to its partner's movement. Their method is based on Dynamics Motor Primitives (Ijspeert et al., 2013), where a distribution over the DMP's parameters is learned. Notably, in this article, we did not follow the same approach to learn Interaction Primitives, since there is a physical interaction that makes the user's and the robot's movements as one joint movements. Moreover, there is no latency between the partner's early movement and the robot's, because the robot's arm is physically driven by the human until the latter breaks the contact.

Indeed, most examples in the literature focus on kinematic trajectories, corresponding to gestures that are typically used in dyadic interactions characterized by a coordination of actions and reactions. Whenever the human and robot are also interacting physically, collaborating on a task with some exchange of forces, then the problem of intention recognition becomes more complex. Indeed, the kinematics information provided by the "trajectories" cannot be analyzed without taking into account the haptic exchange and the estimation of the "roles" of the partners in leading/following each other.

Estimating the current role of the human (master/slave or leader/follower) is crucial, as the role information is necessary to coherently adapt the robot's compliance and impedance at the level of the exchanged contact forces. Most importantly, adapting the haptic interaction can be used by the robot to communicate when it has understood the human intent and is able to finish the task autonomously, mimicking the same type of implicit nonverbal communication that is typical of humans.

For example, in Gribovskaya et al. (2011), the robot infers the human intention utilizing the measure of the human's forces and by using Gaussian Mixture Models. In Rozo Castañeda et al. (2013), the arm impedance is adapted by a Gaussian Mixture Model based on measured forces and visual information. Many studies focused on the robot's ability to act only when and how its user wants (Carlson and Demiris, 2008; Soh and Demiris, 2015) and to not interfere with the partner's forces (Jarrassé et al., 2008) or actions (Baraglia et al., 2016).

In this article, we describe our approach to the problem of recognizing the human intention during collaboration by providing an estimate of the future intended trajectory to be performed by the robot. In our experiments, the robot does not adapt its role during the physical interaction but simply switches from follower to leader when the human breaks contact with it.

2.2. Movement Primitives

Movement Primitives (MPs) are a well established paradigm for representing complex motor skills. The most known method for representing movement primitives is probably the Dynamic Movement Primitives (DMPs) (Schaal, 2006; Ijspeert et al., 2013; Meier and Schaal, 2016). DMPs use a stable non-linear attractor in combination with a forcing term to represent the movement. The forcing term enables to follow specific movement, while the attractor asserts asymptotic stability. In a recent paper, Meier and Schaal (2016) proposed an extension to DMPs, called PDMP (Probabilistic Dynamic Movement Primitive). This method improves DMP with probabilistic properties to measure the likelihood that the movement primitive is executed correctly and to perform inference on sensor measurement. However, the PDMPs do not have a data-driven generalization and can deviate arbitrarily from the demonstrations. This last difference can be critical for our applications with the humanoid robot iCub, since uncertainties are unavoidable and disturbances may happen frequently and destabilize the robot movement (for example, an unexpected collision during the movement). Thus, the ProMPs method is more accurate for our software.

Ewerton et al. (2015), Paraschos et al. (2013b), and Maeda et al. (2014) compared ProMPs and DMPs for learning primitives and specifically interaction primitives. With the DMP model, at the end of the movement, only a dynamic attractor is activated. Thus, it always reaches a stable goal. The properties allowed by both methods are temporal scaling of the movement, learning from a single demonstration, and generalizing to new final position. With ProMPs, we have in addition the ability to do inference (thanks to the distribution), to force the robot to pass by several initial via points (the early observations), to know the correlation between the input of the model, and to co-activate some ProMPs. In our study, we need these features, because the robot must determine a trajectory that passes by the early observations (beginning of the movement where the user guides physically the robot).

A Recurrent Neural Networks (RNN) approach (Billard and Mataric, 2001) used a hierarchy of neural networks to simulate the activation of areas in human brain. The network can be trained to infer the state of the robot at the next point in time, given the current state. The authors propose to train the RNN by minimizing the error between the inferred position of the next time step and the ground truth obtained from demonstrations.

Hidden Markov Models (HMMs) for movement skills were introduced by Fine et al. (1998). This method is often used to categorize movements, where a category represents a movement primitive. This method also allows to represent the temporal sequence of a movement. In Nguyen et al. (2005), they use learned Hierarchical Hidden Markov Model (HHMMs) to recognize human behaviors efficiently. In Ren and Xu (2002), they present the Primitive-based Coupled-HMM (CHMM) approach,

for human natural complex action recognition. In this approach, each primitive is represented by a Gaussian Mixture Model.

Adapting Gaussian Mixture Models is another method used to learn physical interaction with learning. In Evrard et al. (2009), they use GMMs and Gaussian Mixture Regression to learn, in addition to the position (joint information), force information. Using this method, a humanoid robot is able to collaborate in one dimension with its partner for a lifting task. In this article, we will also use (Cartesian) position and force information to allow our robot to interact physically with its partner.

A subproblem of movement recognition is that robots need to estimate the duration of the trajectory to align a current trajectory with learned movements. In our case, at the beginning of the physical Human–Robot Interaction (pHRI), the robot observes a partial movement guided by its user. Given this partial movement, the robot must first estimate what the current state of the movement is to understand what its partner intent is. Thus, it needs to estimate the partial movement's speed.

Fitts' law models the movement duration for goal-directed movements. This model is based on the assumption that the movement duration is a linear function of the difficulty to achieve a target (Fitts, 1992). In Langolf et al. (1976), they show that by modifying the target's width, the shape of the movement changes. Thus, it is difficult to apply Fitt's law when the size of the target can change. In Langolf et al. (1976) and Soechting (1984), they confirm this idea by showing that the shape of the movement changes with the accuracy required by the goal position of the movement.

Dynamics Time Warping (DTW) is a method to find the correlation between two trajectories that have different durations, in a more robust way than the Euclidean distance. In Amor et al. (2014), they modify the DTW algorithm to match a partial movement with a reference movement. Many improvements over this method exist. In Keogh (2002), they propose a robust method to improve the indexation. The calculation speed of DTW is improved using different methods, such as FastDTW, Lucky Time Warping, or FTW. An explanation and comparison of these methods are presented in Silva and Batista (2016), where they add their own computation speed improvement by using a method called Pruned Warping Paths. This method allows the deletion of unlikely data. However, a drawback of this well-known DTW method is they do not preserve the global trajectory's shape.

In Maeda et al. (2014), where they use a probabilistic learning of movement primitives, they improve the duration estimation of movements by using a different time warping method. This method is based on a Gaussian basis model to represent a time warping function and, instead of DTW, it forces a local alignment between the two movements without "jumping" some index. Thus, the resulting trajectories are more realistic, smoother, and this method preserves the global trajectories' shapes.

For inferring the intention of the robot's partner, we use Probabilistic Movement Primitives (ProMPs) (Paraschos et al., 2013a). Specifically, we use the ProMP's conditioning operator to adapt the learned skills according to observations. The ProMPs can encode the correlations between forces and positions and allow better prediction of the partner's intention. Further, the phase of the partner's movement can be inferred, and therefore the robot can adapt to the partner's velocity changes. ProMPs are more efficient for collaborative tasks, as shown in Maeda et al. (2014), where in comparison to DMPs, the root-mean square error of the predictions is lower.

2.3. Related Open-Source Software

One of the goals of this article is to introduce an open-source software for the iCub (but potentially for any other robot), where the ProMP method is used to recognize human intention during collaboration, so that the robot can execute initiated actions autonomously. This is not the first open-source implementation for representing movement primitives: however, it has a novel application and a rationale that makes it easy to use with the iCub robot.

In **Table 1**, we report on the main software libraries that one can use to learn movement primitives. Some have been also used to realize learning applications with iCub, e.g., Lober et al. (2014) and Stulp et al. (2013) or to recognize human intention. However, the software we propose here is different: it provides an implementation of ProMPs used explicitly for intention recognition and prediction of intended trajectories. It is interfaced with iCub, both

TABLE 1 | Open-source software libraries implementing movement primitives and their application to different known robots.

Software/library	Method	Code link	Language	Robot	Reference
Dynamical System Modulation for Robot Adaptive Learning via Kinesthetic Demonstrations	GMR	Hersch et al. (2008)	Matlab	Hoap3	Micha and Aude (2008)
pbdlib-matlab	HMM, GMM, and others	Calinon (2015)	Matlab	Baxter	Calinon (2016)
DMP learning with GMR	DMP and GMR	Calinon et al. (2012a)	Matlab or C	Coman	Calinon et al. (2012b)
Stochastic Machine Learning Toolbox	Kernel Functions, Gaussian Processes, Bayesian Optimization	Lober (2014)	C++ or Python	–	
pydmps	DMP	DeWolf (2013)	Python	Sarcos	Ijspeert et al. (2013)
Dynamical Systems approach to Learn Robot Motions	GMM and SEDS	Khansari (2011)	Matlab	iCub	Khansari-Zadeh and Billard (2011, 2012)
Function Approximation, DMP, and Black-Box Optimization (dmpbbo)	DMP	Stulp (2014)	Python or C++	iCub	Stulp et al. (2013), Lober et al. (2014)
Learning Motor Skills from Partially Observed Movements Executed at Different Speeds	ProMP	Ewerton (2016)	Matlab or Python	–	Ewerton et al. (2015)
icubLearningTrajectories	ProMP	Dermy (2017)	Matlab and C++	iCub	–

real and simulated, and addresses in the specific case of physical interaction between the human and the robot. In short, it is a first step toward adding intention recognition ability to the iCub robot.

3. THEORETICAL FRAMEWORK

In this section, we present the theoretical framework that we use to tackle the problem of intent recognition: we describe the ProMPs and how they can be used to predict trajectories from early observations.

In Section 2, we formulate the problem of learning a primitive for a simple case, where the robot learns the distribution from several demonstrated trajectories. In Section 3.3, we formulate and provide the solution to the problem of predicting the "future" trajectory from early observations (i.e., the initial data points). In Section 3.4, we discuss the problem of predicting the time modulation, i.e., predicting the global duration of the predicted trajectory. This problem is non-trivial, as by construction the demonstrated trajectories are "normalized" in duration when the ProMP is learned.[7] In Section 3.5, we explain how to recognize, from the early observations, to which of many known skills (modeled by ProMPs) the current trajectory belongs. In all these sections, we tried to present the theoretical aspects related to the use of ProMPs for the intention recognition application.

Practical examples of these theoretical problems are presented and explained later in sections 5–7. Section 5 explains how to use our software, introduced in Section 4, for learning one ProMP for a simple set of 1-DOF trajectories. Section 6 presents an example with the simulated iCub in Gazebo, while Section 7 presents an example with the real iCub.

3.1. Notation

To facilitate understanding of the theoretical framework, we first introduce the notations we use in this section and throughout the remainder of the article.

3.1.1. Trajectories

- $X(t) \in \mathbb{R}^3$, $X(t) = [x(t), y(t), z(t)]^{\mathrm{T}}$: the x/y/z-axis Cartesian coordinate of the robot's end effector.
- $F(t) \in \mathbb{R}^6$, $F(t) = [f_x, f_y, f_z, m_x, m_y, m_z]^{\mathrm{T}}$: the wrench contact forces, i.e., the external forces and moments measured by the robot at the contact level (end effector).
- $\xi(t) \in \mathbb{R}^D$: the generic vector containing the current value or state of the trajectories at time t. It can be monodimensional (e.g., $\xi(t) = [z(t)]$), or multidimensional (e.g., $\xi(t) = [X(t), F(t)]^{\mathrm{T}}$), depending on the type of trajectories that we want to represent with the ProMP.
- $\Xi = \Xi_{[1:t_f]} = [\xi(1), \ldots, \xi(t_f)]^{\mathrm{T}} \in \mathbb{R}^{D \cdot t_f}$ is an entire trajectory, consisting of t_f samples or data points.
- $\Xi_{i[1:t_{fi}]}$ is the i-th demonstration (trajectory) of a task, consisting of t_{fi} samples or data points.

3.1.2. Movement Primitives

- $k \in [1 : K]$: the k-th ProMP, among a set of K ProMPs that represent different tasks/actions.
- n_k: number of recorded trajectories for each ProMP.
- $S_k = \{\Xi_{\{k,1\}}, \ldots, \Xi_{\{k,n_k\}}\}$: set of n_k trajectories for the k-th ProMP.
- $\xi(t) = \Phi_t \boldsymbol{\omega} + \epsilon_\xi$ is the model of the trajectory with:
  - $\epsilon_\xi \sim \mathcal{N}(0, \beta)$: expected trajectory noise.
  - $\Phi_t \in \mathbb{R}^{D \times D \cdot M}$: radial basis functions (RBFs) used to model trajectories. It is a block diagonal matrix.
    - M: number of RBFs.
    - $\psi_{ji}(t) = \frac{e^{\frac{-(t-c_i)^2}{2h}}}{\sum_{m=1}^{M} e^{\frac{-(t-c_m)^2}{2h}}}$: i-th RBF for all inputs $j \in [1 : D]$.

    It must be noted that the upper term comes from a Gaussian $\frac{1}{\sqrt{2\pi h}} e^{\frac{-(t-c_i)^2}{2h}}$, where c_i and h are, respectively, the center and variance of the i-th Gaussian. In our RBF formulation, we normalize all the Gaussians.
  - $\boldsymbol{\omega} \in \mathbb{R}^{D \cdot M}$: time-independent parameter vector weighting the RBFs, i.e., the parameters to be learned.
- $p(\boldsymbol{\omega}) \sim \mathcal{N}(\mu_{\boldsymbol{\omega}}, \Sigma_{\boldsymbol{\omega}})$: normal distribution computed from a set $\{\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_n\}$. It represents the distribution of the modeled trajectories, also called *prior* distribution.

3.1.3. Time Modulation

- $\bar{s}$: number of samples used as reference to rescale all the trajectories to the same duration.
- $\Phi_{\alpha_i t} \in \mathbb{R}^{D \times D \cdot M}$: the RBFs rescaled to match the Ξ_i trajectory duration.
- $\alpha_i = \frac{\bar{s}}{t_{fi}}$: temporal modulation parameter of the i-th trajectory.
- $\alpha = \Psi_{\delta_{n_o}} \boldsymbol{\omega}_\alpha + \epsilon_\alpha$ is the model of the function mapping δ_{n_o} into the temporal modulation parameter α, with:
  - Ψ: a set of RBFs used to model the mapping between δ_{n_o} and α;
  - δ_{n_o} is the variation of the trajectory during the first n_o observations (data points); it can be $\delta_{n_o} = \xi(n_o) - \xi(1)$ if the entire trajectory variables (e.g., Cartesian position and forces) are considered, or more simply $\delta_{n_o} = X(n_o) - X(1)$ if only the variation in terms of Cartesian position is considered;
  - $\boldsymbol{\omega}_\alpha$: the parameter vector weighting the RBFs of the Ψ matrix.

3.1.4. Inference

- $\Xi^o = [X^o, F^o]^{\mathrm{T}} = [\xi^o(1), \ldots, \xi^o(n_o)]^{\mathrm{T}}$: early-trajectory observations, composed of n_o data points.
- Σ_ξ^o: noise of the initiated trajectory observation.
- $\hat{\alpha}$: estimated time modulation parameter of a trajectory to infer.
- $\hat{t}_f = \frac{\bar{s}}{\hat{\alpha}}$: estimated duration of a trajectory to infer.
- $\Xi^* = [\xi^o(1), \ldots, \xi^o(n_o), \xi^*(n_o+1), \ldots, \xi^*(t_f)]$: ground truth of the trajectory for the robot to infer.
- $\hat{\Xi} = [\hat{X}, \hat{F}]^{\mathrm{T}} = [\xi^o(1), \ldots, \xi^o(n_o), \hat{\xi}(n_o+1), \ldots, \hat{\xi}(\hat{t}_f)]^{\mathrm{T}}$: the estimated trajectory.
- $p(\hat{\boldsymbol{\omega}}) \sim \mathcal{N}(\hat{\mu}_{\boldsymbol{\omega}}, \hat{\sigma}_{\boldsymbol{\omega}})$: posterior distribution of the parameter vector of a ProMP using the observation Ξ^o.
- $\hat{k}$: index of the recognized ProMP from the set of K known (previously learned) ProMPs.

[7] In some tasks, *e.g.,* reaching, it is reasonable to assume that the difference of duration of the demonstrated trajectories is negligible; however, in other tasks, the duration of the demonstrated trajectories may vary significantly.

3.2. Learning a Probabilistic Movement Primitive (ProMP) from Demonstrations

Our toolbox to learn, replay and infer the continuation of trajectories is written in Matlab and available at:

https://github.com/inria-larsen/icubLearningTrajectories/tree/master/MatlabProgram

Let us assume the robot has recorded a set of n_1 trajectories: $\{\Xi_1, \ldots, \Xi_{n_1}\}$, where the i-th trajectory is $\Xi_i = \{\xi(1), \ldots, \xi(t_{f_i})\}$. $\xi(t)$ is the generic vector containing all the variables to be learned at time t, with the ProMP method. It can be monodimensional (e.g., $\xi(t) = [z(t)]$ for the z-axis Cartesian coordinate), or multidimensional (e.g., $\xi(t) = [X(t), F(t)]^{\mathrm{T}}$). Note that the duration of each recorded trajectory (i.e., t_{f_i}) may be variable. To find a common representation in terms of primitives, a time modulation is applied to all trajectories, such that they have the same number of samples $\bar{s}$ (see details in Section 3.4). Such modulated trajectories are then used to learn a ProMP.

A ProMP is a Bayesian parametric model of the demonstrated trajectories in the following form:

$$\xi(t) = \Phi_t \boldsymbol{\omega} + \epsilon_\xi, \tag{1}$$

where $\boldsymbol{\omega} \in R^M$ is the time-independent parameter vector weighting the RBFs, $\epsilon_\xi \sim \mathcal{N}(0, \beta)$ is the trajectory noise, and Φ_t is a vector of M radial basis functions evaluated at time t:

$$\Phi_t = [\psi_1(t), \psi_2(t), \ldots, \psi_M(t)]$$

with

$$\begin{cases} \psi_i(t) = \frac{1}{\sum_{j=1}^{M} \psi_j(t)} exp\left\{\frac{-(t-c(i))^2}{2h}\right\} \\ c(i) \quad = i/M \\ h \quad = 1/M^2. \end{cases} \tag{2}$$

Note that all the ψ functions are scattered across time.

For each Ξ_i trajectory, we compute the ω_i parameter vector to have $\xi_i(t) = \Phi_t \omega_i + \epsilon_\xi$. This vector is computed to minimize the error between the observed $\xi_i(t)$ trajectory and its model $\Phi_t \omega_i + \epsilon_\xi$. This is done using the Least Mean Square algorithm, i.e.:

$$\omega_i = (\Phi_t^\top \Phi_t)^{-1} \Phi_t^\top \xi_i(t). \tag{3}$$

To avoid the common issue of the matrix $\Phi_t^{\mathrm{T}} \Phi_t$ in equation (3) not being invertible, we add a diagonal term and perform Ridge Regression:

$$\omega_i = (\Phi_t^\top \Phi_t + \lambda)^{-1} \Phi_t^\top \xi_i(t), \tag{4}$$

where $\lambda = 10^{-11} \cdot 1_{D \cdot M \times D \cdot M}$ is a parameter that can be tuned by looking at the smallest singular value of the matrix $\Phi_t^{\mathrm{T}} \Phi_t$.

Thus, we obtain a set of these parameters: $\{\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_n\}$, upon which a distribution is computed. Since we assume Normal distributions, we have:

$$p(\boldsymbol{\omega}) \sim \mathcal{N}(\mu_{\boldsymbol{\omega}}, \Sigma_{\boldsymbol{\omega}}) \tag{5}$$

$$\text{with } \mu_{\boldsymbol{\omega}} = \frac{1}{n} \sum_{i=1}^{n} \boldsymbol{\omega_i} \tag{6}$$

$$\text{and } \Sigma_{\boldsymbol{\omega}} = \frac{1}{n-1} \sum_{i=1}^{n} (\boldsymbol{\omega_i} - \mu_{\boldsymbol{\omega}})^\top (\boldsymbol{\omega_i} - \mu_{\boldsymbol{\omega}}). \tag{7}$$

The ProMP captures the distribution over the observed trajectories. To represent this movement primitive, we usually use the movement that passes by the mean of the distribution. **Figure 2** shows the ProMP for a 1-DOF lifting motion, with a number of reference samples $\bar{s} = 100$ and number of basis functions $M = 5$.

This example is included in our Matlab toolbox as `demo_plot1DOF.m`. The explanation of this Matlab script is presented in Section 5. More complex examples are also included in the scripts `demo_plot*.m`.

3.3. Predicting the Future Movement from Initial Observations

Once the ProMP $p(\boldsymbol{\omega}) \sim \mathcal{N}(\mu_{\boldsymbol{\omega}}, \Sigma_{\boldsymbol{\omega}})$ of a certain task has been learned,[8] we can use it to predict the evolution of an initiated movement. An underlying hypothesis is that the observed movement follows to this learned distribution.

[8] That is, we computed the $p(\boldsymbol{\omega})$ distribution from the dataset $\{\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_n\}$, where each $\boldsymbol{\omega}_i$ is an estimated parameter computed from the trajectory demonstrations.

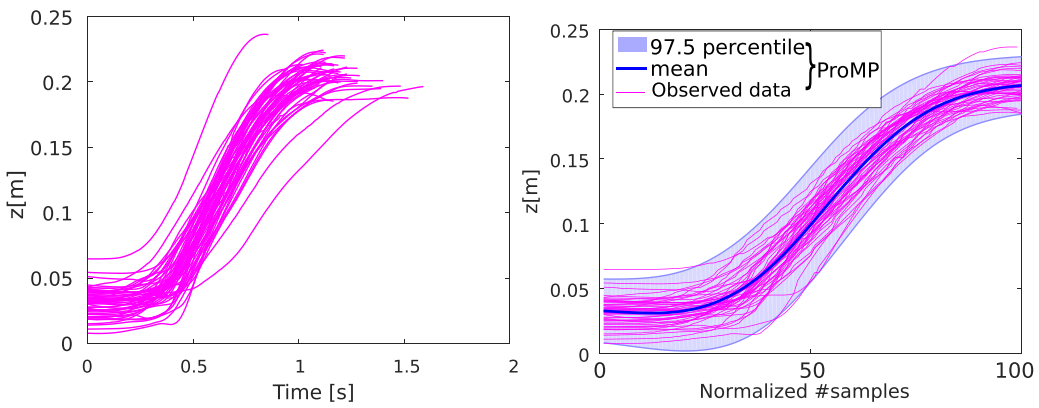


FIGURE 2 | The observed trajectories are represented in magenta. The corresponding ProMP is represented in blue. The following parameters are used: $\bar{s} = 100$ for the reference number of samples, $M = 5$ for the number of RBFs spread over time, and $h = 0.04$ $\left(= \frac{1}{M^2}\right)$ the variance of the RBFs.

Suppose that the robot measures the first n_o observations of the trajectory to predict (e.g., lifting the arm). We call these observations $\Xi^o = [\xi^o(1), \ldots, \xi^o(n_o)]$. The goal is then to predict the evolution of the trajectory after these n_o observations, i.e., find $\{\hat{\xi}(n_o + 1), \ldots, \hat{\xi}(\hat{t}_f)\}$, where $\hat{t}_f$ is the estimation of the trajectory duration (see Section 3.4). This is equivalent to predicting the entire $\hat{\Xi}$ trajectory where the first n_o samples are known and equal to the observations: $\hat{\Xi} = \{\xi^o(1), \ldots, \xi^o(n_o), \hat{\xi}(n_o + 1), \ldots, \hat{\xi}(t_{\hat{t}_f})\}$. Therefore, our prediction problem consists of predicting $\hat{\Xi}$ given the Ξ^o observations.

To do this prediction, we start from the learned prior distribution $p(\boldsymbol{\omega})$, and we find the $\hat{\boldsymbol{\omega}}$ parameter within this distribution that generates $\hat{\Xi}$. To find this $\hat{\boldsymbol{\omega}}$ parameter, we update the learned distribution $p(\hat{\boldsymbol{\omega}}) \sim \mathcal{N}(\hat{\mu}_{\boldsymbol{\omega}}, \hat{\Sigma}_{\boldsymbol{\omega}})$ using the following formulae:

$$\begin{cases} \hat{\mu}_{\boldsymbol{\omega}} = \mu_{\boldsymbol{\omega}} + K(\Xi^o - \Phi_{[1:n_o]}\, \mu_{\boldsymbol{\omega}}) \\ \hat{\Sigma}_{\boldsymbol{\omega}} = \Sigma_{\boldsymbol{\omega}} - K(\Phi_{[1:n_o]}\, \Sigma_{\boldsymbol{\omega}}), \end{cases} \tag{8}$$

where K is a gain computed by the following equation:

$$K = \Sigma_{\boldsymbol{\omega}} \Phi_{[1:n_o]}^{\top} \left(\Sigma_{\xi}^{o} + \Phi_{[1:n_o]} \Sigma_{\boldsymbol{\omega}} \Phi_{[1:n_o]}^{\top} \right)^{-1}. \tag{9}$$

Equations (8) and (9) can be computed through the marginal and conditional distributions (Bishop, 2006; Paraschos et al., 2013a), as detailed in Appendix A.

Figure 3 shows the predicted trajectory for the lifting motion of the left arm of iCub. The different graphs show inferred trajectories when the robot observed $n_o = 10$, 30, 50, and 80% of the total trajectory duration. This example is also available in the toolbox as `demo_plot1DOF.m`. The `nbData` variable changes the percentage of known data. Thus, it will be visible how the inference improves according to this variable. An example of predicted trajectories of the arm lifting in Gazebo can be found in a provided video (see Section 8).

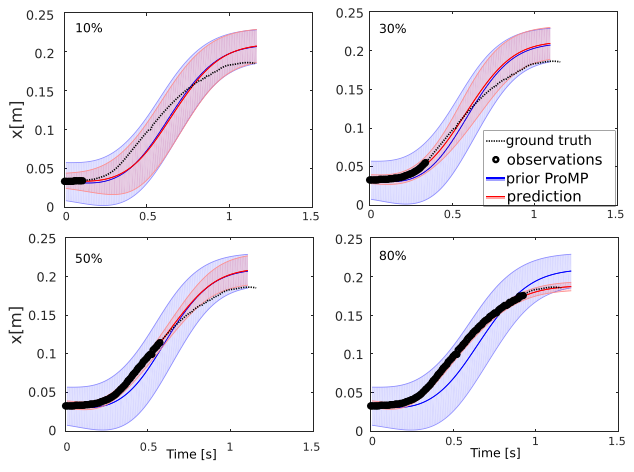


FIGURE 3 | The prediction of the future trajectory given early observations, exploiting the information of the learned ProMP (**Figure 2**). The plots show the predicted trajectories after 10, 30, 50, and 80% of observed data points.

3.4. Predicting the Trajectory Time Modulation

In the previous section, we presented the general formulation of ProMPs, which makes the implicit assumption that all the observed trajectories have the same duration and thus the same sampling.[9] That is why the duration of the trajectories generated by the RBF is fixed and equal to $\bar{s}$. Of course, this is valid only for synthetic data and not for real data.

To be able to address real experimental conditions, we now consider the variation of the duration of the demonstrated trajectories. To this end, we introduce a time modulation parameter α that maps the actual trajectory duration t_f to $\bar{s}$: $\alpha = \bar{s}/t_f$. The normalized duration $\bar{s}$ can be chosen arbitrarily; for example it can be set to the average of the duration of the trajectories, e.g., $\bar{s} = mean(t_{f1}, \ldots, t_{fK})$. Notably, in the literature sometimes α is called *phase* (Paraschos et al., 2013a,b). The effect of α is to change the phase of the RBFs, which are scaled in time.

The time modulation of the i-th trajectory Ξ_i is computed by $\alpha_i = \frac{\bar{s}}{t_{fi}}$. Thus, we have $\alpha \cdot t \in [1 : \bar{s}]$. Thus, the improved ProMP model is as follows:

$$\xi_t = \Phi_{\alpha t} \boldsymbol{\omega} + \epsilon_t, \tag{10}$$

where $\Phi_{\alpha t}$ is the RBFs matrix evaluated at time αt. All the M Gaussian functions of the RBFs are spread over the same number of samples $\bar{s}$. Thus, we have the following:

$$\Phi_{\alpha t} = [\psi_1(\alpha t), \psi_2(\alpha t), \ldots, \psi_M(\alpha t)].$$

During the learning step, we record a set of α parameters: $S_\alpha = \{\alpha_1, \ldots, \alpha_n\}$. Then, using this set, we can replay the learned ProMP with different speeds. By default (e.g., when $\alpha = 1$), the speed allows to finish the movement in $\bar{s}$ samples.

During the inference, the time modulation α of the partially observed trajectory is not known. Unless fixed *a priori*, the robot must estimate it. This estimation is critical to ensure a good recognition, as shown in **Figure 4**: the inferred trajectory (represented by the mean of the posterior distribution in red) does not have the same duration as the "real" intended trajectory (which is the ground truth). This difference is due to the estimation error of the time modulation parameter. This estimation $\hat{\alpha}$ by default is computed as the mean of all the α_k observed during the learning:

$$\hat{\alpha} = \frac{\sum \alpha_k}{n_k}. \tag{11}$$

However, using the mean value for the time modulation is an appropriate choice only when the primitive represents goal-directed motions that are very regular, or for which we can reasonably assume that differences in the duration can be neglected (which is not a general case). In many applications, this estimation may be too rough.

Thus, we have to find a way to estimate the duration of the observed trajectory, which corresponds to accurately estimating the time modulation parameter $\hat{\alpha}$. To estimate $\hat{\alpha}$, we implemented

[9] Actually, we call here duration what is in fact the total number of samples for the trajectory.

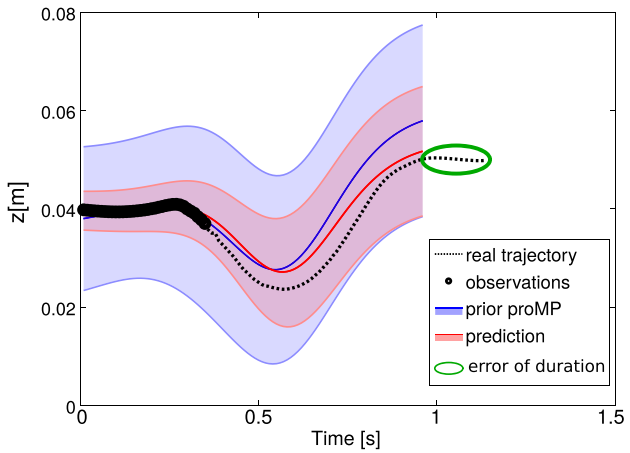


FIGURE 4 | This plot shows the predicted trajectory given early observations (data points, in black), compared to the ground truth (e.g., the trajectory that the human intends to execute with the robot). We show the prior distribution (in light blue) and the posterior distribution (in red), which is computed by conditioning the distribution to match the observations. Here, the posterior simply uses the average α computed over the $\alpha_1, \ldots, \alpha_K$ of the K demonstrations. Without predicting the time modulation from the observations and using the average α, the predicted trajectory has a duration that is visibly different from the ground truth.

four different methods. The first is the **mean of all the** α_k, as in equation (11). The second is the **maximum likelihood**, with

$$\hat{\alpha} = \text{argmax}_{\alpha \in S_{\alpha k}} \{\text{loglikelihood}(\Xi^o, \mu_{\boldsymbol{\omega}_k}, \sigma_{\boldsymbol{\omega}_k}, \alpha_k)\}. \quad (12)$$

The third is the **minimum distance** criterion, where we seek the best $\hat{\alpha}$ that minimizes the difference between the observed trajectory Ξ_t^o and the predicted trajectory for the first n_o data points:

$$\hat{\alpha} = \text{argmin}_{\alpha \in S_{\alpha k}} \left\{ \sum_{t=1}^{n_o} |\Xi_t^o - \Phi_{\alpha t} \mu_{\boldsymbol{\omega}_k}| \right\}. \quad (13)$$

The fourth method is based on a **model**: we assume that there is a correlation between α and the variation of the trajectory δ_{n_o} from the beginning until the time n_o. This "variation" δ_{n_o} can be computed as the variation of the position, e.g., $\delta_{n_o} = X(n_o) - X(1)$, or the variation in the entire trajectory, $\delta_{n_o} = \Xi(n_o) - \Xi(1)$, or any other measure of progress, if this hypothesis is appropriate for the type of task trajectories of the application.[10] Indeed, the α can be linked also to the movement speed, which can be roughly approximated by $\dot{X} = \frac{\delta X}{t_f} \left(\dot{\Xi} = \frac{\delta \Xi}{t_f} \right)$. We model the mapping between δ_{n_o} and α by the following equation:

$$\alpha = \Psi(\delta_{n_o})^\top \boldsymbol{\omega}_\alpha + \epsilon_\alpha, \quad (14)$$

where Ψ are RBFs, and ϵ_α is a zero-mean Gaussian noise. During learning, we compute the $\boldsymbol{\omega}_\alpha$ parameter, using the same method as in equation (3). During the inference, we compute $\hat{\alpha} = \Psi(\delta_{n_o})^\top \boldsymbol{\omega}_\alpha$.

[10] In our case, this assumption can be appropriate, because the reaching trajectories in our application are generally monotonic increasing/decreasing.

A comparison of the four methods for estimating α on a test study with iCub in simulation is presented in Section 6.6.

There exist other methods in the literature for computing α. For example, Ewerton et al. (2015) propose a method that models local variability in the speed of execution. In Maeda et al. (2016), they use a method that improves Dynamic Time Warping by imposing a smooth function on the time alignment mapping using local optimization. These methods will be implemented in the future works.

3.5. Recognizing One among Many Movement Primitives

Robots should not learn only one skills but many: different skills for different tasks. In our framework, tasks are represented by movement primitives, precisely ProMP. So it is important for the robot to be able to learn K different ProMPs and then be able to recognize from the early observations of a trajectory which of the K ProMPs the observations belong to.

During the learning step of a movement primitive $k \in [1 : K]$, the robot observes different trajectories $S_k = \{\Xi_1, \ldots, \Xi_n\}$. For each ProMP, it learns the distribution over the parameters vector $p(\boldsymbol{\omega}) \sim \mathcal{N}(\mu_{\boldsymbol{\omega}_k}, \Sigma_{\boldsymbol{\omega}_k})$, using equation (3). Moreover, the robot records the different phases of all the observed trajectories: $S_{\alpha k} = \{\alpha_{1k}, \ldots, \alpha_{nk}\}$.

After having learned these K ProMPs, the robot can use this information to autonomously execute a task trajectory. Since we are targeting collaborative movements, performed together with a partner at least at the beginning, we want the robot to be able to recognize from the first observations of a collaborative trajectory which is the current task that the partner is doing and what is the intention of the partner. Finally, we want the robot to be able to complete the task on its own, once it has recognized the task and predicted the future trajectory.

Let $\Xi^o = [\Xi_1 \ldots \Xi_{n_o}]^\mathrm{T}$ be the early observations of an initiated trajectory.

From these partial observations, the robot can recognize the "correct" (i.e., most likely) ProMP $\hat{k} \in [1 : K]$. First, for each ProMP $k \in [1 : K]$, it computes the most likely phase (time modulation factor) $\hat{\alpha}_k$ (as explained in Section 3.4), to obtain the set of ProMPs with the most likely duration: $S_{[\mu_{\boldsymbol{\omega}_k}, \hat{\alpha}_k]} = \{(\mu_{\boldsymbol{\omega}_1}, \hat{\alpha}_1), \ldots, (\mu_{\boldsymbol{\omega}_K}, \hat{\alpha}_K)\}$.

Then we compute the most likely ProMP $\hat{k}$ in $S_{[\mu_{\boldsymbol{\omega}_k}, \hat{\alpha}_k]}$ according to some criterion. One possible way is to minimize the distance between the early observations and the mean of the ProMP for the first portion of the trajectory:

$$\hat{k} = \arg \min_{k \in [1:K]} \left[\frac{1}{n_o} \sum_{t=1}^{n_o} |\Xi_t - \Phi_{\hat{\alpha}_k t} \mu_{\boldsymbol{\omega}_k}| \right]. \quad (15)$$

In equation (15), for each ProMP $k \in [1 : K]$, we compute the average distance between the observed early-trajectory Ξ_t and the mean trajectory of the ProMP $\Phi_{\hat{\alpha}_k t} \mu_{\omega_k}$, with $t = [1 : n_o]$. The most likely ProMP $\hat{k}$ is selected by computing the minimum distance (arg min). Other possible methods for estimating the most likely ProMPs could be inspired by those presented in the previous section for estimating the time modulation, i.e., maximum likelihood or learned models.

Once identified the $\hat{k}$-th most likely ProMP, we update its posterior distribution to take into account the initial portion of the observed trajectory, using equation (8):

$$\begin{cases} \hat{\mu}_{\boldsymbol{\omega}_{\hat{k}}} &= \mu_{\boldsymbol{\omega}_{\hat{k}}} + K\left(\Xi^o - \Phi_{\hat{\alpha}_{\hat{k}}[1:n_o]}\mu_{\boldsymbol{\omega}_{\hat{k}}}\right) \\ \hat{\Sigma}_{\boldsymbol{\omega}_{\hat{k}}} &= \Sigma_{\boldsymbol{\omega}_{\hat{k}}} - K\left(\Phi_{\hat{\alpha}_{\hat{k}}[1:n_o]}\Sigma_{\boldsymbol{\omega}_{\hat{k}}}\right) \\ K &= \Sigma_{\boldsymbol{\omega}_{\hat{k}}}\Phi^{\top}_{\hat{\alpha}_{\hat{k}}[1:n_o]}\left(\Sigma_{\xi^o} + \Phi_{\hat{\alpha}_{\hat{k}}[1:n_o]}\Sigma_{\boldsymbol{\omega}_{\hat{k}}}\Phi^{\top}_{\hat{\alpha}_{\hat{k}}[1:n_o]}\right)^{-1} \end{cases} \tag{16}$$

with $\hat{\alpha}_{\hat{k}}[1:n_o] = \hat{\alpha}_{\hat{k}}\, t$ (in matrix form), with $t \in [1:n_o]$.

Finally, the inferred trajectory is given by the following equation:

$$\forall t \in [1:\hat{t}_f], \hat{\xi}(t) = \Phi_t\, \hat{\mu}_{\boldsymbol{\omega}_{\hat{k}}}$$

with the expected duration of the trajectory $\hat{t}_f = \hat{\alpha}_{\hat{k}}\bar{s}$. The robot is now able to finish the movement executing the most likely "future" trajectory $\hat{\Xi} = [\hat{\xi}_{n_o+1} \ldots \hat{\xi}_{t_f}]^{\mathrm{T}}$.

4. SOFTWARE OVERVIEW

In this section, we introduce our open-source software with an overview of its architecture. This software is composed of two main modules, represented in **Figure 5**.

While the robot is learning the Probabilistic Movement Primitives (ProMPs) associated with the different tasks, the robot is controlled by its user. The user's guidance can be either manual for the real iCub, or through a haptic device for the simulated robot.

A Matlab module allows replaying movement primitives or finishing a movement that has been initiated by its user. By using this module, the robot can learn distributions over trajectories, replay movement primitives (using the mean of the distribution), recognize the ProMP that best matches a current trajectory, and infer the future evolution (until the end target) of this trajectory.

A C++ module forwards to the robot the control that comes either from the user or from the Matlab module. Then, the robot is able to finish a movement initiated by its user (directly or through a haptic device) in an autonomous way, as shown in **Figure 1**.

We present the C++ module in Section 6.2 and the theoretical explanation of the Matlab module algorithms in Section 3. A guide to run this last module is first presented in Section 5 for a simple example, and in Section 6 for our application, where a simulated robot learns many measured information of the movements. Finally, we present results on the real iCub application in Section 7.

Our software is available through the GPL license, and publicly available at:

https://github.com/inria-larsen/icubLearningTrajectories.

Tutorial, readme, and videos can be found in that repository. First, the readme file describes how to launch simple demonstrations of the software. Videos present these demonstrations to simplify the understanding. In the next sections, we detail the operation of the demo program for a first case of 1-DOF primitive, followed by the presentation of the specific applications on the iCub (first simulated and then real).

5. SOFTWARE EXAMPLE: LEARNING A 1-DOF PRIMITIVE

In this section, we present the use of the software to learn ProMPs in a simple case of 1-DOF primitive. This example only uses the *MatlabProgram* folder, composed of:

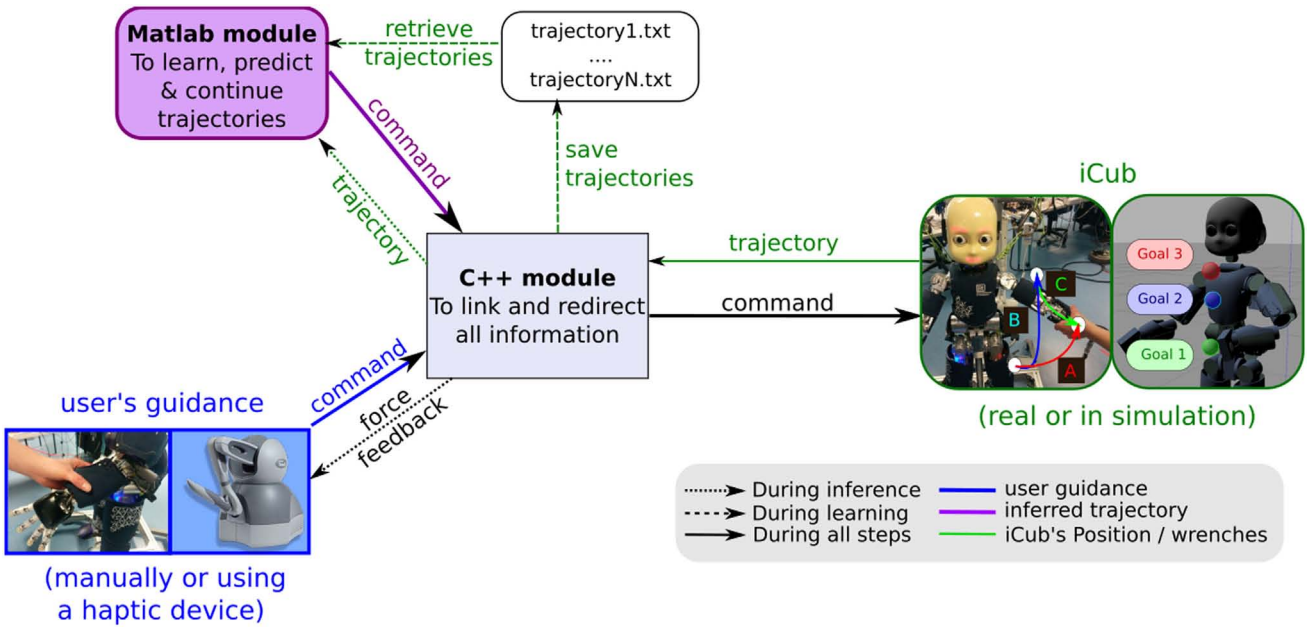


FIGURE 5 | Software architecture and data flows. The robot's control is done either by the user's guidance (manually or through a haptic device) represented in blue, or by the Matlab module, in purple. The C++ module handles the control source to command the robot, as represented in black. Moreover, this module forwards information that comes from the iCub.

- A sub-folder called "Data," where there are trajectory sets used to learn movement primitives. These trajectories are stored in text files with the following information:
  - **input parameters**: # input_1 # input_2 [. . .]
  - **input parameters with time step**: # timeStep # input_1 # input_2 [. . .]
  - ***recordTrajectories.cpp** program recording*: See Section 6.3 for more information.
- A sub-folder called "used_functions." It contains all the functions used to retrieve trajectories, compute ProMPs, infer trajectories, and plot results. Normally, using this toolbox does not require understanding these functions. The first lines of these functions give an explanation of their functioning and precise what are the input(s) and output(s) parameters.
- Matlab scripts called "demo_*.m." They are simple examples of how to use this toolbox.

The script `demo_plot1DOF.m`, can be used to compute a ProMP and to continue an initiated movement. The ProMP is computed from a dataset stored in a ".mat" file, called *traj1_1DOF.mat*. In this script, variables are first defined to make the script specific to the current dataset:

Variable assignation	Commentary
DataPath = 'Datatraj1_1DOF.mat';	Can be either ".mat" or ".txt". In the current demo, you can also write DataPath = 'Data/traj1' if you want to use the text files of this dataset.
typeRecover = '.mat'	Or .txt, it depends on your choice of data file.
inputName = {'z[m]'};	Label of your input(s). Here *z* represents the *z*-axis Cartesian coordinate.
s_ref = 100;	Number of samples used as reference to rescale all the trajectories to the same duration.
nbInput = 1;	Dimension of the generic vector containing the state of the trajectory.
M = 5;	Number of radial basis functions per input.
expNoise = 0.00001;	Expected trajectory noise.
percentData = 20;	Percent of observed data during the inference.

The variables include the following:

- `DataPath` is the path to the recorded data. If the data are stored in text files, this variable contains the folder name where text files are stored. These text files are called "record*X*.txt," with $X \in [0: n-1]$ if there are n trajectories. One folder is used to learn one ProMP. If the data are already loaded from a ".mat" file, write the whole path with the extension. The data in ".mat" match with the output of the Matlab function `loadTrajectory`.
- `nbInput` $= D$ is the dimension of the input vector ξ_t.
- `expNoise` $= \Sigma_\xi^o$ is the expected noise of the initiated trajectory. The smaller this variable is, the stronger the modification of the ProMP distribution will be, given new observations.

We will now explain more in detail the script. To recover data recorded in a ".txt" file, we call the function:

```
t{1} = loadTrajectory(PATH, nameT, varargin)
```

Its input parameters specify the path of the recorded data, the label of the trajectory. Other information can be added by using the `varargin` variable (for more detail, check the header of the function with the help comments). The output is an object that contains all the information about the demonstrated trajectories. It is composed of `nbTraj`, the number of trajectory; `realTime`, the simulation time; and `y` (and `yMat`), the vector (and matrix) trajectory set. Thus, `t{1}.y{i}` contains the *i*-th trajectory.

The Matlab function `drawRecoverData(t{1}, inputName,'namFig', nFig, varargin)` plots in a Matlab figure (numbered `nFig`) the dataset of loaded trajectories. An example is shown in **Figure 2**, on the left. Incidentally, the different duration of the trajectories is visible: on average, it is 1.17 ± 0.42 s.

To split the entire dataset of demonstrated trajectories `t{1}` into a training dataset (used for learning the ProMPs) and a test dataset (used for the inference), call the function

```
[train, test] = partitionTrajectory(t{1},
partitionType, percentData, s_ref)
```

where if `partitionType` $= 1$, only one trajectory is in the test set and the others are placed in the training set, and if `partitionType` > 1 it corresponds to the percentage of trajectories that will be included in the training set.

The ProMP can be computed from the training set by using the function:

```
promp = computeDistribution(train, M, s_ref, c, h)
```

The output variable `promp` is an object that contains all the ProMP information. The first three input parameters have been presented before: `train` is the training set, `M` is the number of RBFs, and `s_ref` is the number of samples used to rescale all the trajectories. The last two input parameters `c` and `h` shape the RBFs of the ProMP model: $\mathtt{c} \in \mathbb{R}^M$ is the center of the Gaussians and $\mathtt{h} \in \mathbb{R}$ their variance.

To visualize this ProMP, as shown in **Figure 2**, call the function:

```
drawDistribution(promp, inputName, s_ref)
```

For debugging purposes and to understand how to tune the ProMPs' parameters, it is interesting to plot the overlay of the basis functions in time. Choosing an appropriate number of basis functions is important, as too few may be insufficient to approximate the trajectories under consideration, and too many could result in overfitting problems. To plot the basis functions, simply call:

```
drawBasisFunction(promp.PHI, M)
```

where `promp.PHI` is a set of RBFs evaluated in the normalized time range $t \in [1 : \bar{s}]$.

Figure S1 in Supplementary Material shows at the top the basis functions before normalization, and at the bottom the ProMP modeled from these basis functions. From left to right, we change the number of basis functions. When there are not enough basis functions (left), the model is not able to correctly represent the shape of the trajectories. In the middle, the trajectories are well represented by the five basis functions. With more basis functions (right), the variance of the distribution decreases because

the model is more accurate. However, arbitrarily increasing the number of basis functions is not a good idea, as it may not improve the accuracy of the model and worse it may cause overfitting.

Once the ProMP is learned, the robot can reproduce the movement primitive using the mean of the distribution. Moreover, it can now recognize a movement that has been initiated in this distribution and predict how to finish it. To do so, given the early n_o observations of a movement, the robot updates the prior distribution to match the early observed data points: through conditioning, it finds the posterior distribution, which can be used by the robot to execute the movement on its own.

The first step in predicting the evolution of the trajectory is to infer the duration of this trajectory, which is encoded by the time modulation parameter $\hat{\alpha}$. The computation of this inference, which was detailed in Section 3.4, can be done by using the function:

```
[expAlpha, type, x] = inferenceAlpha(promp,
test{1}, M, s_ref, c, h, test{1}.nbData,...
expNoise, typeReco)
```

where `typeReco` is the type of criteria used to find the expected time modulation ("MO," "DI," or "ML" for model, distance or maximum likelihood methods); `expAlpha` $= \hat{\alpha}$ is the expected time modulation; `type` is the index of the ProMP from which `expAlpha` has been computed, which we note in this article as k. To predict the evolution of the trajectory, we use equation (8) from Section 3.3. In Matlab, this is done by the function:

```
infTraj = inference(promp, test{1}, M, s_ref, c,
h,... test{1}.nbData, expNoise, expAlpha).
```

where `test{1}.nbData` has been computed during the `partitionTrajectory` step. This variable is the number of observations n_o, representing the percentage of observed data (`percentData`) of the test trajectory (i.e., to be inferred) that the robot observes. `infTraj`$= \hat{\Xi}$ is the inferred trajectory. Finally, to draw the inferred trajectory, we can call the function `drawInference(promp, inputName, infTraj, test1, s_ref)`.

It can be interesting to plot the quality of the predicted trajectories as a function of the number of observations, as done in **Figure 3**.

Note that when we have observed a larger portion of the trajectory, the prediction of the remaining portion is more accurate.

Now we want to measure the quality of the prediction. Let $\Xi^* = [\xi^o(1), \ldots, \xi^o(n_o), \xi^*(n_o+1), \ldots, \xi^*(t_f^*)]$ be the real trajectory expected by the user. To measure the quality of the prediction, we can use:

- The likelihood of having the Ξ^* trajectory given the updated distribution $p(\hat{\boldsymbol{\omega}})$.
- The distance between the Ξ^* trajectory and the $\hat{\Xi}$ inferred trajectory.

However, according to the type of recognition `typeReco` used to estimate the time modulation parameter α from the early observations, a visible mismatch between the predicted trajectory and the real one can be visible even when a lot of observations are used. This is due to the error of the expectation of this time

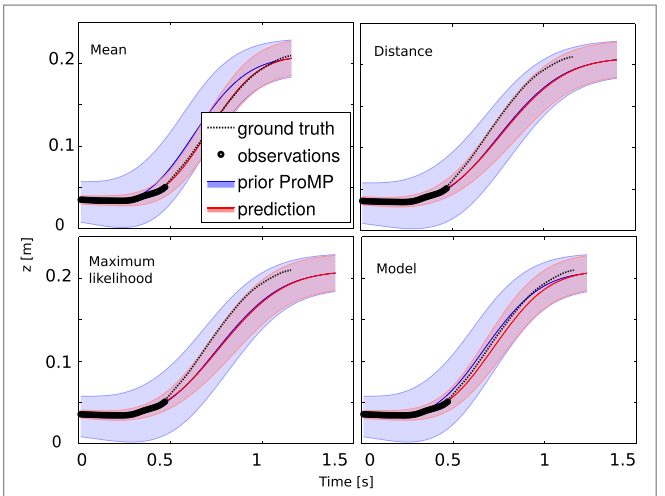


FIGURE 6 | The prediction of the future trajectory given $n_o = 40\%$ of early observations from the learned ProMP computed for the test dataset (**Figure 2**). The plots show the predicted trajectory, using different criteria to estimate the best phases of the trajectory: using the average time modulation (equation (11)); using the distance criteria (equation (13)); using the maximum log-likelihood (equation (12)); or using a model of time modulation according to the time variation (equation (14)).

TABLE 2 | Information about trajectories' duration.

	Traj. samples	$\alpha = \frac{\bar{s}}{\text{Iterations}}, \bar{s} = 100$	Duration [s]
Min	83	1.2048	0.83
Max	115	0.8696	1.15
Mean	100	1	0.99
SD	9	11.1111	0.09

modulation parameter. In Section 3.4, we present the different methods used to predict the trajectory duration. These methods select the most likely $\hat{\alpha}$ according to different criteria: distance; maximum likelihood; model of the α variable[11]; and average of the observed α during learning.

Figure 6 shows the different trajectories predicted after $n_o = 40\%$ of the length of the desired trajectory is observed, according to the method used to estimate the time modulation parameter.

On this simple test, where the variation time is little as shown in **Table 2**, the best result is accomplished by the average of time modulation parameter of the trajectories used during the learning step. In more complicated cases, when the time modulation varies, the other methods will be preferable as seen in Section 3.5.

6. APPLICATION ON THE SIMULATED iCub: LEARNING THREE PRIMITIVES

In this application, the robot learns multiple ProMPs and is able to predict the future trajectory of a movement initiated by the user, assuming the movement belongs to one of the learned primitives.

[11] In this model, we assume that we can find the time modulation parameter according to the global variation of the position during the n_o first observed data.

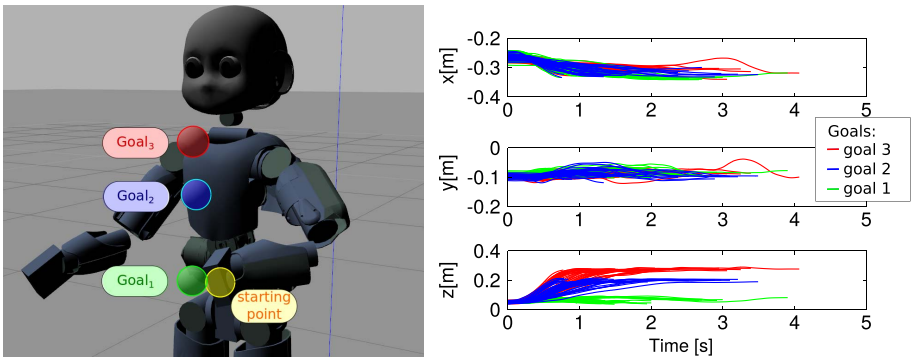


FIGURE 7 | Left: the three colored targets that the robot must reach from the starting point; the corresponding trajectories are used to learn three primitives representing three skills. Right: the Cartesian position information of the demonstrated trajectories for the three reaching tasks.

Based on this prediction, it can also complete the movement once it has recognized the appropriate ProMP.

We simplify the three actions/tasks by reaching three different targets, represented by three colored balls in the reachable workspace of the iCub. The example is performed with the simulated iCub in Gazebo. **Figure 7** shows the three targets, placed at different heights in front of the robot.

In Section 6.1, we formulate the intention recognition problem for the iCub: the problem is to learn the ProMP from trajectories consisting of Cartesian positions in 3D[12] and the 6D wrench information measured by the robot during the movement. In Section 6.2, we describe the simulated setup of iCub in Gazebo, then in Section 6.3, we explain how trajectories are recorded, including force information, when we use the simulated robot.

6.1. Predicting Intended Trajectories by Using ProMPs

The model is based on Section 3, but here we want to learn more information during movements. We record this information in a multivariate parameter vector:

$$\forall t, \xi_t = \begin{bmatrix} X_t \\ F_t \end{bmatrix} \in \mathbb{R}^9,$$

where $X_t \in \mathbb{R}^3$ is the Cartesian position of the robot's end effector and $F_t \in \mathbb{R}^6$ the external forces and moments. In particular, F_t contains the user's contact forces and moments. Let us call $\dim(\xi_t) = D$, the dimension of this parameter vector.

The corresponding ProMP model is as follows:

$$\xi_t = \begin{bmatrix} X_t \\ F_t \end{bmatrix} = \Phi_{\alpha t}\boldsymbol{\omega} + \epsilon_t,$$

where $\boldsymbol{\omega} \in \mathbb{R}^{D \cdot M}$ is the time-independent parameter vector, $\epsilon_t = \begin{bmatrix} \epsilon_{X_t} \\ \epsilon_{F_t} \end{bmatrix} \in \mathbb{R}^D$ is the zero-mean Gaussian i.i.d. observation noise, and $\Phi_{\alpha t} \in \mathbb{R}^{D \times D \cdot M}$ a matrix of Radial Basis Functions (RBFs) evaluated at time αt.

Since we are in the multidimensional case, this $\Phi_{\alpha t}$ block diagonal matrix is defined as follows:

$$\Phi_{\alpha t} = BlockdiagonalMatrix(\phi_1, \ldots, \phi_D) \in R^{D \times D \cdot M}.$$

It is a diagonal matrix of D Radial Basis Functions (RBFs), where each RBF represents one dimension of the ξ_t vector and it is composed of M Gaussians, spread over same number of samples $\bar{s}$.

6.1.1. Learning Motion Primitives

During the learning step of each movement primitive $k \in [1:3]$, the robot observes different trajectories $S_k = \{\Xi_1, \ldots, \Xi_n\}_k$, as presented in Section 6.3.

For each trajectory $\Xi_{i[1:t_{fi}]} = [\xi_{i(1)}, \ldots, \xi_{i(tf_i)}]^{\mathrm{T}}$, it computes the optimal $\boldsymbol{\omega}_{ki}$ parameter vector that best approximates the trajectory.

We saw in Section 3.5 how these computations are done. In our software, we use matrix computation instead of t_{fi} iterative ones done for each observation t (as in equation (3)). Thus, we have the following:

$$\boldsymbol{\omega}_{ki} = \left(\Phi^{\top}_{\alpha[1:t_{fi}]}\Phi_{\alpha[1:t_{fi}]}\right)^{-1}\Phi^{\top}_{\alpha[1:t_{fi}]} * \Xi_{i[1:t_{fi}]} \tag{17}$$

with $\Phi_{\alpha[1:t_{fi}]} = [\Phi_{\alpha 1}, \Phi_{\alpha 2} \ldots, \Phi_{\alpha t_{fi}}]^{\mathrm{T}}$.

6.1.2. Prediction of the Trajectory Evolution from Initial Observations

After having learned the three ProMPs, the robot is able to finish an initiated movement on its own. In Sections 3.3–3.5, we explained how to compute the future intended trajectory given the early observations.

In this example, we add specificities about the parameters to learn.

Let $\Xi^o = \begin{bmatrix} X^o \\ F^o \end{bmatrix} = [\Xi_1 \ldots \Xi_{n_o}]^{\mathrm{T}}$ be the early observations of the trajectory.

First, we only consider the partial observations: $X^o = [X_1 \ldots X_{n_o}]^{\mathrm{T}}$. Indeed, we assume the recognition of a trajectory is done with Cartesian position information only, because the

[12] Note that in that particular example we do not use the orientation because we want the robot's hand to keep the same orientation during the movement. But in principle, it is possible to learn trajectories consisting of the 6D/7D Cartesian position and orientation of the hand, to make the robot change also the orientation of the hand during the task.

same movement can be done and recognized with different force profiles than the learned ones.

From this partial observation X^o, the robot recognizes the current ProMP $\hat{k} \in [1:3]$, as seen in Section 3.5. It also computes an expectation of the time modulation $\hat{t}_f$, as seen in Section 3.4. Using the expected value of the time modulation, it approximates the trajectory speed and its total time duration.

Second, we use the total observation Ξ^o to update the ProMP, as seen in Section 3.3. This computation is based on equation (18), but here again, we use the following matrix computation:

$$\begin{cases} \hat{\mu}_{\boldsymbol{\omega}_k} &= \mu_{\boldsymbol{\omega}_k} + K\big(\Xi^o - \Phi_{\alpha[1:n_o]}\mu_{\boldsymbol{\omega}_k}\big) \\ \hat{\Sigma}_{\boldsymbol{\omega}_k} &= \Sigma_{\boldsymbol{\omega}_k} - K\big(\Phi_{\alpha[1:n_o]}\Sigma_{\boldsymbol{\omega}_k}\big) \\ K &= \Sigma_{\boldsymbol{\omega}_k}\Phi^{\mathrm{T}}_{\alpha[1:n_o]}\Big(\Sigma_{\xi^o} + \Phi_{\alpha[1:n_o]}\Sigma_{\boldsymbol{\omega}_k}\Phi^{\mathrm{T}}_{\alpha[1:n_o]}\Big)^{-1}. \end{cases}$$

From this posterior distribution, we retrieve the inferred $\hat{\Xi} = \{\hat{\xi}_1, \ldots, \hat{\xi}_{\hat{t}_f}\}$ trajectory, with:

$$\forall t \in [1:\hat{t}_f], \hat{\xi}_t = \begin{bmatrix} \hat{X}_t \\ \hat{F}_t \end{bmatrix} = \Phi_{\alpha t}\hat{\mu}_{\boldsymbol{\omega}_k}.$$

Note that the inferred wrenches $\hat{F}_t$, here, correspond to the simulated wrenches in Gazebo. In this example, there is little use for them in simulation; the interest for predicting also wrenches will be clearer in Section 7, with the example on the real robot.

6.2. Setup for Simulated iCub

For this application, we created a prototype in Gazebo, where the robot must reach three different targets with the help of a human. To interact physically with the robot simulated in Gazebo, we used the Geomagic touch, a haptic device.

The setup consists of the following:

- the iCub simulation in Gazebo, complete with the dynamic information provided by *wholeBodyDynamicsTree* (https://github.com/robotology/codyco-modules/tree/master/src/modules/wholeBodyDynamicsTree) and the Cartesian information provided by *iKinCartesianController*;
- the Geomagic Touch, installed following the instructions in https://github.com/inria-larsen/icub-manual/wiki/Installation-with-the-Geomagic-Touch, which not only install the SDK and the drivers of the GeoMagic but also point to how to create the yarp drivers for the Geomagic;
- a C++ module (https://github.com/inria-larsen/icubLearningTrajectories/tree/master/CppProgram) that connects the output command from the Geomagic to the iCub in Gazebo and eventually enables recording the trajectories on a file. A tutorial is included in this software.

The interconnection among the different modules is represented in **Figure 5**, where the Matlab module is not used. The tip of the Geomagic is virtually attached to the end effector of the robot:

$$x_{geo} \longrightarrow x_{icub\_hand}.$$

When the operator moves the Geomagic, the position of the Geomagic tip x_{geo} is scaled (1:1 by default) in the iCub workspace as x_{icub_hand}, and the Cartesian controller is used to move the iCub hand around a "home" position, or default starting position:

$$x_{icub\_hand} = hapticDriverMapping(x_0 + x_{geo}),$$

where *hapticDriverMapping* is the transformation applied by the haptic device driver, which essentially maps the axis from the Geomagic reference frame to the iCub reference frame. By default, no force feedback is sent back to the operator in this application, as we want to emulate the zero-torque control mode of the real iCub, where the robot is ideally transparent and not opposing any resistance to the human guidance. A default orientation of the hand ("katana" orientation) is set.

6.3. Data Acquisition

The dark button of the Geomagic is used to start and stop the recording of the trajectories. The operator must click and hold the button during the whole movement and release the button at the end. The trajectory is saved on a file called *recordX.txt* for the *X*-th trajectory. The structure of this file is:

```
1 #time #xgeo #ygeo #zgeo #fx #fy #fz #mx #my #mz #x_icub_hand
  #y_icub_hand #z_icub_hand
2 5.96046e-06 -0.0510954 -0.0127809 -0.0522504 0.284382
  -0.0659538 -0.0239582 -0.0162418 -0.0290078 -0.0607215
  -0.248905 -0.0872191 0.0477496$
```

A video showing the iCub's arm moved by a user through the haptic device in Gazebo is available in Section 8 (tutorial video). The graph in **Figure 7** represents some trajectories recorded with the Geomagic, corresponding to lifting the left arm of the iCub.

Demonstrated trajectories and their corresponding forces can be recorded directly from the robot, by accessing the Cartesian interface and the *wholeBodyDynamicsTree* module.[13]

In our project on Github, we provide the acquired dataset with the trajectories for the interested reader who wishes to test the code with these trajectories. Two datasets are available at https://github.com/inria-larsen/icubLearningTrajectories/tree/master/MatlabProgram/Data/: the first dataset called "heights" is composed of three goal-directed reaching tasks, where the targets vary in height; the second dataset called "FLT" is composed of trajectories recorded on the real robot, whose arms move forward, to the left and to the top.

A Matlab script that learns ProMPs with such kinds of datasets is available in the toolbox, called `demo_plotProMPs.m`. It contains all the following steps.

To load the first "heights" dataset with the three trajectories, write:

```
1 t{1}=loadTrajectory('Data/heights/bottom', 'bottom', 'refNb',
  s_bar, 'nbInput', nbInput, ... 'Specific', 'FromGeom');
2 t{2}=loadTrajectory('Data/heights/top', 'top', 'refNb',
  s_bar, 'nbInput', nbInput, ... 'Specific', 'FromGeom');
3 t{3}=loadTrajectory('Data/heights/middle', 'forward',
  'refNb', s_bar, 'nbInput', nbInput, ... 'Specific',
  'FromGeom');
```

[13] In our example, we do not use the simulated wrench information as it is very noisy. However, we provide the code and show how to retrieve it and use it, in case the readers should not have access to the real iCub.

Figure 7 shows the three sets of demonstrated trajectories. In the used dataset called "heights," we have recorded 40 trajectories per movement primitive.

6.4. Learning the ProMPs

We need to first learn the ProMPs associated with the three observed movements. First, we partition the collected dataset into a training set and test dataset for the inference. One random trajectory for the inference is used:

```
[train{i}, test{i}] = partitionTrajectory(t{i}, 1, percentData, s_bar);
```

The second input parameter specifies that we select only one trajectory, randomly selected, to test the ProMP.

Now, we compute the three ProMPs with:

```
promp{1} = computeDistribution(train{1}, M, s_bar, c, h);
promp{2} = computeDistribution(train{2}, M, s_bar, c, h);
promp{3} = computeDistribution(train{3}, M, s_bar, c, h)
```

We set the following parameters:

- `s_bar` = 100: reference number of samples, which we note in this article as $\bar{s}$.
- `nbInput(1)` = 3; `nbInput(2)` = 6: dimension of the generic vector containing the state of the trajectories. It is composed of 3D Cartesian position and 6D forces and wrench information.[14]
- `M(1)` = 5; `M(2)` = 5: number of basis functions for each `nbInput` dimension.
- `c` = `1/M;h` = `1/(M*M)`: RBF parameters (see equation (2)).
- `expNoise` = `0.00001`: the expected data noise. We assume this noise to be very low, since this is a simulation.
- `percentData` = 40: this variable specifies the percentage of the trajectory that the robot will be observed, before inferring the end.

These parameters can be changed at the beginning of the Matlab script.

Figure 8 shows the three ProMPs of the reaching movements toward the three targets. To highlight the most useful dimension, we only plot the z-axis Cartesian position. However, each dimension is plotted by the Matlab script with:

```
drawRecoverData(t{1}, inputName, 'Specolor', 'b', 'namFig', 1);
drawRecoverData(t{1}, inputName, 'Interval', [4 7 5 8 6 9], 'Specolor', 'b', 'namFig', 2);
drawRecoverData(t{2}, inputName, 'Specolor', 'r', 'namFig', 1);
drawRecoverData(t{2}, inputName, 'Interval', [4 7 5 8 6 9], 'Specolor', 'r', 'namFig', 2);
drawRecoverData(t{3}, inputName, 'Specolor', 'g', 'namFig', 1);
drawRecoverData(t{3}, inputName, 'Interval', [4 7 5 8 6 9], 'Specolor', 'g', 'namFig', 2);
```

[14] Note that in our example wrenches are separated from the Cartesian position, because they are not used to recognize the index of the current ProMP during the inference.

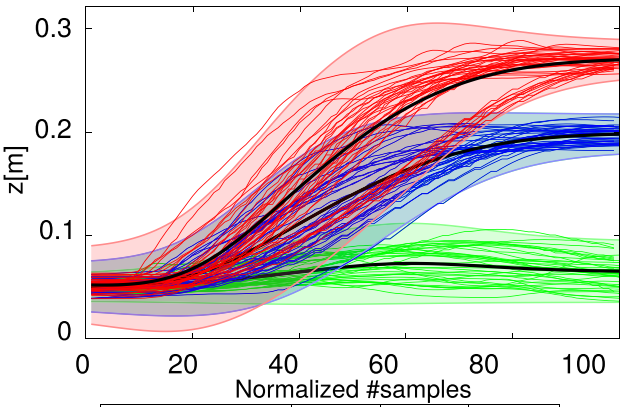


FIGURE 8 | The Cartesian position in the z-axis of the three ProMPs corresponding to reaching three targets. There are 39 trajectory demonstrations per each ProMPs with M = 5 basis functions, $c = \frac{1}{M}, h = \frac{1}{M^2}$ and $\bar{s} = 100$.

6.5. Predicting the Desired Movement

Now that we have learned the different ProMPs, we can predict the end of a trajectory according to the early observation n_o. This number is computed from the variable `percentData` written at the beginning of the trajectory by: $n_o = \lfloor \frac{percentData}{100} * t_{fi} \rfloor$, where i is the index of the test trajectory.

To prepare the prediction, the model the time modulation of each trajectory is computed with:

```
w = computeAlpha(test.nbData, t, nbInput);
promp1.w_alpha = w1;
promp2.w_alpha = w2;
promp3.w_alpha = w3;
```

This model relies on the global variation of Cartesian position during the first n_o observations. The model's computations are explained in Section 3.4.

Now, to estimate the time modulation of the trajectory, call the function:

```
[alphaTraj, type, x] = inferenceAlpha(promp, test{1}, M, s_bar, c, h, test{1}.nbData, expNoise, 'MO');
```

where `alphaTraj` contains the estimated time modulation $\hat{\alpha}$ and `type` gives the index of the recognized ProMP. The last parameter `x` is used for debugging purposes.

Using this estimation of the time modulation, the end of the trajectory is inferred with:

```
infTraj = inference(promp, test{1}, M, s_bar, c, h, test{1}.nbData, expNoise, alphaTraj);
```

As shown in the previous example, the quality of the prediction of the future trajectory depends on the accuracy of the time

modulation estimation. This estimation is computed by calling the function:

```
%Using the model:
[alphaTraj, type, x] = inferenceAlpha(promp, test{1}, M, s_bar,
  c, h, test{1}.nbData, expNoise, 'MO');
%Using the distance criteria:
[alphaTraj, type, x] = inferenceAlpha(promp, test{1}, M, s_bar,
  c, h, test{1}.nbData, expNoise, 'DI');
%Using the Maximum likelihood criteria:
[alphaTraj, type, x] = inferenceAlpha(promp, test{1}, M, s_bar,
  c, h, test{1}.nbData, expNoise, 'ML');
%Using the mean of observed temporal modulation during learning:
alphaTraj = (promp{1}.mu_alpha + promp{2}.mu_alpha + promp{3}.
  mu_alpha)/3.0;
```

6.6. Predicting the Time Modulation

In Section 3.4, we presented four main methods for estimating the time modulation parameter, discussing why this is crucial for a better estimation of the trajectory. Here, we compare the methods on the three goals experiment. We recorded 40 trajectories for each movement primitive, for a total of 120 trajectories. After having computed the corresponding ProMPs, we tested the inference by providing early observations of a trajectory that the robot must finish. For that purpose, it recognizes the correct ProMP among the three precedently learned (see Section 3.5) and then it estimates the time modulation parameter $\hat{\alpha}$. **Figure 9** represents the average error of the $\hat{\alpha}$ during inference for 10 trials according to the number of observations (from 30 to 90% of observed data) and according to the used method. These methods are the ones we have just presented before that we called mean (equation (11)), maximum likelihood (equation (12)), minimum distance (equation (13)) or model (equation (14)). Each time, the tested trajectory is chosen randomly from the data set of observed trajectories (of course, the test trajectory does not belong to the training set, so it was not used in the learning step). The method that takes the average of α observed during learning is taken as comparison (in black). We can see that other methods are more accurate. The *maximum likelihood* is increasingly more accurate, as expected. The fourth method (*model*) that models the α according to the global variation of the trajectory's positions during the early observations is the best performing when the portion of observed trajectory is small (e.g., 30–50%). Since it is our interest to predict the future trajectory as early as possible, we adopted the *model* method for our experiments.

7. APPLICATION ON THE REAL iCub

In this section, we present and discuss two experiments with the real robot iCub.

In the first, we take inspiration from the experiment of the previous Section 6, where the "tasks" are exemplified by simple point-to-point trajectories demonstrated by a human tutor. In this experiment, we explore how to use wrench information and use known demonstrations as ground truth, to evaluate the quality of our prediction.

In the second experiment, we set up a more realistic collaborative scenario, inspired by collaborative object sorting. In such applications, the robot is used to lift an object (heavy, or dangerous, or that the human cannot manipulate, as for some chemicals or food), the human inspects the object and then decides if it is accepted or rejected. Depending on this decision, the object goes on a tray or bin in front of the robot, or on a bin located on the robot side. Dropping the objects in two cases must be done in a different way. Realizing this application with iCub is not easy, as iCub cannot lift heavy objects and has a limited workspace. Therefore, we simplify the experiment with small objects and two bins. The human simply starts the robots movement with physical guidance, and then the robot finishes the movement on its own. In this experiment the predicted trajectories are validated on-the-fly by the human operator.

In a more complex collaborative scenario, tasks could be elementary tasks such as pointing, grasping, reaching, and manipulating tools (the type of task here is not important, as long as it can be represented by a trajectory).

7.1. Three Simple Actions with Wrench Information

Task trajectories, in this example, have both position and wrench information. In general, it is a good idea to represent collaborative motion primitives in terms of both position and wrenches, as this representation enables using them in the context of physical interaction. Contrarily to the simulated experiment, here the inferred wrenches $\hat{F}_t$ correspond to the wrenches the robot should perceive if the partner was manually guiding the robot to perform the entire movement: indeed, these wrenches are computed from the demonstrations used to learn the primitive. The predicted wrenches can be used in different ways, depending on the application. For example, if the partner breaks the contact with the robot, the perceived wrenches will be different. If the robot is not equipped with tactile or contact sensors, this information can be used by the robot to "perceive" the contact breaking and interpret it, for example, as the sign that the human wants the robot to continue the task on its own. Another use for the demonstrated wrenches is for detecting abnormal forces while the robot is moving: this use can have different applications, from adapting the motion to new environment to automatically detecting new demonstrations. Here, they are simply used to detect when the partner breaks the contact with the robot, and the latter must continue the movement on its own.

In the following, we present how to realize the experiment for predicting the user intention with the real iCub, using our software. The robot must learn three task trajectories represented in **Figure 10**. In red, the first trajectory goes from an initial position in front of the robot to its left (task A). In green, the second trajectory goes from the same initial position to the top (task C). In blue, the last trajectory goes from the top position to the position on the left (task B).

To provide the demonstrations for the tasks, the human tutor used three visual targets shown on the iCub_GUI, a basic module of the iCub code that provides a real-time synthetic and augmented view of the robot status, with arrows for the external forces and colored objects for the targets. One difficulty for novice

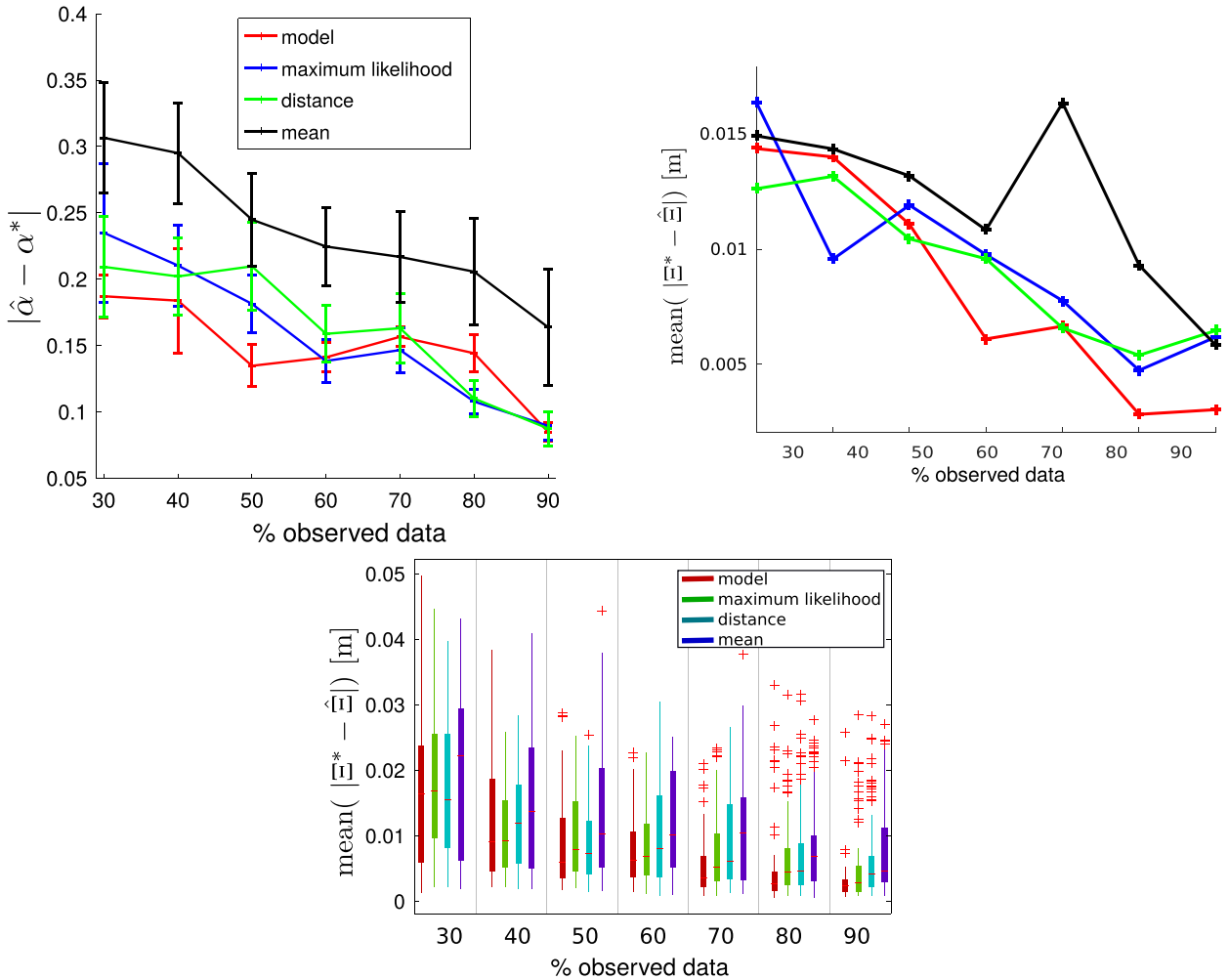


FIGURE 9 | (Top left) Error of α estimation; (top right and bottom) error of trajectory prediction according to the number of known data and the method used. We executed 10 different trials for each case.

users of iCub is to be able to drive the robot's arm making it perform desired complex 3D trajectories (Ivaldi et al., 2017), but after some practice in moving the robot's arm the operator recorded all the demonstrations. We want to highlight that having variations in the starting or ending points of the trajectories is not at all a problem, since the ProMPs are able to deal with this variability.

We will see that by using the ProMPs method and by learning the end-effector Cartesian position, the robot will be able to learn distributions over trajectories, recognize when a movement belongs to one of these distributions, and infer the end of the movement.

In this experiment, the robot received 10 demonstrated trajectories per movement primitive, all provided by the same user. We recorded the Cartesian end-effector position and the wrenches of the robot's left arm. Data are retrieved using the function `used_functions/retrieveRealDataWithoutOrientation.m`. The output parameters of this function are three objects (one per ProMP) that contain all the required information to learn the ProMPs.

In this function, the wrench information are filtered using a Matlab function called `envelope.m`[15]: for each trajectory `traj` and its subMatrix $M = F([1: t])$:

```
1 [envHigh, envLow] = envelope(traj.M);
2 traj.M = (envHigh + envLow)/2;
```

These three objects are saved in `'Data/realIcub.mat'`. A Matlab script called `demo_plotProMPsIcub.m` recovers these data, using the function `load('Data/realIcub.mat')`. This script follows the same organization as the ones we previously explained in Sections 5 and 6. By launching this script, the recovered data are plotted first.

Then, the ProMPs are computed and plotted, as presented in **Figure 11**. In this figure, the distributions are visibly overlaid:

- during the whole trajectories duration for the wrench information;

[15] Information about this function can be found here: https://fr.mathworks.com/help/signal/ref/envelope.html?requestedDomain=www.mathworks.com.

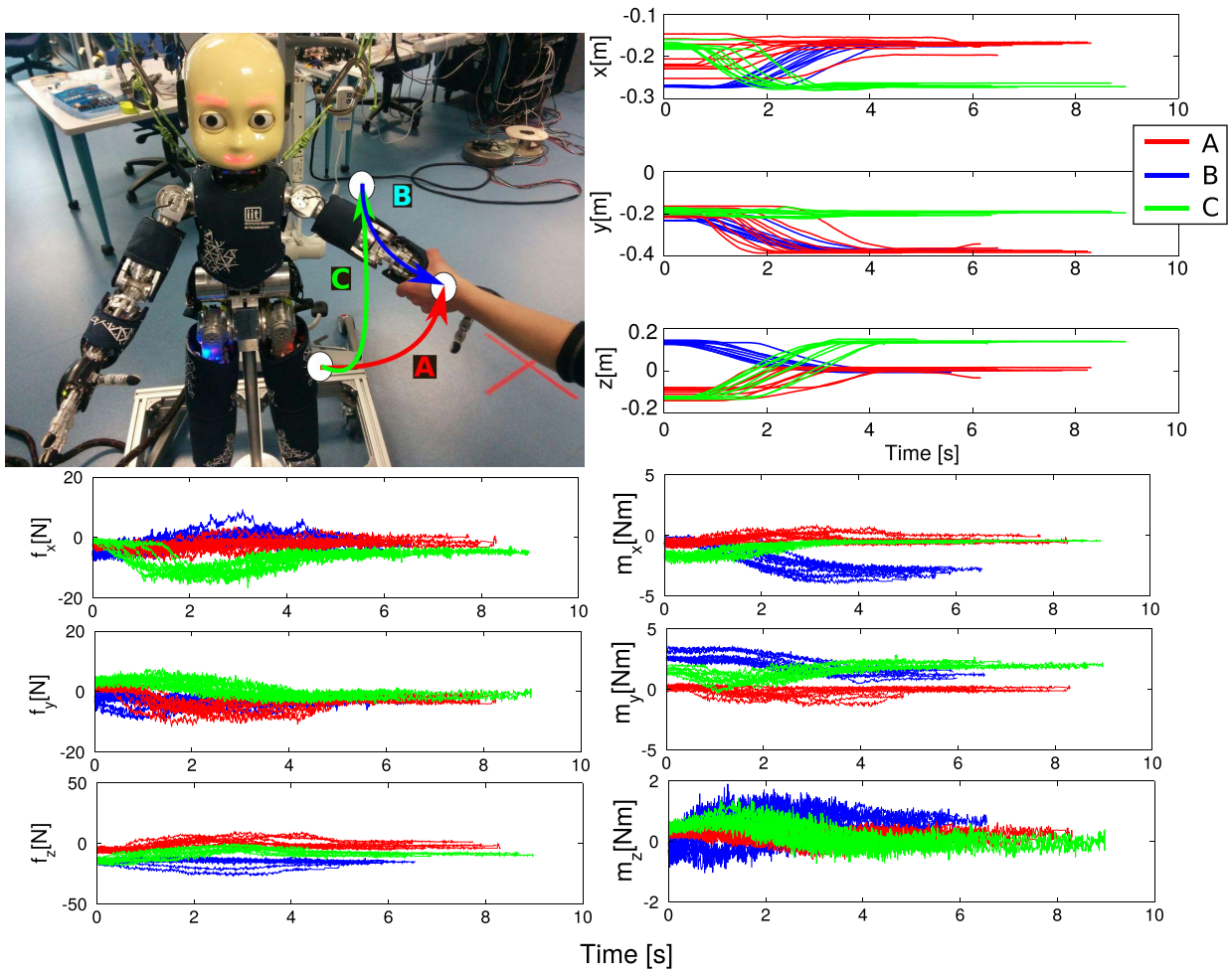


FIGURE 10 | Top left: the iCub and the visualization of the three targets in its workspace, defining the three tasks A–B–C. Top right: Cartesian position information of the demonstrated trajectories for the three tasks. Bottom left and right: wrench (force and moment) information of the demonstrated trajectories.

- during the 40% first samples of the trajectories for the Cartesian position information.

After this learning step, the user chooses which ProMP to test. Using a variable that represents the percentage of observed data to be used for the inference, the script computes the number of early observations n_o[16] that will be measured by the robot. Using this number, the robot models the time modulation parameter α[17] of each ProMP, as explained in Section 3.4. Using this model, the time modulation of the test trajectory is estimated, and the corresponding ProMP is identified.

Then, the inference of the trajectory's target is performed. **Figure 12** represents the inference of the three tested trajectories when wrench information is not used by the robot to infer the trajectory. To realize this figure, with the comparison between the predicted trajectory and the ground truth, we applied our algorithm offline. In fact, it is not possible at time t to have the ground truth of the trajectory intended by the human from $t+1$ to t_f: even if we would tell to the human in advance the goal that he/she must reach for, the trajectory to reach that goal could vary. So, for the purpose of these figures and comparisons with the ground truth, we show here the offline evaluation: we select one demonstrated task trajectory from the test set (not the training set used to learn the ProMP) as ground truth, and imagine that this is the intended trajectory. In **Figure 12**, the ground truth is shown in black, whereas the portion of this trajectory that is fed to the inference, and that corresponds to the "early observations," is represented with bigger black circles. We can see that the inference of the Cartesian position is correct, although we can see an error of about 1 s of the estimated duration time for the last trial. Also, the wrench inference is not accurate. We can assume that it is: because the robot infers the trajectory using only position information without wrench information, or because the wrenches' variation is not correlated to the position variation. To improve this result, we can make the inference using wrench in addition to Cartesian position information, as shown in **Figure 13**. We can see in this figure that the estimation of the trajectory's duration is accurate. The disadvantage is that the inference of the Cartesian position

[16] n_o is not the same for each trajectory test, because it depends on the total duration of the trajectory to be inferred.

[17] Since the model uses the n_o parameter, its computation cannot be performed before this step.

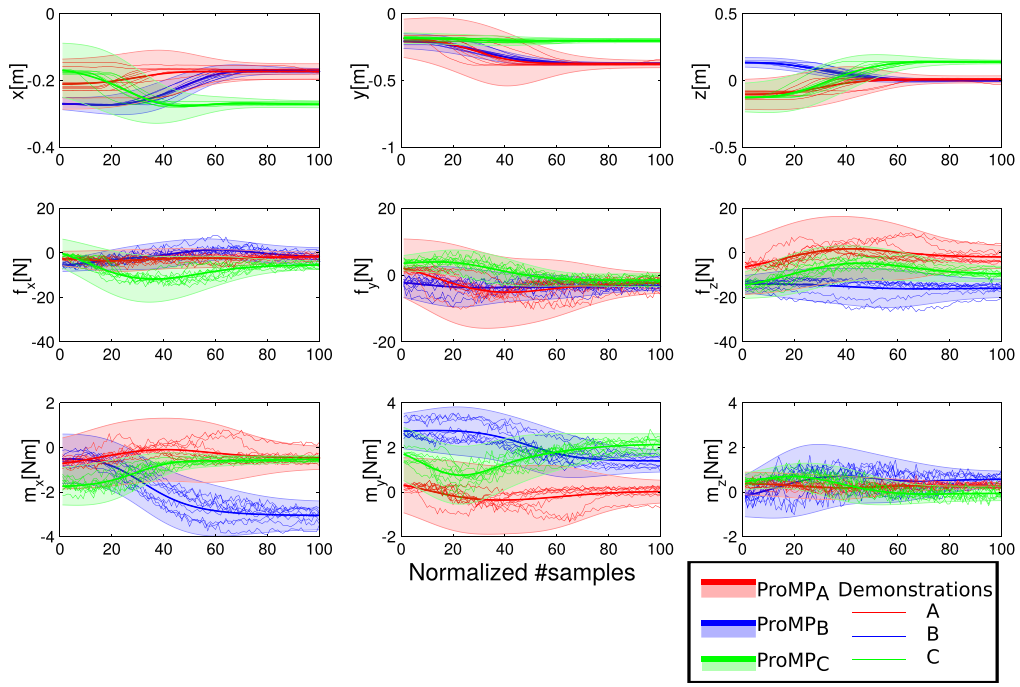


FIGURE 11 | The ProMPs learned by the robot from the demonstrations of **Figure 10**.

is less accurate because the posterior distribution computation makes a trade-off between fitting Cartesian position and wrench early observations. Moreover, to allow a correct inference using wrench information, the noise expectation must be increased to consider forces.[18]

To confirm these results, we analyzed the trajectory inference and α estimation considering different percentages of each trajectory as observed data (30–90%). For each percentage, we performed 20 tests, with and without force information.

In **Figure 14**, each box-plot represents errors for 20 tests. On the top, the error criterion is the average distance between the inferred trajectory and the real one. We can see that the inference of Cartesian end-effector trajectory is more accurate without wrench information. On the bottom, the error criterion is the distance between the estimated α and the real one. We can see that using wrench information, the estimation of the α is more accurate. Thus, these two graphs confirm what we assumed from **Figures 12** and **13**.

Median, mean, and variance of the prediction errors, computed with the normalized root-mean-square error (NRMSE), are reported in Table S1 in Supplementary Material. The prediction error for the time modulation is a scalar: $|\alpha_{\text{prediction}} - \alpha_{\text{real}}|$. The prediction error for the trajectory is computed by the NRMSE of $|\Xi_{\text{prediction}} - \Xi_{\text{real}}|$.

In future upgrades for this application, we will probably use the wrench information only to estimate the time modulation parameter α, to have both the best inference of the intended trajectory and the best estimation of the time modulation parameter to combine the benefits of inference with and without wrench information.

Table S1 in Supplementary Material also reports the average time for computing the prediction of both time modulation and posterior distribution. The computation was performed in Matlab, on a single core laptop (no parallelization). While the computation time for the case "without wrenches" is fine for real-time application, using the wrench information delays the prediction and represents a limit for real-time applications if fast decisions have to taken by the robot. Computation time will be improved in the future works, with the implementation of the prediction in an iterative way.

7.2. Collaborative Object Sorting

We realized another experiment with iCub, where the robot has to sort some objects in different bins (see Figure S2 in Supplementary Material). We have two main primitives: one for a bin located on the left of the robot, and one for the bin to the front. Dropping the object is done at different heights, with a different gesture that also has a different orientation of the hand. For this reason, the ProMP model consists of the Cartesian position of the hand $X_t = [x_t, y_t, z_t] \in R^3$ and its orientation $A_t \in R^4$, expressed as a quaternion:

$$\xi_t = \begin{bmatrix} X_t \\ A_t \end{bmatrix} = \Phi_{\alpha t}\, \boldsymbol{\omega} + \epsilon_t.$$

As in the previous experiment, we first teach the robot the primitives by kinesthetic teaching, with a dozen of demonstrations. Then we start the robot movement: the human operator physically grabs the robot's arm and start the movement toward one of the bins. The robot's skin is used twice. First, to detect the contact

[18] In future versions, we will include the possibility to have different noise models for the observations, e.g., we will have $\Sigma_\Xi^o = \begin{bmatrix} \Sigma_X & 0 \\ 0 & \Sigma_F \end{bmatrix}$. We will therefore set a bigger covariance for the wrench information than for the position information.

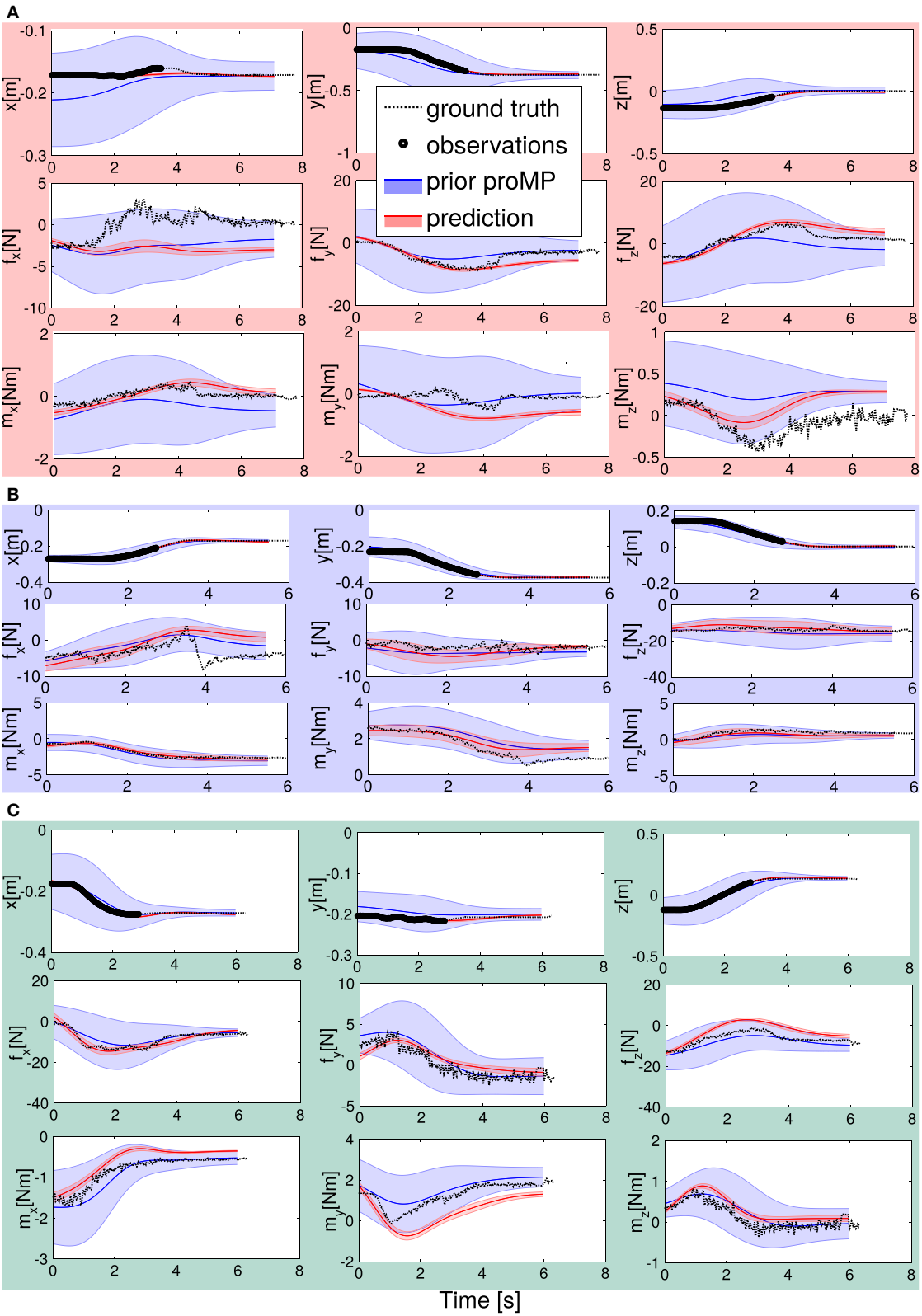


FIGURE 12 | The prediction of the future trajectory from the learned ProMPs computed from the position information for the 3-targets dataset on the real iCub (**Figure 11**) after 40% of observations.

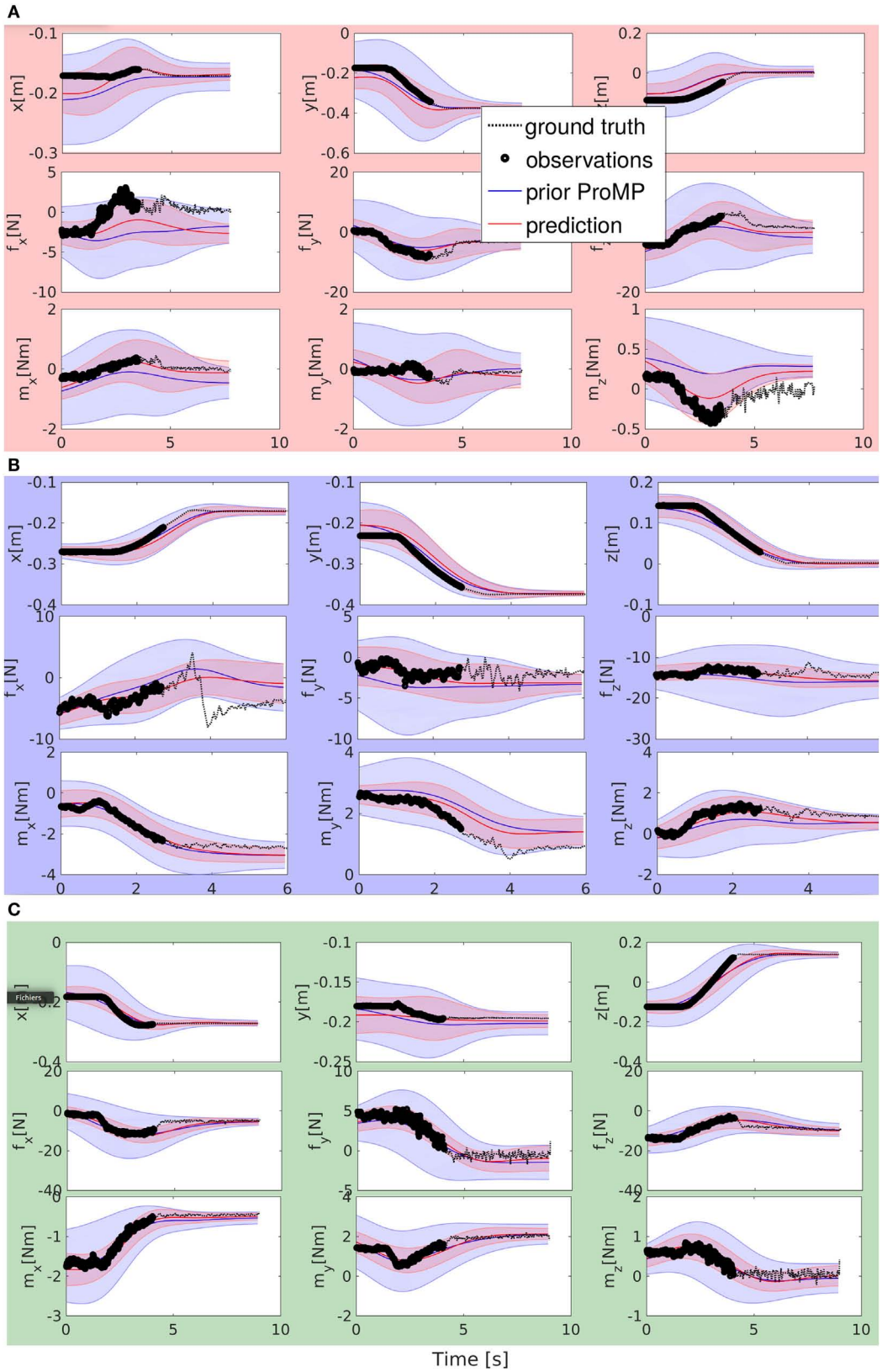


FIGURE 13 | The prediction of the future trajectory from the learned ProMPs computed from the position and wrench information for the 3-targets dataset on the real iCub (**Figure 11**) after 40% of observations.

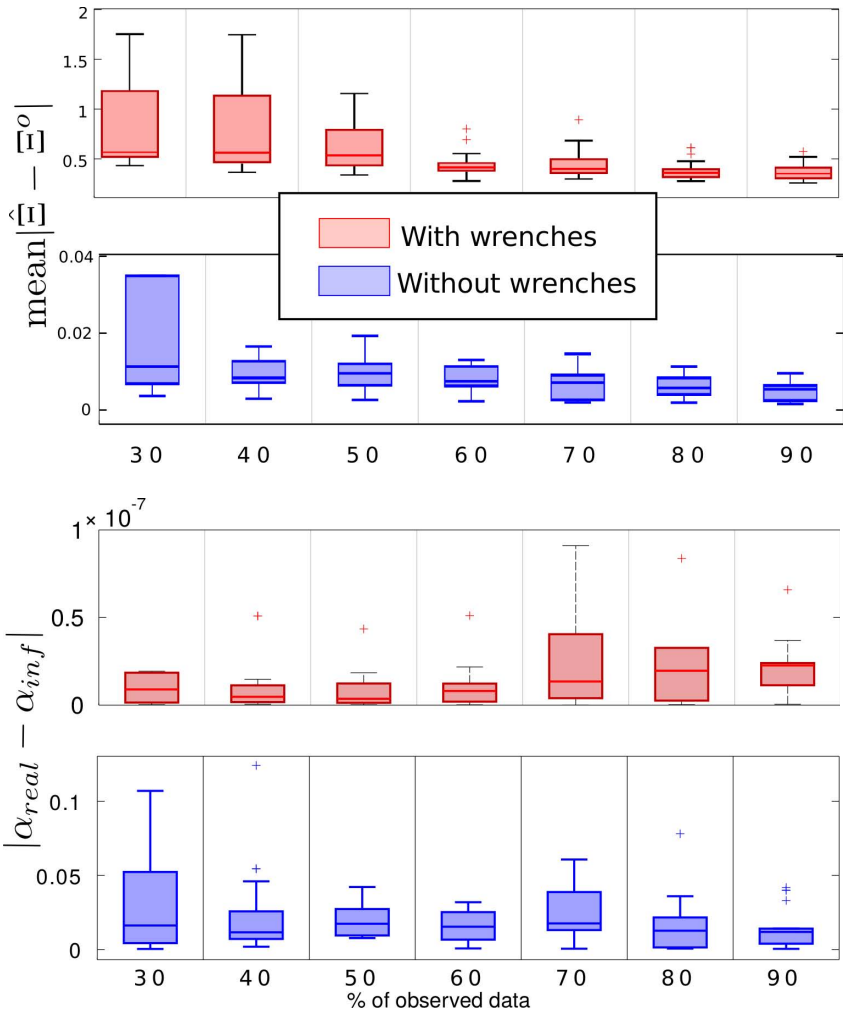


FIGURE 14 | Trajectory prediction error (top) and time modulation estimation error (bottom) of the future trajectory with and without wrench information, for the 3-targets dataset on the real iCub (**Figure 11**) with respect to the number of observed data points.

when the human grabs the arm, which marks the beginning of the observations. Second, when the human breaks the contact with the arm, which marks the end of the observations. Using the first portion of the observed movement, the robot recognizes the current task that is being executed, predicts the future movement that is intended by the human, and then executes it on its own. In the video (see link in Section 8), we artificially introduced a pause to let the operator "validate" the predicted trajectory, using a visual feedback on the iCubGui. Figure S3 in Supplementary Material shows one of the predictions made by the robot after the human releases the arm. Of course in this case, we do not have a "ground truth" for the predicted trajectory, only a validation of the predicted trajectory by the operator.

8. VIDEOS

We recorded several videos that complement the tutorials. The videos are presented in the github repository of our software: https://github.com/inria-larsen/icubLearningTrajectories/tree/master/Videos.

9. DISCUSSION

While we believe that our proposed method is principled and has several advantages for predicting intention in human–robot interaction, there are numerous improvements that can be done. Some will be object of our future works.

9.1. Improving the Estimation of the Time Modulation

Our experiments showed that estimating the time modulation parameter α, determining the duration of the trajectory, greatly improves the prediction of the trajectory in terms of difference with the human intended trajectory (i.e., our ground truth). We proposed four simple methods in Section 3.4, and in the iCub experiment, we showed that the method that maps the time modulation and the variation of the trajectory in the first n_o observations provides a good estimate of the time modulation α for our specific application. However, it is an *ad hoc* model that cannot be generalized to all possible cases. Overall, the estimation of the time modulation (or phase) can be improved. For example,

Maeda et al. (2016) used Dynamic Time Warping, while Ewerton et al. (2015) proposed to improve the estimation by having local estimations of the speed in the execution of the trajectory, to comply with cases where the velocity of task trajectory may not be constant throughout the task execution. In the future, we plan to explore more solutions and integrate them into our software.

9.2. Improving Prediction

Another point that needs further investigation and improvement is how to improve the prediction of the trajectories exploiting different information. In our experiment with iCub, we improved the estimation of the time modulation using position and wrench information; however, we observed that the noisy wrench information does not help in improving the prediction of the position trajectory. One improvement is to certainly exploit more information from the demonstrated trajectories, such as estimating the different noise of every trajectory component and exploiting this information to improve the prediction. Another possible improvement would consist in using contextual information about the task trajectories. Finally, it would be interesting to try to identify automatically the characteristic such as velocity profiles or accelerations, which are renown to play a key role in attributing intentions to human movements. For example, in goal-directed tasks such as reaching, the arm velocity profile, and the hand configuration are cues that helps us detect intentions. Extracting these cues automatically, leveraging the estimation of the time modulation, would probably improve the prediction of the future trajectory. This is a research topic on its own, outside the scope of this article, with strong links to human motor control.

9.3. Continuous Prediction

In Section 3.5, we described how to compute the prediction of the future trajectory after recognizing the current task. However, we did not explore what happens if the task recognition is wrong: this may happen, if there are two or more task with a similar trajectory at the beginning (e.g., moving the object from the same initial point toward one of four possible targets), or simply because there were not enough observed points. So what happens if our task recognition is wrong? How to re-decide on a previously identified task? And how should the robot decide if its current prediction is finally correct (in statistical terms)? While implementing a continuous recognition and prediction is easy with our framework (one has simply to do the estimation at each time step), providing a generic answer to these question may not be straightforward. Re-deciding about the current task implies also changing the prediction of the future trajectory. If the decision does not come with a confidence level greater than a desired value, then the robot could face a stall: if asked to continue the movement but unsure about the future trajectory, should it continue or stop? The choice may be application dependent. We will address these issues and the continuous prediction in future works.

9.4. Improving Computational Time

Finally, we plan to improve the computational time for the inference and the portability of our software by porting the entire framework in C++.

9.5. Learning Tasks with Objects

In many collaborative scenarios, such as object carrying and cooperative assembly, the physical interaction between the human and the robot is mediated by objects. In these cases, if specific manipulations must be done on the objects, our method still applies, but not only on the robot. It must be adapted to the new "augmented system" consisting of robot and object. Typically, we could image a trajectory for some frame or variable or point of interest for the object and learn the corresponding task. Since ProMPs support multiplication and sequencing of primitives, we could exploit the properties of the ProMPs to learn the joint distribution of the robot task trajectories and the object task trajectories.

10. CONCLUSION

In this article, we propose a method for predicting the intention of a user physically interacting with the iCub in a collaborative task. We formalize the intention prediction as predicting the target and "future" intended trajectory from early observations of the task trajectory, modeled by Probabilistic Movement Primitives (ProMPs). We use ProMPs because they capture the variability of the task, in the form of a distribution of trajectories coming from several demonstrations of the task. From the information provided by the ProMP, we are able to compute the future trajectory by conditioning the ProMP to match the early observed data points. Additional features of our method are the estimation of the duration of the intended movement, the recognition of the current task among the many known in advance, and multimodal prediction.

Section 3 described the theoretical framework, whereas Sections 4–7 presented the open-source software that provides the implementation of the proposed method. The software is available on `github`, and tutorials and videos are provided.

We used three examples of increasing complexity to show how to use our method for predicting the intention of the human in collaborative tasks, exploiting the different features. We presented experiments with both the real and the simulated iCub. In our experiments, the robot learns a set of motion primitives corresponding to different tasks, from several demonstrations provided by a user. The resulting ProMPs are the prior information that is later used to make inferences about human intention. When the human starts a new collaborative task, the robot uses the early observations to infer which task the human is executing and predicts the trajectory that the human intends to execute. When the human releases the robot, the predicted trajectory is used by the robot to continue executing the task on its own.

In Section 9, we discussed some current issues and challenges for improving the proposed method and make it applicable to a wider repertoire of collaborative human–robot scenarios. In our future works, our priority would be in accelerating the time for computing the inference and finding a principled way to do continuous estimation, by letting the robot re-decide continuously about the current task and future trajectory.

AUTHOR CONTRIBUTIONS

Designed study: OD, AP, FC, and SI. Wrote software: OD, ME, AP, and SI. Wrote paper: OD, AP, ME, FC, JP, and SI.

ACKNOWLEDGMENTS

The authors wish to thank the IIT researchers of the CoDyCo project for their support with iCub, Ugo Pattacini and Olivier Rochel for their help with the software for the Geomagic, and Iñaki Fernández Pérez for all his relevant feedback.

FUNDING

This study was partially funded by the European projects CoDyCo (no. 600716 ICT211.2.1) and AnDy (no. 731540 H2020-ICT-2016-1) and the French CPER project SCIARAT.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at http://journal.frontiersin.org/article/10.3389/frobt.2017.00045/full#supplementary-material.

REFERENCES

Alami, R., Clodic, A., Montreuil, V., Sisbot, E. A., and Chatila, R. (2006). "Toward human-aware robot task planning," in *AAAI Spring Symposium: To Boldly Go Where No Human-Robot Team Has Gone Before*, Beijing, 39–46.

Amor, H. B., Neumann, G., Kamthe, S., Kroemer, O., and Peters, J. (2014). "Interaction primitives for human-robot cooperation tasks," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on* (Hong Kong: IEEE), 2831–2837.

Baraglia, J., Cakmak, M., Nagai, Y., Rao, R., and Asada, M. (2016). "Initiative in robot assistance during collaborative task execution," in *Human-Robot Interaction (HRI), 2016 11th ACM/IEEE International Conference on* (Christchurch: IEEE), 67–74.

Billard, A., and Mataric, M. J. (2001). Learning human arm movements by imitation: evaluation of a biologically inspired connectionist architecture. *Rob. Auton. Syst.* 37, 145–160. doi:10.1016/S0921-8890(01)00155-5

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Secaucus, NJ: Springer-Verlag New York, Inc.

Busch, B., Grizou, J., Lopes, M., and Stulp, F. (2017). Learning legible motion from human–robot interactions. *Int. J. Soc. Robot.* 1–15. doi:10.1007/s12369-017-0400-4

Buxton, H. (2003). Learning and understanding dynamic scene activity: a review. *Image Vision Comput.* 21, 125–136. doi:10.1016/S0262-8856(02)00127-0

Calinon, S. (2015). *pbdlib-matlab*. Available at: https://gitlab.idiap.ch/rli/pbdlib-matlab/

Calinon, S. (2016). A tutorial on task-parameterized movement learning and retrieval. *Intell. Serv. Robot.* 9, 1–29. doi:10.1007/s11370-015-0187-9

Calinon, S., Bruno, D., and Caldwell, D. G. (2014). "A task-parameterized probabilistic model with minimal intervention control," in *IEEE International Conference on Robotics and Automation (ICRA)*, Hong Kong, 3339–3344.

Calinon, S., Li, Z., Alizadeh, T., Tsagarakis, N. G., and Caldwell, D. G. (2012a). *Statistical Dynamical Systems for Skills Acquisition in Humanoids*. Available at: http://www.calinon.ch/showPubli.php?publi=3031

Calinon, S., Li, Z., Alizadeh, T., Tsagarakis, N. G., and Caldwell, D. G. (2012b). "Statistical dynamical systems for skills acquisition in humanoids," in *Proc. IEEE Intl Conf. on Humanoid Robots (Humanoids)*, Osaka, Japan, 323–329.

Carlson, T., and Demiris, Y. (2008). "Human-wheelchair collaboration through prediction of intention and adaptive assistance," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (Pasadena: IEEE), 3926–3931.

Caron, S., and Kheddar, A. (2016). "Multi-contact walking pattern generation based on model preview control of 3d com accelerations," in *Humanoid Robots, 2016 IEEE-RAS International Conference on*, Cancun.

Csibra, G., and Gergely, G. (2007). 'Obsessed with goals': functions and mechanisms of teleological interpretation of actions in humans. *Acta Psychol.* 124, 60–78. doi:10.1016/j.actpsy.2006.09.007

Demiris, Y. (2007). Prediction of intent in robotics and multi-agent systems. *Cogn. Process.* 8, 151–158. doi:10.1007/s10339-007-0168-9

Dermy, O. (2017). *icubLearningTrajectories*. Available at: https://github.com/inria-larsen/icubLearningTrajectories

DeWolf, T. (2013). *pydmps*. Available at: https://github.com/studywolf/pydmps

Dragan, A., and Srinivasa, S. (2013). "Generating legible motion," in *Proceedings of Robotics: Science and Systems*, Berlin, Germany.

Dragan, A., and Srinivasa, S. (2014). Integrating human observer inferences into robot motion planning. *Auton. Robots* 37, 351–368. doi:10.1007/s10514-014-9408-x

Dumora, J., Geffard, F., Bidard, C., Aspragathos, N. A., and Fraisse, P. (2013). "Robot assistance selection for large object manipulation with a human," in *IEEE International Conference on Systems, Man, and Cybernetics*, Manchester, 1828–1833.

Evrard, P., Gribovskaya, E., Calinon, S., Billard, A., and Kheddar, A. (2009). "Teaching physical collaborative tasks: object-lifting case study with a humanoid," in *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on* (Paris: IEEE), 399–404.

Ewerton, M. (2016). *Learning Motor Skills from Partially Observed Movements Executed at Different Speeds*. Available at: https://github.com/studywolf/pydmps

Ewerton, M., Neumann, G., Lioutikov, R., Ben Amor, H., Peters, J., and Maeda, G. (2015). "Learning multiple collaborative tasks with a mixture of interaction primitives," in *Robotics and Automation (ICRA), 2015 IEEE International Conference on* (Seattle: IEEE), 1535–1542.

Ferrer, G., and Sanfeliu, A. (2014). Bayesian human motion intentionality prediction in urban environments. *Pattern Recognit. Lett.* 44, 134–140. doi:10.1016/j.patrec.2013.08.013

Fine, S., Singer, Y., and Tishby, N. (1998). The hierarchical hidden Markov model: analysis and applications. *Mach. Learn.* 32, 41–62. doi:10.1023/A:1007469218079

Fitts, P. M. (1992). The information capacity of the human motor system in controlling the amplitude of movement. *J. Exp. Psychol. Gen.* 121, 262. doi:10.1037/0096-3445.121.3.262

Fumagalli, M., Ivaldi, S., Randazzo, M., Natale, L., Metta, G., Sandini, G., et al. (2012). Force feedback exploiting tactile and proximal force/torque sensing. *Auton. Robots* 33, 381–398. doi:10.1007/s10514-012-9291-2

Gribovskaya, E., Kheddar, A., and Billard, A. (2011). "Motion learning and adaptive impedance for robot control during physical interaction with humans," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on* (Shanghai: IEEE), 4326–4332.

Hersch, M., Guenter, F., Calinon, S., and Billard. (2008). *Dynamical System Modulation for Robot Adaptive Learning via Kinesthetic Demonstrations*. Available at: http://lasa.epfl.ch/sourcecode/counter.php?ID=11&index=1

Hoffman, G. (2010). "Anticipation in human-robot interaction," in *AAAI Spring Symposium: It's All in the Timing*, Palo Alto, CA.

Huang, S. H., Held, D., Abbeel, P., and Dragan, A. D. (2017). Enabling robots to communicate their objectives. *CoRR*. abs/1702.03465. Available at: http://arxiv.org/abs/1702.03465

Ijspeert, A. J., Nakanishi, J., Hoffmann, H., Pastor, P., and Schaal, S. (2013). Dynamical movement primitives: learning attractor models for motor behaviors. *Neural Comput.* 25, 328–373. doi:10.1162/NECO_a_00393

Ivaldi, S., Anzalone, S., Rousseau, W., Sigaud, O., and Chetouani, M. (2014a). Robot initiative in a team learning task increases the rhythm of interaction but not the perceived engagement. *Front. Neurorobot.* 8:5. doi:10.3389/fnbot.2014.00005

Ivaldi, S., Nguyen, S. M., Lyubova, N., Droniou, A., Padois, V., Filliat, D., et al. (2014b). Object learning through active exploration. *IEEE Trans. Auton. Ment. Dev.* 6, 56–72. doi:10.1109/TAMD.2013.2280614

Ivaldi, S., Fumagalli, M., Nori, F., Baglietto, M., Metta, G., and Sandini, G. (2010). "Approximate optimal control for reaching and trajectory planning in a humanoid robot," in *Proc. of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems – IROS*, Taipei, Taiwan, 1290–1296.

Ivaldi, S., Fumagalli, M., Randazzo, M., Nori, F., Metta, G., and Sandini, G. (2011). "Computing robot internal/external wrenches by means of inertial, tactile and

f/t sensors: theory and implementation on the icub," in *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on* (Bled: IEEE), 521–528.

Ivaldi, S., Lefort, S., Peters, J., Chetouani, M., Provasi, J., and Zibetti, E. (2017). Towards engagement models that consider individual factors in HRI: on the relation of extroversion and negative attitude towards robots to gaze and speech during a human-robot assembly task. *Int. J. Soc. Robot.* 9, 63–86. doi:10.1007/s12369-016-0357-8

Jamone, L., Ugur, E., Cangelosi, A., Fadiga, L., Bernardino, A., Piater, J., et al. (2017). Affordances in psychology, neuroscience and robotics: a survey. *IEEE Trans. Cognit. Dev. Syst.* 99, 1. doi:10.1109/TCDS.2016.2594134

Jarrassé, N., Paik, J., Pasqui, V., and Morel, G. (2008). "How can human motion prediction increase transparency?," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on* (Pasadena, CA: IEEE), 2134–2139.

Keogh, E. (2002). "Exact indexing of dynamic time warping," in *Proceedings of the 28th International Conference on Very Large Data Bases* (Hong Kong: VLDB Endowment), 406–417.

Khansari, M. (2011). *Dynamical Systems Approach to Learn Robot Motions*. Available at: https://bitbucket.org/khansari/seds

Khansari-Zadeh, S. M., and Billard, A. (2011). Learning stable nonlinear dynamical systems with gaussian mixture models. *IEEE Trans. Robot.* 27, 943–957. doi:10.1109/TRO.2011.2159412

Khansari-Zadeh, S. M., and Billard, A. (2012). A dynamical system approach to realtime obstacle avoidance. *Auton. Robots* 32, 433–454. doi:10.1007/s10514-012-9287-y

Kim, J., Banks, C. J., and Shah, J. A. (2017). "Collaborative planning with encoding of users' high-level strategies," in *AAAI Conference on Artificial Intelligence (AAAI-17)*, San Francisco, CA.

Langolf, G. D., Chaffin, D. B., and Foulke, J. A. (1976). An investigation of fitts? Law using a wide range of movement amplitudes. *J. Mot. Behav.* 8, 113–128. doi:10.1080/00222895.1976.10735061

Lober, R. (2014). *Stochastic Machine Learning Toolbox*. Available at: https://github.com/rlober/smlt

Lober, R., Padois, V., and Sigaud, O. (2014). "Multiple task optimization using dynamical movement primitives for whole-body reactive control," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 193–198.

Maeda, G., Ewerton, M., Lioutikov, R., Ben Amor, H., Peters, J., and Neumann, G. (2014). "Learning interaction for collaborative tasks with probabilistic movement primitives," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 527–534.

Maeda, G. J., Neumann, G., Ewerton, M., Lioutikov, R., Kroemer, O., and Peters, J. (2016). "Probabilistic movement primitives for coordination of multiple human–robot collaborative tasks," in *Autonomous Robots*, 1–20.

Meier, F., and Schaal, S. (2016). A probabilistic representation for dynamic movement primitives. *CoRR*. abs/1612.05932. Available at: http://arxiv.org/abs/1612.05932

Micha, H., and Aude, B. (2008). Dynamical system modulation for robot learning via kinesthetic demonstrations. *IEEE Trans. Robot.* 24, 1463–1467. doi:10.1109/TRO.2008.2006703

Nguyen, N. T., Phung, D. Q., Venkatesh, S., and Bui, H. (2005). "Learning and detecting activities from movement trajectories using the hierarchical hidden Markov model," in *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, Vol. 2 (San Diego, CA: IEEE), 955–960.

Palinko, O., Sciutti, A., Patané, L., Rea, F., Nori, F., and Sandini, G. (2014). "Communicative lifting actions in human-humanoid interaction," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on* (Madrid: IEEE), 1116–1121.

Paraschos, A., Daniel, C., Peters, J. R., and Neumann, G. (2013a). "Probabilistic movement primitives," in *Advances in Neural Information Processing Systems*, Stateline, NV, 2616–2624.

Paraschos, A., Neumann, G., and Peters, J. (2013b). "A probabilistic approach to robot trajectory generation," in *Humanoid Robots (Humanoids), 2013 13th IEEE-RAS International Conference on* (Atlanta, GA: IEEE), 477–483.

Paraschos, A., Rueckert, E., Peters, J., and Neumann, G. (2015). "Model-free probabilistic movement primitives for physical interaction," in *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on* (Hamburg: IEEE), 2860–2866.

Pattacini, U., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "An experimental evaluation of a novel minimum-jerk cartesian controller for humanoid robots," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on* (Taipei: IEEE), 1668–1674.

Peters, J., Lee, D. D., Kober, J., Nguyen-Tuong, D., Bagnell, J. A., and Schaal, S. (2016). "Robot learning," in *Springer Handbook of Robotics*, 357–398.

Ren, H., and Xu, G. (2002). "Human action recognition with primitive-based coupled-HMM," in *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, Vol. 2 (Quebec: IEEE), 494–498.

Rozo Castañeda, L., Calinon, S., Caldwell, D., Jimenez Schlegl, P., and Torras, C. (2013). "Learning collaborative impedance-based robot behaviors," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 1422–1428.

Sahin, E., Çakmak, M., Dogar, M. R., Ugur, E., and Üçoluk, G. (2007). To afford or not to afford: a new formalization of affordances toward affordance-based robot control. *Adapt. Behav.* 15, 447–472. doi:10.1177/1059712307084689

Sato, T., Nishida, Y., Ichikawa, J., Hatamura, Y., and Mizoguchi, H. (1994). "Active understanding of human intention by a robot through monitoring of human behavior," in *Proceedings of the IEEE/RSJ/GI International Conference on Intelligent Robots and Systems*, Vol. 1, Munich, 405–414.

Schaal, S. (2006). "Dynamic movement primitives: a framework for motor control in humans and humanoid robotics," in *Adaptive Motion of Animals and Machines* Springer, 261–280.

Sciutti, A., Bisio, A., Nori, F., Metta, G., Fadiga, L., and Sandini, G. (2013). Robots can be perceived as goal-oriented agents. *Interact. Stud.* 14, 329–350. doi:10.1075/is.14.3.02sci

Shah, J., Wiken, J., Williams, B., and Breazeal, C. (2011). "Improved human-robot team performance using chaski, a human-inspired plan execution system," in *Proceedings of the 6th International Conference on Human-Robot Interaction* (Lausanne: ACM), 29–36.

Silva, D. F., and Batista, G. E. (2016). "Speeding up all-pairwise dynamic time warping matrix calculation," in *Proceedings of the 2016 SIAM International Conference on Data Mining* (Miami, FL: SIAM), 837–845.

Soechting, J. (1984). Effect of target size on spatial and temporal characteristics of a pointing movement in man. *Exp. Brain Res.* 54, 121–132. doi:10.1007/BF00235824

Soh, H., and Demiris, Y. (2015). Learning assistance by demonstration: smart mobility with shared control and paired haptic controllers. *J. Hum. Robot Interact.* 4, 76–100. doi:10.5898/JHRI.4.3.Soh

Stulp, F. (2014). *DmpBbo – a c++ Library for Black-Box Optimization of Dynamical Movement Primitives*. Available at: https://github.com/stulp/dmpbbo

Stulp, F., Raiola, G., Hoarau, A., Ivaldi, S., and Sigaud, O. (2013). "Learning compact parameterized skills with a single regression," in *Proc. IEEE-RAS International Conference on Humanoid RObots – HUMANOIDS*, Atlanta, GA, 1–7.

Thill, S., and Ziemke, T. (2017). "The role of intention in human-robot interaction," in *Proceedings of the Companion of the 2017 ACM/IEEE International Conference on Human-Robot Interaction, HRI '17* (New York, NY, USA: ACM), 427–428.

Wang, J. M., Fleet, D. J., and Hertzmann, A. (2005). "Gaussian process dynamical models," in *NIPS*, Vol. 18, Vancouver, 3.

Wang, Z., Deisenroth, M. P., Amor, H. B., Vogt, D., Schölkopf, B., and Peters, J. (2012). "Probabilistic modeling of human movements for intention inference," in *Robotics: Science and Systems* (Sydney, NSW: Citeseer).

Wang, Z., Mülling, K., Deisenroth, M. P., Amor, H. B., Vogt, D., Schölkopf, B., et al. (2013). Probabilistic movement modeling for intention inference in human-robot interaction. *Int. J. Robot. Res.* 32, 841–858. doi:10.1177/0278364913478447

Zube, A., Hofmann, J., and Frese, C. (2016). "Model predictive contact control for human-robot interaction," in *Proceedings of ISR 2016: 47st International Symposium on Robotics*, Munich, 1–7.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

APPENDIX

A. Detail of the Inference Formula

In this Appendix, we explain how to obtain the inference formulae used in our software. First, let us recall the Marginal and Conditional Gaussians laws.[19] Given a marginal Gaussian distribution for x and a Gaussian distribution for y given x in the following form:

$$p(x) = \mathcal{N}\left(x|\mu, \Delta^{-1}\right) p(y|x) = \mathcal{N}\left(Ax + b, L^{-1}\right), \quad \text{(A1)}$$

the marginal distribution of y and the conditional distribution of x given y are given by the following equations:

$$p(y) = \mathcal{N}\left(y|A\mu + b, L^{-1} + A\Delta^{-1}A^{\top}\right), \quad \text{(A2)}$$

$$p(x|y) = \mathcal{N}\left(x|\Sigma A^{\top} L(y - b) + \Delta\mu, \Sigma\right), \quad \text{(A3)}$$

where

$$\Sigma = \left(\Delta + A^{\mathrm{T}} L A\right)^{-1}.$$

We computed the parameter's marginal Gaussian distribution from the set of observed movements:

$$p(\boldsymbol{\omega}) \sim \mathcal{N}(\mu_{\boldsymbol{\omega}}, \Sigma_{\boldsymbol{\omega}}), \quad \text{(A4)}$$

[19] From the book (Bishop, 2006).

From the model $\Xi_t = \Phi_{[1:t_f]}\boldsymbol{\omega} + \epsilon_{\Xi}$, we have the conditional Gaussian distribution for Ξ given $\boldsymbol{\omega}$:

$$p(\Xi|\boldsymbol{\omega}) = \mathcal{N}\Big(\Xi|\Phi_{[1:t_f]}\boldsymbol{\omega}, \Sigma_{\Xi}\Big). \quad \text{(A5)}$$

Then, using equation (A2) we have the following:

$$p(\Xi) = \mathcal{N}\Big(\Xi|\Phi_{[1:t_f]}\mu_{\boldsymbol{\omega}}, \Sigma_{\Xi} + \Phi_{[1:t_f]}\Sigma_{\boldsymbol{\omega}}\Phi_{[1:t_f]}^{\top}\Big). \quad \text{(A6)}$$

that is the prior distribution of the ProMP.

Let $\Xi^o = [\xi^o(1), \ldots, \xi^o(n_o)]$ be the first n_o observations of the trajectory to predict with the first n_o elements corresponding to the early observations.

Let $\hat{\Xi} = [\xi^o(1), \ldots, \xi^o(n_o), \hat{\xi}(n_o + 1), \ldots, \hat{\xi}(t_{\hat{t}_f})]$ be the whole trajectory we have to predict. We can then compute the posterior distribution of the ProMP by using the conditional Gaussians equation (A3):

$$p(\boldsymbol{\omega}|\Xi^o) = \mathcal{N}\big(\boldsymbol{\omega}|\mu_{\boldsymbol{\omega}} + K(\Xi^o - \Phi_{[1:n_o]}\mu_{\boldsymbol{\omega}}), \Sigma_{\boldsymbol{\omega}} - K\Phi_{[1:n_o]}\Sigma_{\boldsymbol{\omega}}\big) \quad \text{(A7)}$$

$$\text{with } K = \Sigma_{\boldsymbol{\omega}}\Phi_{[1:n_o]}^{\top}\Big(\Sigma_{\Xi} + \Phi_{[1:n_o]}\Sigma_{\boldsymbol{\omega}}\Phi_{[1:n_o]}^{\top}\Big)^{-1}. \quad \text{(A8)}$$

Thus, we have the posterior distribution of the ProMP $p(\boldsymbol{\omega}|\Xi^o) = \mathcal{N}(\boldsymbol{\omega}|\hat{\boldsymbol{\mu}}_{\omega}, \hat{\boldsymbol{\Sigma}}_{\omega})$ with:

$$\begin{cases} \hat{\mu}_{\boldsymbol{\omega}} &= \mu_{\boldsymbol{\omega}} + K\big(\Xi^o - \Phi_{[1:n_o]}\mu_{\boldsymbol{\omega}}\big) \\ \hat{\Sigma}_{\boldsymbol{\omega}} &= \Sigma_{\boldsymbol{\omega}} - K\big(\Phi_{[1:n_o]}\Sigma_{\boldsymbol{\omega}}\big) \\ K &= \Sigma_{\boldsymbol{\omega}}\Phi_{[1:n_o]}^{\top}\Big(\Sigma_{\xi}^o + \Phi_{[1:n_o]}\Sigma_{\boldsymbol{\omega}}\Phi_{[1:n_o]}^{\top}\Big)^{-1}. \end{cases} \quad \text{(A9)}$$

frontiers in Robotics and AI

CODE
published: 15 November 2017
doi: 10.3389/frobt.2017.00059

Real-time Pipeline for Object Modeling and Grasping Pose Selection *via* Superquadric Functions

Giulia Vezzani[1,2]* *and Lorenzo Natale*[1]

[1] iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy, [2] University of Genova, Genova, Italy

This work provides a novel real-time pipeline for modeling and grasping of unknown objects with a humanoid robot. Such a problem is of great interest for the robotic community, since conventional approaches fail when the shape, dimension, or pose of the objects are missing. Our approach reconstructs in real-time a model for the object under consideration and represents the robot hand both with proper and mathematically usable models, i.e., *superquadric functions*. The volume graspable by the hand is represented by an ellipsoid and is defined *a priori*, because the shape of the hand is known in advance. The superquadric representing the object is obtained in real-time from partial vision information instead, e.g., one stereo view of the object under consideration, and provides an approximated 3D full model. The optimization problem we formulate for the grasping pose computation is solved online by using the Ipopt software package and, thus, does not require off-line computation or learning. Even though our approach is for a generic humanoid robot, we developed a complete software architecture for executing this approach on the iCub humanoid robot. Together with that, we also provide a tutorial on how to use this framework. We believe that our work, together with the available code, is of a strong utility for the iCub community for three main reasons: object modeling and grasping are relevant problems for the robotic community, our code can be easily applied on every iCub, and the modular structure of our framework easily allows extensions and communications with external code.

Keywords: grasping, object modeling, real-time optimization, C++, superquadric functions

OPEN ACCESS

Edited by:
Maxime Petit,
Imperial College London,
United Kingdom

Reviewed by:
Tobias Fischer,
Imperial College London,
United Kingdom
Uriel Martinez-Hernandez,
University of Leeds,
United Kingdom
Ingo Keller,
Heriot-Watt University,
United Kingdom

***Correspondence:**
Giulia Vezzani
giulia.vezzani@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *28 July 2017*
Accepted: *30 October 2017*
Published: *15 November 2017*

Citation:
Vezzani G and Natale L (2017) Real-time Pipeline for Object Modeling and Grasping Pose Selection via Superquadric Functions. Front. Robot. AI 4:59. doi: 10.3389/frobt.2017.00059

1. INTRODUCTION

Industrial robotics shows how high performance in manipulation can be achieved if a very accurate knowledge of the environment and the objects is provided. On the contrary, grasping of unknown objects or whose pose is uncertain is still an open problem. In this work, we present a novel framework for modeling and grasping unknown objects with the iCub humanoid robot.

The iCub humanoid robot is provided with two 7DOF arms, 5 fingers human-like hands, whose fingertips are covered by tactile sensors and two cameras, as described in Metta et al. (2010). Therefore, it turns out to be a suitable platform for investigating objects perception and grasping problem: the stereo vision system and the tactile sensors can be exploited together to get proper information for modeling and grasping unknown objects. The method and the code, we propose in this work, consist of reconstructing an object model through the stereo vision system of the robot and using this information to compute a suitable grasping pose. Once the robot reaches the desired grasping pose on the object surface, the tactile response of the fingertips is used to achieve a stable grasp for lifting the object.

The iCub community put a great effort into the development of a sharable and reusable code. With this work, we want to contribute in this direction, detailing the code we designed for implementing our grasping approach for a possible user interested in executing our technique on the robot.

2. MODELING AND GRASPING *VIA* SUPERQUADRIC MODELS

The superquadric modeling and grasping framework we make use of is based on the idea that low-dimensional, compact, mathematical representation of objects can provide computational and theoretical advantages in hard problems tackled in robotics, such as trajectory planning for exploration, grasping and approaching toward objects. This takes inspiration from theories conceived during the 90s and 2000s (Jaklic et al., 2013) where superquadric functions were proposed as a mathematical and low-dimensional model for representing objects.

In Vezzani et al. (2017), we proposed a novel approach that solves the grasping problem by modeling the object and the volume graspable by the hand with superquadric functions. The latter is represented by an ellipsoid and is defined *a priori*, because the shape of the hand is known in advance. The superquadric representing the object is obtained in real-time from partial vision information instead, e.g., one stereo view of the object under consideration, and provides an approximated 3D full model. Both the modeling and the grasping problem are cast into an optimization framework and solved in real-time with the software package Ipopt (Wächter and Biegler, 2006).

In this article, we do not go into the mathematical details (extensively reported in Vezzani et al. (2017)) whereas we focus on the description of the code designed for using the approach on the iCub, since we believe it to be useful for any user interested in object modeling and grasping tasks. A brief mathematical description of the methodologies is reported in the *README.md* files of the Github repositories.[1]

3. CODE STRUCTURE

We designed two modules, namely, *superquadric-model* and *superquadric-grasp*, which implement, respectively, the modeling and the grasping approached described in Vezzani et al. (2017).

Our leading idea is to develop a *self-contained* code that provides *query services* to the user. In this respect, our code handles only the information strictly necessary for the superquadric modeling and grasping approach and minimizes the dependencies from external modules. The user is asked to write a wrapper code that communicates with the two modules and makes them properly interact. In this respect, we provide a tutorial code,[2] implementing a possible use case of our modules, that can be adapted by the user to fit in his own pipeline (see Section 3.3).

In the next paragraphs, we first describe the implementation of the *superquadric-model* and *superquadric-grasp* modules, which is based on the *Yarp* middleware (Metta et al., 2006). Then, we outline a possible use case implementing a complete pipeline for object modeling and grasping.

3.1. Superquadric-Model

The *superquadric-model* module computes the superquadric function best representing the object of interest given a partial 3D point cloud of the object.

The module, whose structure is outlined in **Figure 1**, consists of the *SuperqModule* class, derived from the YARP *RFModule* class. The SuperqModule launches following two separate YARP *Rate Threads*:

- the *SuperqComputation* class, which manages the superquadric computation;
- the *SuperqVisualization* class, which can be enabled to show the estimated superquadric or the object 3D points overlapped on the camera image.

The SuperqModule also provides some *Thrift IDL services*[3] suitable for getting information on the internal state of the module and setting the thread parameters on the fly. *Thrift* is a software framework for scalable cross-language development, which allows to build services working efficiently with different programming languages.

While there are two threads to decouple the functionalities of computation and visualization, the threads share some variables (in particular the computed superquadric) to increase their speed.

3.1.1. SuperqComputation

The *SuperqComputation* thread includes the following steps:

- Once the object point cloud is provided (see Section 3.3 for a detailed description of how extract the object point cloud), the superquadric is estimated by using Ipopt (Wächter and Biegler, 2006), a C++ software package for large-scale nonlinear optimization. The user can formulate its own optimization problem with the Ipopt C++ interface[4] and, then, solve it through the Ipopt solver.
- A median filter with an adaptive window of width m can be enabled to stabilize the estimated superquadric over the time. Even if the object is not supposed to move during a grasping task, it may happen that the user, or anyone interacting with the robot, moves the object in a different location. In this case, the superquadric modeler should be able to track the object and the estimated superquadric should not be affected by previous estimations in different poses. For this reason, the window width of the median filter changes according to the object velocity. If the object location changes (i.e., its velocity increases), the window width becomes smaller. On the contrary, if the object is not moved, the window width can be increased. In this way, when the object pose is constant, its superquadric estimation is more stable and accurate, while it is not affected by past estimations if the object pose changes. The median filter and the object velocity estimation are achieved by using, respectively, the *iCub MedianFilter Class* and the *iCub AWLinEstimator Class*.

[1] https://github.com/robotology/superquadric-model,https://github.com/robotology/superquadric-grasp.

[2] https://github.com/robotology/superquadric-grasp-example.

[3] https://thrift.apache.org/docs/idl.

[4] https://www.coin-or.org/Ipopt/documentation/node23.html.

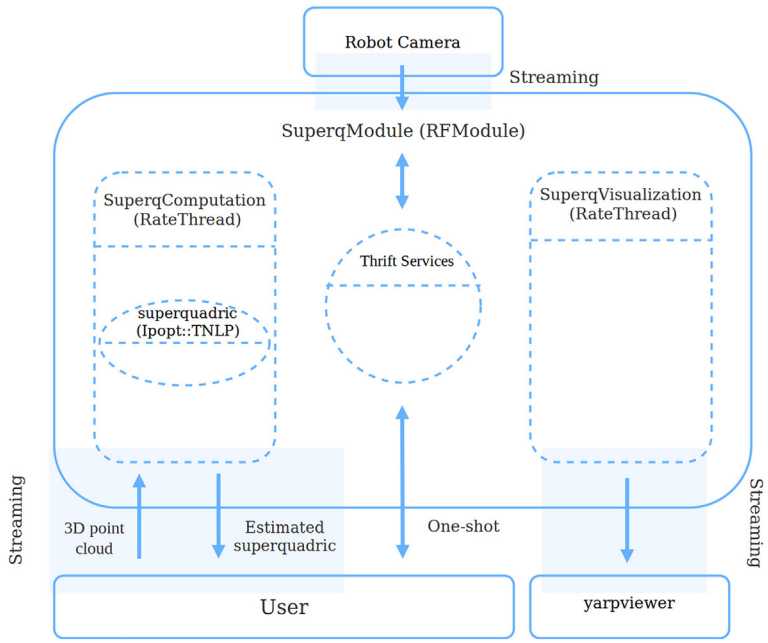


FIGURE 1 | Superquadric-model code structure. The class SuperqModule, derived from the YARP RFModule class, launches two threads, respectively for superquadric computation and visualization. The class provides some thrift services to the user for interacting with the module. More detail on the *user box* is provided in Section 3.3 and in **Figure 2**.

- If prior information is available on the object shape (e.g., given by a classifier or a vision recognition system), the module can use it to speed up the superquadric estimation. Particularly, if the object is labeled as *cylinder, box* or *sphere*, specific constraints can be used for improving the accuracy and reducing the execution time of the optimization problem.

The user can communicate with the *SuperqComputation* thread, through the *SuperqModule*, in the two different modes:

- In **streaming mode**—the 3D point cloud of the object should be sent to the module through a YARP *Buffered port* as a YARP *Property*. The user can access the current estimated superquadric through a dedicated YARP *Buffered port* as a YARP *Property*, where the main components of the superquadric are grouped as: *dimensions*, *exponents*, *center*, and *orientation*.
- In **one-shot mode**—the user can ask the module to compute the object superquadric by sending a single point cloud through a YARP *RpcClient Port* and getting a YARP *Property* including the estimated superquadric parameters as reply. In case the user asks for the superquadric filtered by the median filter, he should send a set of point clouds of the object in the same pose.

The superquadric computation, together with the superquadric filtering process, takes 0.1 s in average on Intel®Core™ i7-4710MQ Processor @2.50 GHz. This values is compatible with our real-time requirements.

3.1.2. SuperqVisualization

The visualization thread overlaps the estimated superquadric or the 3D points used by the optimizer on the camera image, for real-time visual inspection by the user (see **Figure 3** (4)). The average visualization time is equal to 0.01 s and can be enable or disabled by the user while the *SuperqModule* is running.

3.2. Superquadric-Grasping

The *superquadric-grasp* module implements the approach proposed in Vezzani et al. (2017) for the computation of grasping poses by using a superquadric modeling the object.

The *superquadric-grasp* module consists of the *GraspModule class*, derived from the YARP *RFModule class*. The *GraspModule* splits pose computation and visualization and grasp execution in three different classes:

- *GraspComputation class*, computing the pose for grasping the object;
- *GraspVisualization class*, showing the object model and the main information about the computed poses;
- *GraspExecution class*, which allows executing the grasping task once the pose is computed and one of the robot hand is selected.

As for the *superquadric-model* module, the *superquadric-grasp* implementation provides several *Thrfit IDL services* to the user to interact with the module and for getting information on the state of the module. The *superquadric-grasp* module structure is similar to the *superquadric-model* one, shown in **Figure 1**.

3.2.1. GraspComputation

This class handles the pose candidates' computation:

- Given the superquadric modeling the object, received as a YARP *Property* (see 3.1.1), the grasping poses for one or both

the hands (according to the user query) are computed together with a suitable trajectory by using the method proposed in Vezzani et al. (2017). The optimization problem is formulated and solved through the Ipopt C++ interface.
- The user can exploit some prior information for adapting the grasp computation to the desired scenario. In particular, the user can provide the module the height of the support on which the object is located (i.e., a table) to prevent the robot hand from hitting it. In addition, the constraints about the final hand pose can be modified according to the experimental scenario. For instance, the user can define the robot workspace by simply varying the variable upper and lower bounds of the optimization problem from the configuration files.

The pose computation process takes 2.0 s in average, which is consistent with the time requirements of a grasp task execution.

3.2.2. GraspExecution

The *GraspExecution class* controls the arm movements to accomplish the grasping task. In particular:

- The approaching step, i.e., the pose reaching through the trajectory waypoints, is executed through the YARP *Cartesian Interface* (Pattacini et al., 2010);
- Once the final pose is reached, the grasp is executed by using a precision grasp method described in Regoli et al. (2016) and available in the *Tactile Control library*.[5] The hand fingers close until the tactile sensors on the fingertips detect contact. Then, each finger is controlled to find a stable grasp for the object. Alternatively, the grasp can be performed by simply closing the fingers until a minimum pressure of the fingertips is measured. However, such an approach does not guarantee stability while lifting the object.

3.2.3. GraspVisualization

The visualization thread overlaps the computed poses and the received object superquadric on the camera image, for real-time visual inspection by the user (see **Figure 3** (5)). Some additional information, such as the volume graspable by the hand and the trajectory waypoints can be shown at the same time.

3.2.4. Communication with the Module

Unlike the *superquadric-model* framework, the user can communicate with the *GraspModule* only in **one-shot mode**. In particular, the user can query the module to:

- Compute the grasping poses and approaching trajectory, providing to the module the estimated superquadric of the object as a *Yarp Property* (as described in 3.1.1) and selecting one or both the hands. The solutions are given back to the user as a *Yarp Property*.
- Ask the robot to reach the final pose and grasp the object by selecting one robot hand. In the current code implementation, the robot performs a simple lifting test to check the stability of the grasp.

The additional *thrift services* allows setting on the fly parameters for grasp computation, visualization, and execution.

3.3. How to Use the Superquadric Framework

To use our grasping approach, the user is supposed to design a wrapper code to combine together the outcomes of the *superquadric-model* and *superquadric-grasp* modules. In addition, the implementation of a complete modeling and grasping pipeline requires the use of external modules for point cloud computation. We provide a tutorial code, which takes advantage of modules developed by the iCub community to achieve the modeling and grasping task. Hereafter, we report the main steps of the complete pipeline. The entire commented code is available on Github,[6] together with a detail description on how to run the code in the *README.md* file.

1. The object is labeled with a name through a recognition system.[7] The object label, together with information on its 2D bounding box, are stored by the *Object Property Collector*[8] (Moulin-Frier et al., 2017). The wrapper code is given the object name by the user (through a *RpcPort*) and uses it for asking the object property collector for the relative 2D bounding box.
2. The 2D blob of the object is computed by the *lbpExtract module*, once it is provided with the bounding box information. This uses Local Binary Pattern (LBP) (Ojala et al., 1996) to analyze the texture of what is in the robot view (a table in our experimental scenario). This texture is used for getting a general blob information both as an image, containing general white blobs of where the objects are, and as a *Yarp Bottle* containing lists of bounding box points. Then, the general blob information allow using grabCut algorithm (Rother et al., 2004) to properly segment all the objects on the table.
3. Given the 2D blob, the wrapper code reconstructs the 3D point cloud by querying the *Structure from Motion module* (Fanello et al., 2014). This module uses a complete Structure From Motion (SFM) pipeline for the computation of the extrinsics parameters between two different views. These parameters are then used to rectify the images and to compute a depth map.
4. Then, the wrapper code asks the *superquadric-model* to estimate the superquadric modeling the object by sending the acquired point cloud to the module.

```
Bottle cmd, superq_bottle;
//Fill the Bottle for querying
superquadric-model.
```

[5] https://github.com/robotology/tactile-control.

[6] https://github.com/robotology/superquadric-grasp-example.

[7] https://github.com/robotology/iol/tree/master/src/himrepClassifier.

[8] https://github.com/robotology/icub-main/tree/master/src/modules/objectsPropertiesCollector.

```
cmd.addString("get_superq");
Bottle &bottle_point = cmd.addList();
for (size_t i = 0; i < points.size(); i++)
{
 Bottle &in = bottle_point.addList();
 in.addDouble(points[i][0]);
 in.addDouble(points[i][1]);
 in.addDouble(points[i][2]);
}
superqRpc.write(cmd, superq_bottle);
//Then, extract the estimated superqua-
dric from the Bottle superq_bottle.
```

5. Once the superquadric is estimated, the user code asks the *superquadric-grasp* module to compute pose candidates for grasping the object.

```
Bottle cmd, reply;
//Fill the Bottle for querying
superquadric-grasp.
cmd.addString("get_grasping_pose");
//hand_for_computation can be "right",
"left" or "both"
cmd.addString(hand_for_computation);
graspRpc.write(cmd, reply);
//Then, extract the grasping pose
candidate from the Bottle reply.
```

6. Finally, the user can ask the *superquadric-grasp* to perform the grasping task.

```
Bottle cmd, reply;
//Fill the Bottle for moving the arm.
cmd.addString("move");
cmd.addString(hand_for_moving);
graspRpc.write(cmd, reply);
//The grasp is executed.
```

Figure 2 outlines the structure of the entire pipeline, following the steps described in this section. In **Figure 3**, we show some typical outcomes of all the steps described above. In addition, in the *README.md* files of the superquadric-model and superquadric-grasp repository, we provide two videos of the execution of the modeling and the grasping pipeline.[9]

[9]superquadric-model demo: https://www.youtube.com/watch?v=MViX4Ppo4WQ&feature=youtube. superquadric-grasp demo: https://www.youtube.com/watch?v=eGZO8peAVao.

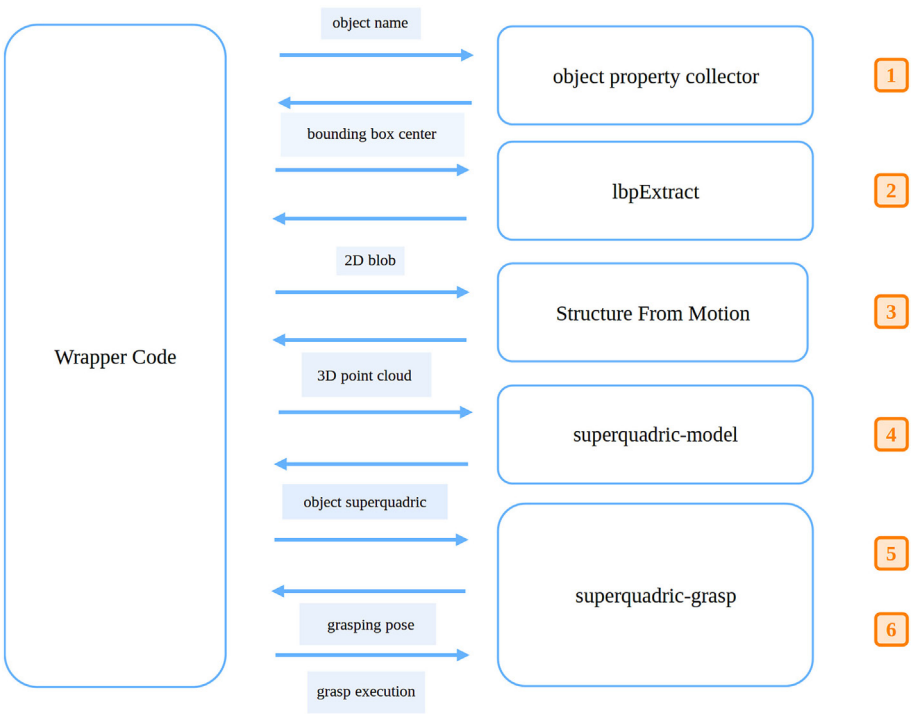


FIGURE 2 | Modules communication for the implementation of the modeling and grasping pipeline. The wrapper code manages the interaction between external modules and the *superquadric-model* and *superquadric-grasp* frameworks. Pipeline steps enumerated as in Section 3: (1) The wrapper code asks the object property collector for the bounding box information of the object. (2) Given that, lbpExtract module provides the 2D blob of the object. (3) The wrapper code sends the 2D blob of the object to the Structure From Motion module for getting the relative 3D point cloud. (4) The 3D point cloud is then sent to the *superquadric-model* for computing the superquadric modeling the object. (5) The wrapper code sends the estimated superquadric to the *superquadric-grasp* module, which computes suitable poses. (6) Finally, the *superquadric-grasp* is asked to perform the grasping task.

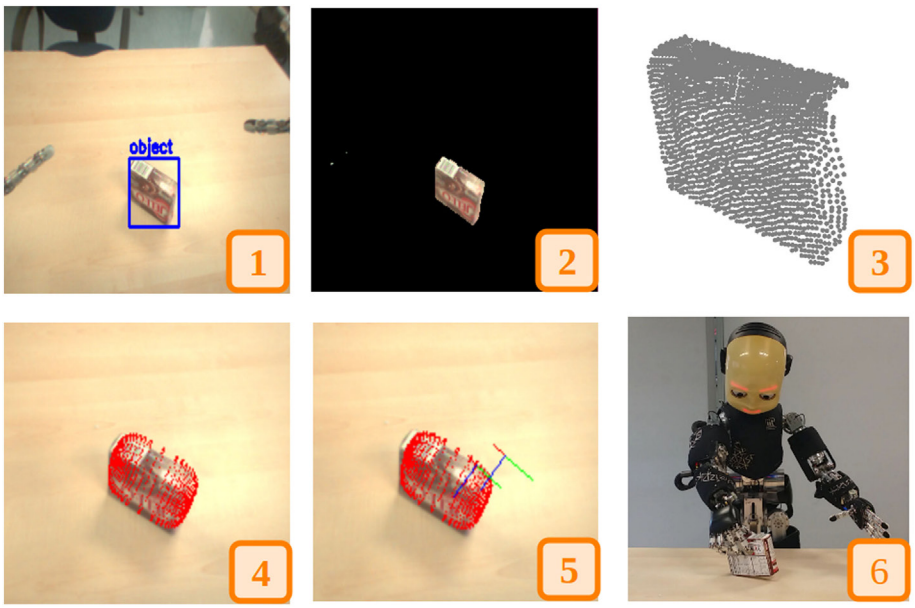


FIGURE 3 | Outcomes of the modeling and grasping pipeline. (1) The object is stored by the object property collector with the label *object*. (2) LbpExtract provides the 2D blob of the object. (3) The 3D point cloud is extracted from the disparity map, by querying the Structure From Motion module. (4) The superquadric modeling the object is reconstructed. (5) The grasping pose and approaching trajectory for the right hand are computed. (6) The robot grasps the object. (Steps (1), (2), (4), and (5) are represented by screenshots from the visualizers.).

4. KNOWN ISSUES

In this section, we report the limitations of our approach, together with possible solutions for facing them.

- Our approach is currently an open-loop approach. Once the object model and the grasping pose are computed, the robot reaches for the final pose without checking if the object pose changes. However, we could monitor the object pose, by estimating only the pose of the reconstructed superquadric - leaving its shape unchanged - with new point clouds while the robot is moving and until the object is in the robot field of view. This is a viable solution since our modeling approach is compatible with real-time requirements (as shown in Section 3.1).
- A further limitation caused by the open-loop nature of our approach is the missing compensation of errors between the robot stereo vision and system. To properly run the grasping pipeline, the user is required to properly calibrate the vision and the robot kinematics. In case errors between the two are still a problem for grasping the object, empirical offsets can be added for compensating for the errors. More information are provided in the *README.md* of the superquadric-grasp repository.
- A quite strong limitation of our approach is that it cannot automatically distinguish between good and wrong poses. For this reason, the user need to supervise the entire process and ask for a new model and pose in case the current outcome is not suitable for grasping the objects. In particular, this problem arises when the object cannot be represented with a single superquadric for its geometric shape. As future work, we aim at extend our approach for modeling more complex objects with multiple superquadrics.

5. CONCLUSION

In this work, we detail the implementation of the modeling and grasping approach pipeline described in Vezzani et al. (2017). We developed two modules, namely *superquadric-model* and *superquadric-grasp*, that respectively model objects through superquadric functions and computes suitable grasping poses for the iCub robot. Our leading idea was to develop a self-contained code that provides query services to the user. Our software handles only the information strictly necessary for the modeling and grasping approach and minimizes the dependencies from external modules. The user is supposed to design a wrapper code to combine together the outcomes of the two modules. We provide also an example of a external code in the *superquadric-grasp-example* repository for the implementation of a complete modeling and grasping pipeline.

In the next future, we would like to improve the approach we use for reaching the final grasping pose, which is a current limitation of our approach, as described in Section 4. The iCub proprioception is in fact affected by a number of impairments, mainly caused by elastic elements, which introduce errors in the computation of direct kinematics. Also, the iCub is provided with moving cameras for simulating the human oculomotor system. This makes the knowledge of extrinsic parameters and, thus, the object information estimation quite noisy. These sources of error might be crucial for grasping tasks, when a final pose is required to be reached with errors in order of 1 cm. We can solve this problem

by using the approach described in Fantacci et al. (2017), which provides a precise estimate of the robot end-effector pose over time and a visual servoing approach without the use of markers. Another extension of the modeling pipeline consists in using the recognition system[10] described in Pasquale et al. (2016) to classify the objects of interest according to their geometric property for using some prior information on their shape for improving and speeding up the superquadric estimation process, as mentioned in 3.1.1.

[10]https://github.com/robotology/onthefly-recognition.

AUTHOR CONTRIBUTIONS

GV developed the method and the code and described them in the manuscript. LN supervised the code and method development and the manuscript writing.

REFERENCES

Fanello, S. R., Pattacini, U., Gori, I., Tikhanoff, V., Randazzo, M., Roncone, A., et al. (2014). "3D stereo estimation and fully automated learning of eye-hand coordination in humanoid robots," in *2014 14th IEEE-RAS International Conference on Humanoid Robots (Humanoids)* (Madrid, Spain: IEEE), 1028–1035.

Fantacci, C., Pattacini, U., Tikhanoff, V., and Natale, L. (2017). "Visual end-effector tracking using a 3D model-aided particle filter for humanoid robot platforms," in *IEEE Conference on Intelligent Robots and Systems (IROS)* (Vancouver, Canada: IEEE).

Jaklic, A., Leonardis, A., and Solina, F. (2013). *Segmentation and Recovery of Superquadrics*, Vol. 20. Springer Science & Business Media.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 8. doi:10.5772/5761

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi:10.1016/j.neunet.2010.08.010

Moulin-Frier, C., Fischer, T., Petit, M., Pointeau, G., Puigbo, J., Pattacini, U., et al. (2017). Dac-h3: a proactive robot cognitive architecture to acquire and express knowledge about the world and the self. *IEEE Trans. Cogn. Dev. Syst.* doi:10.1109/TCDS.2017.2754143

Ojala, T., Pietikäinen, M., and Harwood, D. (1996). A comparative study of texture measures with classification based on featured distributions. *Pattern Recognit.* 29, 51–59. doi:10.1016/0031-3203(95)00067-4

Pasquale, G., Ciliberto, C., Rosasco, L., and Natale, L. (2016). "Object identification from few examples by improving the invariance of a deep convolutional neural network," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Deajeon, South Korea: IEEE), 4904–4911.

Pattacini, U., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "An experimental evaluation of a novel minimum-jerk Cartesian controller for humanoid robots," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei, Taiwan: IEEE), 1668–1674.

Regoli, M., Pattacini, U., Metta, G., and Natale, L. (2016). "Hierarchical grasp controller using tactile feedback," in *IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun, Mexico: IEEE), 387–394.

Rother, C., Kolmogorov, V., and Blake, A. (2004). Grabcut: interactive foreground extraction using iterated graph cuts. *ACM Trans. Graph.* 23, 309–314. doi:10.1145/1015706.1015720

Vezzani, G., Pattacini, U., and Natale, L. (2017). "A grasping approach based on superquadric models," in *IEEE International Conference on Robotics and Automation (ICRA)* (Singapore), 1579–1586.

Wächter, A., and Biegler, L. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.* 106, 25–57. doi:10.1007/s10107-004-0559-y

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer, TF, and handling editor declared their shared affiliation.

frontiers in Robotics and AI

CODE
published: 18 December 2017
doi: 10.3389/frobt.2017.00067

Connecting YARP to the Web with Yarp.js

*Carlo Ciliberto**

University College London, London, United Kingdom

We present yarp.js, a JavaScript framework enabling robotics networks to interface and interact with external devices by exploiting modern Web communication protocols. By connecting a YARP server module with a browser client on any external device, yarp.js allows to access on board sensors using standard Web APIs and stream the acquired data through the yarp.js network without the need for any installation. Communication between YARP modules and yarp.js clients is bi-directional, opening also the possibility for robotics applications to exploit the capabilities of modern browsers to process external data, such as speech synthesis, 3D data visualization, or video streaming to name a few. Yarp.js requires only a browser installed on the client device, allowing for fast and easy deployment of novel applications. The code and sample applications to get started with the proposed framework are available for the community at the yarp.js GitHub repository.

Keywords: yarp, robotics, iCub, web, websocket, Internet of things

OPEN ACCESS

Edited by:
Ugo Pattacini,
Fondazione Istituto Italiano di Technologia, Italy

Reviewed by:
Tobias Fischer,
Imperial College London, United Kingdom
Alessandro Roncone,
Yale University, United States
Lars Schillingmann,
Bielefeld University, Germany

***Correspondence:**
Carlo Ciliberto
c.ciliberto@ucl.ac.uk

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *19 August 2017*
Accepted: *22 November 2017*
Published: *18 December 2017*

Citation:
Ciliberto C (2017) Connecting YARP to the Web with Yarp.js. Front. Robot. AI 4:67. doi: 10.3389/frobt.2017.00067

1. INTRODUCTION

Smartphones, tablets, and wearable devices have drastically changed human communication and are nowadays a key component of everyday life, enabling humans to connect with each other and other devices in real time, forming a dense network of complex and frequent interactions. In this revolution, the Internet and Web technologies in general are playing the key role of a "lingua franca," establishing novel standards for modern communication protocols adopted by most platforms and operating systems. Indeed, as information technologies advance, we are steadily moving toward an "Internet of Things (IoT)" (Xia et al., 2012), where everyday object will be able to offer an interface for digital communication with humans and other devices.

In this scenario, robotic agents designed to operate in human environments will undoubtedly need to be well-versed in these new practices to seamlessly integrate within the IoT network. Towards this goal, in this paper we present *yarp.js*, a novel framework developed with the goal of connecting the YARP network with external devices using modern Internet protocols. YARP (Metta et al., 2006) is to date one of the most efficient and flexible robotics middlewares, adopted by many robotics laboratories worldwide and used as main communication tool for robotic platforms, such as the humanoid iCub (Metta et al., 2008) and R1 (Parmiggiani et al., 2017). In this sense, yarp.js provides a platform-independent approach to establish a two-way communication between YARP modules (e.g., the robot itself or other machines on the YARP network) and external systems whose only requirement is the ability to run an Internet browser.

Yarp.js decouples a server side, which must run on the YARP network, from a client side, which simply needs to be capable of tcp/ip communication with the server. The server side is built over a Node.js (Tilkov and Vinoski, 2010) abstraction layer wrapping the main YARP functionalities (e.g., opening/connecting ports, creating bottles or images, and writing/reading them via ports). Two

main benefits arise from this choice of server-side language: 1) the possibility to write YARP modules in Node.js and therefore, leverage the wide range of packages made available by the related community via the well-established Node Package Manager (NPM),[1] and 2) the event-based philosophy of Node.js offers a different perspective for programming the robot cognitive skills, possibly allowing for novel and more reactive behaviors. Yarp.js server is supported on OSX 10.11.6+ and Ubuntu 16.04+.

The client side of yarp.js consists of a pure JavaScript library and runs on both Google Chrome[2] and Firefox[3] browsers. Communication is performed across WebSockets, which allow for real-time exchange of data between the device on which the client is running and the server. Yarp.js endows both client and server with same functionalities, allowing also clients on external device to open and write/read on a YARP port. This is particularly useful to connect an external sensor, such as a smartphone microphone, inertial sensors, camera, etc., to the YARP network and allowing other modules to access its measurements. In this sense, yarp.js allows to effortlessly extend YARP functionalities to non-YARP devices by simply serving the required JavaScript library so that there is no need for custom installation, essentially making yarp.js automatically platform-independent on any browser-enabled device.

Yarp.js v1.0.0[4] is available for the community as a GitHub repository.[5] We have provided a number of examples for new users to get started with the proposed framework.

2. BACKGROUND AND MOTIVATIONS

We introduce the necessary background and motivations to understand the main contributions of yarp.js.

2.1. YARP

Yet Another Robot Platform (YARP) (Metta et al., 2006) is a framework developed to handle the low-level communication processes between different sensors, processors, and actuators in robotics applications. The main goal of YARP is to provide researchers and developers with a unifying cross-platform layer of communication in order to foster the diffusion and reproducibility of novel results in robotics. **Figure 1** (left half) reports a pictorial representation of a YARP network, where a number of computational nodes (gray circles) communicate with each other by leveraging on the abstraction layer offered by YARP (blue lines). In a spirit similar to YARP, several robotics frameworks have been proposed in the recent literature, such as Player (Gerkey et al., 2003), ROS (Quigley et al., 2009), OROCOS (Bruyninckx, 2001), MIRO (Utz et al., 2002), and LCM (Huang et al., 2010) to name a few. We refer to Fitzpatrick et al. (2014) for a discussion on the topic.

Unarguably, the most successful example of YARP application is the iCub (Metta et al., 2008), a humanoid robot adopted by more than 30 laboratories worldwide: Exploiting the flexibility of YARP functionalities, computational models developed by a number of different research groups to perform diverse tasks ranging from torque control (Fumagalli et al., 2010, 2012; Del Prete et al., 2012) to grasping (Gori et al., 2014), balancing (Pucci et al., 2016), visual attention (Ruesch et al., 2008), visual or haptic object recognition (Ciliberto et al., 2013; Higy et al., 2016), supervised learning (Gijsberts and Metta, 2011), can be combined on the same platform, enabling the robot with advanced cognitive capabilities such as in Ivaldi et al. (2013); Fischer and Demiris (2016); Morse and Cangelosi (2017).

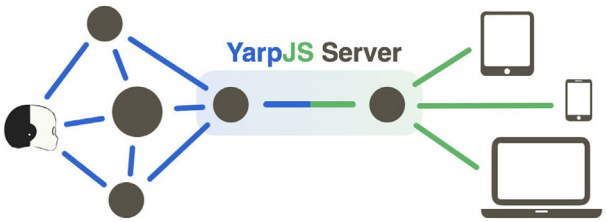


FIGURE 1 | Example of YARP network connected with non-YARP-capable devices on the Web via yarp.js.

2.2. Robots, Modern Web APIs, and Node.js

With the diffusion of lightweight portable devices, such as smartphones and tables, in recent years it has become a necessity for web applications to efficiently access and process information acquired from diverse sensors, such as microphones, embedded cameras, or inertial sensors. To this end, most modern browser has designed a wide range of APIs that allow accessing such resources across most devices, platforms, and operating systems. This has significantly fostered the deployment and diffusion of many novel applications capable of running natively in the browser, such as image object recognition,[6] GPS mapping and route planning,[7] speech-based assistants,[8] videoconferencing,[9] navigation in virtual reality environments[10] to name a few.

Making these capabilities available to a robot is clearly appealing and indeed the potential benefits of such interaction have been thoroughly investigated in the literature (Taylor and Wright, 1995; Hu et al., 2012; Kamei et al., 2012; Kehoe et al., 2015). However, robotics application typically requires real-time performance and deploying the necessary communication infrastructure to satisfy such requirements can be difficult or not possible due to compatibility issues. On the contrary, Web APIs are already designed to take care of the low-level communication with embedded sensors as well as the transmission of data across

[1] https://www.npmjs.com.
[2] https://www.google.com/chrome.
[3] https://www.mozilla.org.
[4] Yarp.js DOI: https://doi.org/10.5281/zenodo.1007786.
[5] https://github.com/robotology/yarp.js.
[6] https://www.clarifai.com/.
[7] maps.google.com.
[8] https://sdkcarlos.github.io/sites/artyom.html.
[9] https://appr.tc/.
[10] https://playcanv.as/p/sAsiDvtC/.

a network (i.e., the Internet). In this sense, the yarp.js framework proposed in this work acts as an intermediate layer allowing YARP and a browser to communicate, essentially "assimilating" non-YARP capable devices within the robot's network.

The above motivations are shared with recent work (Osentoski et al., 2011; Toris et al., 2015), where a JavaScript framework was developed to allow portable devices to communicate with the Robot Operating System (ROS) using Websockets and JavaScript. In this sense, the client side of yarp.js can be interpreted as the equivalent of the ros.js framework for the YARP environment, and one interesting byproduct of this work is the possibility to create applications that naturally bridge YARP and ROS frameworks by leveraging the two corresponding JavaScript libraries.

A second relevant byproduct of our work is the extension of standard YARP C++ routines to Node.js. This could be beneficial in developing robotics applications. Indeed, Node.js (and more generally JavaScript) is based on a system of callbacks that are activated when the corresponding registered event occurs (Tilkov and Vinoski, 2010). While this approach can be equivalently implemented in more traditional languages used in robotics (indeed its core is based on a C++ engine), Node.js encourages a programming style that is asynchronous by design and in this sense could be helpful in speeding-up the development of high-level applications in robotics without the need for ad-hoc careful synchronization between multiple modules and threads. As a practical example, consider the ActionsRenderingEngine (ARE)[11]: this iCub module manages a number of possible behaviors for the robot, combining both visual cues and motor actions and requires several threads (e.g., a vision thread, a motor thread, a visuo-motor thread, etc.) to be carefully synchronized in order to avoid low-level errors (e.g., concurrent memory access). This module would be significantly easier to develop (and read/debug), if written in an event-based language where the low-level details related to asynchrony are taken care of by design.

In the rest of this paper, we describe yarp.js and present a number of sample applications highlighting the potential benefits of the proposed framework in robotics.

3. SYSTEM OVERVIEW

Yarp.js is conceptually organized in two separate components: a server side, equipped with YARP communication capabilities and a client side, which is able to transmit and receive data from other nodes on the YARP network by exploiting the server side as a proxy. **Figure 1** reports a pictorial representation of a yarp.js network, where messages from non-YARP equipped devices (e.g., smartphones, tablets, etc.) are first sent via WebSockets (green lines) to the yarp.js server and then propagated through the YARP network (blue lines). The communication with YARP and WebSockets is bi-directional, allowing to transmit data from the network to the client.

The two-level structure of yarp.js is imposed by the nature of web technologies. Indeed, while on one hand browsers offer flexible cross-platform solutions to the deployment of novel applications, they also need to cope with extremely critical security issues (e.g., handling of passwords or sensitive data over the Internet). As a consequence, code running in the browser is allowed very limited interaction with the rest of the machine hosting it, let alone other machines on the same local network. In this sense, the server side of yarp.js can be interpreted as a standard YARP module that is also able to communicate with the browser, effectively acting as the missing link between the client and the YARP network.

As a final note, we care to point out that YARP is already equipped with basic HTTP communication functionalities[12] via Representation State Transfer (REST) (Fielding, 2000). However, RESTful interoperability is not suited for real-time two-way communication between server and client; one of the main motivations that led to the design of the WebSocket standard (Lubbers and Greco, 2010).

3.1. Server Side: YARP in Node.js

The server side of yarp.js is written in Node.js (Tilkov and Vinoski, 2010) and comprises two layers: first, a low-level library of C++ addons for Node.js [13] that allows to access and use YARP objects and functionalities from the Node.js environment. Second, a set of Node.js APIs offering easier management of the YARP addons (e.g., opening and connections of ports) as well as communication with client browsers. Below, we discuss these two layers in detail.

3.1.1. First Layer: Node.js Addons for YARP (Language C++ → Node.js)

This layer exposes the APIs to create the following YARP objects as Node.js objects: Bottle, Image, Sound, BufferedPort, RPCPort, and Network. It is written in C++ using the *Native Abstraction for Node.js* (NAN)[14] library and provides a set of Node.js wrappers for the corresponding YARP objects. As an example, below we report the minimal Node.js code to open a YARP port and write a Bottle on it using yarp.js.

```
var yarp = require('<yarp.js-folder>/build/Release/
Yarp JS');
                                          //get yarp.js
var yarp_net = new yarp.Network();
                                //get the YARP network


var port = new yarp.BufferedPortBottle();
                                       //create a port
port.open('/yarpjs/example');
                         //open it on the YARP network


var bottle = port.prepare();
                         //prepare the Bottle to write
bottle.fromString('hello yarp.js!');
                                      //fill the Bottle
port.write();
                           //write it over the network
```

[11] http://wiki.icub.org/brain/group__actionsRenderingEngine.html.

[12] http://www.yarp.it/yarp_http.html.

[13] https://nodejs.org/api/addons.html.

[14] https://github.com/nodejs/nan.

Note that these addons can be used as a standalone package to develop YARP modules in Node.js. This is extremely advantageous that it allows to effortlessly import Node.js packages from NPM to YARP applications. As a matter of fact, the second layer of yarp.js leverages a number of NPM packages to manage the communication between YARP and the browsers.

Callbacks. Callbacks can be provided dynamically to YARP objects. Below, we report the minimal code for reading from a port and printing the content of the received message on the terminal.

```
port.onRead(function(yarp_object){
   console.log('Message received: '+yarp_object.
   toString());
});
```

Extending yarp.js. By leveraging on the NAN abstraction layer, it is possible to easily extend yarp.js addons with new functionalities or create new ones wrapping other YARP objects. However, one aspect of this process deserves particular care, namely the conceptual separation between the threaded nature of YARP applications and the event-based philosophy of Node.js. To this end, we provide the `C++` class `YarpJS_Callback`, which stems a separate Node.js worker thread from the main one and runs the prescribed callback function when the required event occurs. This allows to dynamically provide callback functions to YARP objects as discussed above.

3.1.2. Second Layer: Yarp.js Server Manager (Language Node.js)

The second layer is a JavaScript module wrapping the yarp.js addons provided and offering (opinionated) management functionalities: 1) a *Port Manager* handling operations on the YARP network, such as opening/closing/connection of ports and 2) a *Browser Communicator* in charge of the communication with the client via WebSocket. In particular, this latter component interprets messages from the browser as either messages to be propagated to the network or as YARP commands that cannot be executed directly from the browser (e.g., opening a port).

Port Manager. This component exposes a set of functions meant to simplify the management of the YARP network from the Node.js module. Specifically, it allows to recover ports by name, connect two ports, and offer fallbacks in case of name conflicts (e.g., more clients trying to open the same port). It also manages to close all hanging objects when the Node.js module ends, cleaning memory and the YARP network. The code snippet below shows the difference in using the manager rather than the rawNode.js addons.

```
var yarp = require('<yarp.js-folder>/yarp.js');
                                          //get yarp.js
//no need to call YARP network

var port = new yarp.Port('bottle');
                                    //create a port
port.open('/yarpjs/example');
                     //open it on the YARP network

var bottle = port.prepare();
                     //prepare the Bottle to write
bottle.fromString('hello yarp.js!');
                                    //fill the Bottle
port.write();
                          //write it over the network
//alternatively, port.write('hello yarp.js!'); would do
the same
```

Browser Communicator. The browser communication component is based on the Socket.io package, which is designed to create webservers with robust WebSockets functionalities. To initialize the yarp.js manager it is sufficient to provide a Socket.io object to the `Browsercommunicator` method. All the communication with client browsers is then automatically handled. The following code makes use of the standard HTTP[15] and Express[16] packages to provide a minimal example on how to create a webserver offering yarp.js functionalities and listening on a port for incoming connections.

```
var http = require('http').Server(require
('express')());
                              //create the web server
var io = require('socket.io')(http);
                       //create the Socket.io object
http.listen(3000);
                //Run the server on locahlhost:3000

var yarp = require('<path to yarp.js>');
                         //get the yarp.js addons layer
yarp.browserCommunicator(io);
                    //Initialize the yarp.js manager
```

Once the yarp.js manager is initialized with Socket.io, all messages coming from the client side of yarp.js are automatically captured and processed by it. In Section 3.2, we list the main functionalities offered by using this intermediate layer.

This component is in charge of communicating to the Port Manager in which YARP ports are to be opened instead of the browser clients. In particular, whenever such a port reads a message in input, the Browser Communicator recovers it and pushes to the corresponding clients via WebSockets. This piping of the message is meant to create the "illusion" of having the browsers directly reading from the port. This is extremely helpful to develop code for the client side of yarp.js, however, it is important to keep in mind that for computationally intensive applications the Browser Communicator could become a bottleneck through which all messages from YARP to the clients need to flow. Clearly, this issue could be mitigated by having more than one yarp.js server module running on the network.

3.2. Client Side: YARP in the Browser (Language JavaScript)

The client side of yarp.js is a lightweight JavaScript library that leverages the browser implementation of Socket.io to communicate with the server side described in Section 3.1. The only requirement in this sense is for the browser to have WebSocket functionalities. Yarp.js can be initialized using the following code,

[15]https://nodejs.org/api/all.html.
[16]https://expressjs.com.

which here is assumed to be placed in the HTML page served to the browser:

```
<script src = "/socket.io/socket.io.js"></script>
<script src = "/yarp.js"></script>
<script>
   yarp.init(io());
   yarp.onInit(function(){
     //yarp.jscode
   });
</script>
```

The yarp.js manager on the client side offers the same APIs of the Port Manager on the server side. Specifically, it exposes a `Network` object that can be used to create new connections among ports on the YARP network and also a `Port` object that can be used to create new buffered ports and open them. As explained before, these operations cannot be performed directly by the client but are rather executed on the server side of yarp.js after receiving the corresponding message via WebSocket. Below, we report a code sample showing how to open a port and write/read messages which are automatically sent to the YARP network. All JavaScript code is to be assumed to be run within the onInit.

```
let port = new Port(port_type);
                    //port_type default: 'bottle'
port.open(port_name);
      //if the port does not exist the server open
                                          ones.

port.write([1,2,3]);
            //write a bottle containing 3 integers

port.onRead(function(yarp_object){
   console.log(yarp_object.toString());
});
```

The functionalities of yarp.js in the browser allow to easily develop and deploy YARP applications on the hosting device as we describe in the following.

4. APPLICATIONS

On the yarp.js repository we provide a number of sample applications to get new users started with the proposed framework. They are organized in a single bundle[17] that can be run by executing the code

```
$>node examples/examples.js
```

on the machine, where the server side of yarp.js is installed. Then, from any other device on the same local network, the example bundle can be accessed by navigating with Firefox on Google Chrome browser on http://<ip.of.yarpjs.machineer:3000.

Figure 2 shows how examples are rendered to the user.

4.1. Reading and Transmitting Inertial Data

This application shows how sensors on external devices (e.g., where YARP is not installed) can be accessed from the YARP network. We make use of the Web API[18] to read from the inertial sensor of a smartphone and stream it through a port.

```
window.addEventListener("deviceorientation",
function(event){
   port_orientation_out.write([event.alpha,event.
   beta,event.gamma]);
}, true);
```

Another client can read the inertial data streamed through the network and visualize the corresponding 3D orientation of the device using WebGL functionalities (a topic addressed in more detail in Section 4.4). **Figure 2B** shows a snapshot of this application.

4.2. Speech Recognition and Synthesis

This application uses the Web Speech API[19] for speech recognition and synthesis. To simplify the access to the Web Speech API yarp.js provide a synthetizer

```
yarpSpeakPort.onRead(function(msg){
   yarp.Synthetizer.speak(msg);
});
```

which allows to speak aloud text, read from a YARP port and Recognizer module

```
yarp.Recognizer.enableAutorestart();
    \\starts the speech recognition module
yarp.Recognizer.addEventListener('yarp speech finished',
function (e) {
   yarpSpeechRecPort.write(e.detail[0].transcript);
}, false);
```

which recognizes human speech from the embedded microphone and emits the event "`yarp speech finished`" as soon as the Web Speech API consider the audio signal to have terminated.

4.3. Stream Video (a "yarpview" in the Browser)

YARP images can be read from a port on the yarp.js client and visualized in the browser. Ideally, the WebRTC protocol (Johnston and Burnett, 2012) should be adopted for the transmission of large amounts of data over UDP. Unfortunately, to this date a standard solution for server-to-browser WebRTC communication does not exist. To reduce the burden on the server/client communication, we compress the images in either PNG or JPEG before sending them over WebSockets.

Images can be then visualized in a `<canvas>` HTML element using the following code.

```
let canvas = document.getElementByTag('canvas');
let img = new Image();
```

[17] https://github.com/robotology/yarp.js/tree/master/examples.

[18] https://developer.mozilla.org/en-US/docs/Web/API/Window/ondeviceorientation.

[19] https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html.

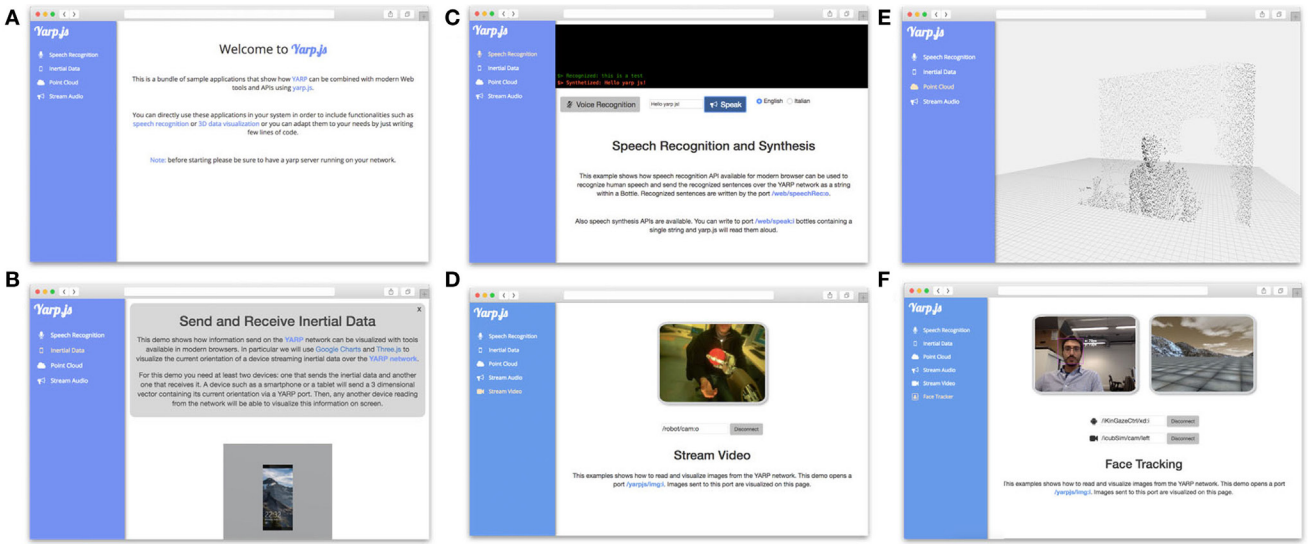


FIGURE 2 | The bundle of yarp.js application examples available on the project repository. **(A)** Landing page of the examples bundle. **(B)** Reading and transmitting inertial data (Section 4.1). **(C)** Speech Recognition and Synthesis (Section 4.2). **(D)** Visualizing Yarp Images in the Browser (Section 4.3). **(E)** 3D Visualization of YARP data (Section 4.4). **(F)** Face tracking for robot teleoperation 4.5. All depicted individuals provided their consent for the publication of this image.

```
port_video_in.onRead(function(yarp_img){
    img.src = yarp.getImageSrc(yarp_img.compres-
    sion_type,yarp_img.buffer);
    canvas.getContext('2d').drawImage(img,0,0);
});
```

Figure 2D shows the yarp.js acting as a `yarpviewer`[20] in the browser.

4.4. 3D Visualization of YARP Data

WebGL[21] is a standard Web API providing 3D graphics functionalities on the browser. Exploiting the three.js-WebGL library[22], we built a simple application to visualize point clouds read from YARP ports received as Bottles of one or more 3D array which are interpreted as 3D cordinates and rendered in a navigable virtual scene (**Figure 2E**).

Note that allowing the browser to directly interact with the graphic card of the hosting machine opens a wide range of possibilities. Indeed, recently there has been interest in developing applications to run Deep Learning models directly in the browser.[23]

4.5. Teleoperation with Face Tracking

We conclude by proposing a teleoperation application, where a face tracker running in the browser is used to actively control the head of the iCub robot. We used the Tracker.js[24] library to capture images from the device camera, detect the face of a user, and obtain the (u,v) position of the corresponding rectangle in the image. Then, the position was translated to a 3D point

$$(x, y, z) = (-1, u/w - 0.5, v/h - 0.3)$$

which is sent to the `/xd:i` port of the `iKinGazeCtrl`[25] (Roncone et al., 2016) to control the gaze of the robot to point toward it. See the following code.

```
let tracker = new tracking.Object Tracker
('face');
tracker.on('track', function(event){
    let rect = event.data[0];
    let u = rect.x + rect.width/2;
    let v = rect.y + rect.height/2;

gazePort.write([-1, (u/w - 0.5), (v/h - 0.3)]);
});
```

where w and h, respectively denote the height and width of the device camera. **Figure 2F** shows an example of this application, where images streamed from the robot camera are send back to the browser are described in Section 4.3.

5. CONCLUSION

We have presented yarp.js, a JavaScript framework to enable YARP-based robotics systems with modern Web APIs functionalities. Yarp.js allows modules running on the YARP network to access sensors information on devices that are not equipped with the YARP communication layer by exploiting WebSocket communication. By leveraging on Web technologies, applications

[20] http://www.yarp.it/yarpview.html.
[21] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API.
[22] https://threejs.org.
[23] http://cs.stanford.edu/people/karpathy/convnetjs, https://github.com/transcranial/keras-js, https://pair-code.github.io/deeplearnjs/, https://tenso.rs.
[24] https://trackingjs.com.
[25] http://wiki.icub.org/brain/group__iKinGazeCtrl.html.

based on yarp.js are easy to deploy and develop. We have presented a number of applications showing the benefit of the proposed approach.

Yarp.js is easy to extend and a main challenge in the future will be to enrich its capabilities with WebRTC functionalities, which would be the natural solution to the issues related to the transmission of large amounts of data between server and client. We will investigate this direction in future work.

AUTHOR CONTRIBUTIONS

The author confirms being the sole contributor of this work and approved it for publication.

FUNDING

This work was supported by EPSRC grant EP/P009069/1.

REFERENCES

Bruyninckx, H. (2001). "Open robot control software: the Orocos project," in *Proceedings 2001 IEEE International Conference on Robotics and Automation, ICRA 2001*, Vol. 3 (Seoul: IEEE), 2523–2528.

Ciliberto, C., Fanello, S. R., Santoro, M., Natale, L., Metta, G., and Rosasco, L. (2013). "On the impact of learning hierarchical representations for visual recognition in robotics," in *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Tokyo: IEEE), 3759–3764.

Del Prete, A., Nori, F., Metta, G., and Natale, L. (2012). "Control of contact forces: the role of tactile feedback for contact localization," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (San Francisco: IEEE), 4048–4053.

Fielding, R. (2000). "Representational state transfer," in *Architectural Styles and the Design of Netowork-Based Software Architecture*, 76–85.

Fischer, T., and Demiris, Y. (2016). "Markerless perspective taking for humanoid robots in unconstrained environments," in *2016 IEEE International Conference on Robotics and Automation (ICRA)* (Stockholm: IEEE), 3309–3316.

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Software Eng. Robot.* 5, 42–49.

Fumagalli, M., Ivaldi, S., Randazzo, M., Natale, L., Metta, G., Sandini, G., et al. (2012). Force feedback exploiting tactile and proximal force/torque sensing. *Auton. Robots* 33, 381–398. doi:10.1007/s10514-012-9291-2

Fumagalli, M., Randazzo, M., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "Exploiting proximal f/t measurements for the icub active compliance," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 1870–1876.

Gerkey, B., Vaughan, R. T., and Howard, A. (2003). "The player/stage project: tools for multi-robot and distributed sensor systems," in *Proceedings of the 11th International Conference on Advanced Robotics*, Coimbra, Vol. 1, 317–323.

Gijsberts, A., and Metta, G. (2011). "Incremental learning of robot dynamics using random features," in *2011 IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai: IEEE), 951–956.

Gori, I., Pattacini, U., Tikhanoff, V., and Metta, G. (2014). "Three-finger precision grasp on incomplete 3d point clouds," in *2014 IEEE International Conference on Robotics and Automation (ICRA)* (Hong Kong: IEEE), 5366–5373.

Higy, B., Ciliberto, C., Rosasco, L., and Natale, L. (2016). "Combining sensory modalities and exploratory procedures to improve haptic object recognition in robotics," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun: IEEE), 117–124.

Hu, G., Tay, W. P., and Wen, Y. (2012). Cloud robotics: architecture, challenges and applications. *IEEE Netw.* 26. doi:10.1109/MNET.2012.6201212

Huang, A. S., Olson, E., and Moore, D. C. (2010). "LCM: lightweight communications and marshalling," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Taipei: IEEE), 4057–4062.

Ivaldi, S., Lyubova, N., Droniou, A., Gerardeaux-Viret, D., Filliat, D., Padois, V., et al. (2013). "Learning to recognize objects through curiosity-driven manipulation with the icub humanoid robot," in *2013 IEEE Third Joint International Conference on Development and Learning and Epigenetic Robotics (ICDL)* (Osaka: IEEE), 1–8.

Johnston, A. B., and Burnett, D. C. (2012). *WebRTC: APIs and RTCWEB Protocols of the HTML5 Real-Time Web*. Digital Codex LLC.

Kamei, K., Nishio, S., Hagita, N., and Sato, M. (2012). Cloud networked robotics. *IEEE Netw.* 26. doi:10.1109/MNET.2012.6201213

Kehoe, B., Patil, S., Abbeel, P., and Goldberg, K. (2015). A survey of research on cloud robotics and automation. *IEEE Trans. Autom. Sci. Eng.* 12, 398–409. doi:10.1109/TASE.2014.2376492

Lubbers, P., and Greco, F. (2010). HTML5 web sockets: a quantum leap in scalability for the web. *SOA World Mag.* 1.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 8. doi:10.5772/5761

Metta, G., Sandini, G., Vernon, D., Natale, L., and Nori, F. (2008). "The ICUB humanoid robot: an open platform for research in embodied cognition," in *Proceedings of the 8th Workshop on Performance Metrics for Intelligent Systems* (Gaithersburg: ACM), 50–56.

Morse, A. F., and Cangelosi, A. (2017). Why are there developmental stages in language learning? a developmental robotics model of language development. *Cogn. Sci.* 41, 32–51. doi:10.1111/cogs.12390

Osentoski, S., Jay, G., Crick, C., Pitzer, B., DuHadway, C., and Jenkins, O. C. (2011). "Robots as web services: reproducible experimentation and application development using ROSJS," in *2011 IEEE International Conference on Robotics and Automation (ICRA)* (Shanghai: IEEE), 6078–6083.

Parmiggiani, A., Fiorio, L., Scalzo, A., Sureshbabu, A. V., Randazzo, M., Maggiali, M., et al. (2017). "The design and validation of the r1 personal humanoid," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Vancouver: IEEE), 2591–2598.

Pucci, D., Romano, F., Traversaro, S., and Nori, F. (2016). "Highly dynamic balancing via force control," in *2016 IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun: IEEE), 141–141.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Vol. 3 (Kobe), 5.

Roncone, A., Pattacini, U., Metta, G., and Natale, L. (2016). "A cartesian 6-dof gaze controller for humanoid robots," in *Proceedings of Robotics: Science and Systems*, AnnArbor, MI. doi:10.15607/RSS.2016.XII.022

Ruesch, J., Lopes, M., Bernardino, A., Hornstein, J., Santos-Victor, J., and Pfeifer, R. (2008). "Multimodal saliency-based bottom-up attention a framework for the humanoid robot icub," in *2008 IEEE International Conference on Robotics and Automation, ICRA 2008* (Pasadena: IEEE), 962–967.

Taylor, A. L., and Wright, J. T. (1995). "A telerobot on the world wide web," in *In National Conference of the Australian Robot Association* (Melbourne: Citeseer).

Tilkov, S., and Vinoski, S. (2010). Node. js: using javascript to build high-performance network programs. *IEEE Internet Comput.* 14, 80–83. doi:10.1109/MIC.2010.145

Toris, R., Kammerl, J., Lu, D. V., Lee, J., Jenkins, O. C., Osentoski, S., et al. (2015). "Robot web tools: efficient messaging for cloud robotics," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Chicago: IEEE), 4530–4537.

Utz, H., Sablatnog, S., Enderle, S., and Kraetzschmar, G. (2002). Miro-middleware for mobile robot applications. *IEEE Trans. Robot. Autom.* 18, 493–497. doi:10.1109/TRA.2002.802930

Xia, F., Yang, L. T., Wang, L., and Vinel, A. (2012). Internet of things. *Int. J. Commun. Syst.* 25, 1101. doi:10.1002/dac.2417

Conflict of Interest Statement: The author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

CODE
published: 16 January 2018
doi: 10.3389/frobt.2017.00073

The Event-Driven Software Library for YARP—With Algorithms and iCub Applications

Arren Glover, Valentina Vasco, Massimiliano Iacono and Chiara Bartolozzi**

iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy

Event-driven (ED) cameras are an emerging technology that sample the visual signal based on changes in the signal magnitude, rather than at a fixed-rate over time. The change in paradigm results in a camera with a lower latency, that uses less power, has reduced bandwidth, and higher dynamic range. Such cameras offer many potential advantages for on-line, autonomous, robots; however, the sensor data do not directly integrate with current "image-based" frameworks and software libraries. The iCub robot uses Yet Another Robot Platform (YARP) as middleware to provide modular processing and connectivity to sensors and actuators. This paper introduces a library that incorporates an event-based framework into the YARP architecture, allowing event cameras to be used with the iCub (and other YARP-based) robots. We describe the philosophy and methods for structuring *events* to facilitate processing, while maintaining low-latency and real-time operation. We also describe several processing modules made available open-source, and three example demonstrations that can be run on the neuromorphic iCub.

Keywords: iCub, neuromorphic engineering, event-driven vision, software, humanoid robotics

OPEN ACCESS

Edited by:
Lorenzo Jamone, Queen Mary University of London, United Kingdom

Reviewed by:
Garrick Orchard, National University of Singapore, Singapore
Hanme Kim, Imperial College London, United Kingdom

****Correspondence:***
Arren Glover
arren.glover@iit.it;
Chiara Bartolozzi
chiara.bartolozzi@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *26 July 2017*
Accepted: *12 December 2017*
Published: *16 January 2018*

Citation:
Glover A, Vasco V, Iacono M and Bartolozzi C (2018) The Event-Driven Software Library for YARP—With Algorithms and iCub Applications. Front. Robot. AI 4:73. doi: 10.3389/frobt.2017.00073

1. INTRODUCTION

Conventional vision sensors used in robotics rely on the acquisition of sequences of static images at fixed temporal intervals. Such a sensor provides the most information when the temporal dynamics of the scene match the sample-rate. If the dynamics are slower (e.g., a mostly static scene), only a small percentage of pixels change between two consecutive frames, leading to redundant acquisition and processing. Alternatively, if the scene dynamics are much faster (e.g., a falling object), information between images can be distorted by motion blur, or missed entirely.

A newly emerging technology, "event-driven" (ED) cameras, are vision sensors that produce digital "events" only when the amount of light falling on a pixel changes. The result is that the cameras detect only contrast changes (Lichtsteiner et al., 2008) that occur due to the relative motion between the environment and the sensor. There is no fixed sampling rate over time, instead, the sensor adapts to the scene dynamics. Redundant data are simply not produced in slow dynamic scenes, and the sensor output still manages to finely trace the movement of any fast stimuli. Specifically, the camera hardware latency is only 15 μs (Lichtsteiner et al., 2008) and the temporal resolution at which an event can be timestamped is under 1 μs. Events are also produced asynchronously for each pixel, such that processing operations can start without the need to read the entire sensor array, and a low-latency processing pipeline can be realized.

ED cameras provide many potential advantages for robotics applications. The removal of redundant processing can give mobile robots longer operating times and frees computational resources for other tasks. Fast-moving stimuli can always be detected, and visual dynamics estimated with more accuracy than with conventionally available cameras. This low-latency can enable extremely fast

reaction times between environmental change and the response of the robot. In addition, each pixel has a high dynamic range (143 dB (Posch et al., 2011)) which allows robots to operate in both bright and dark environments, and in conditions with widely varying intra-scene lighting. The sensor is low-power, promoting longer operation times for untethered mobile robots.

The *neuromorphic iCub* (Bartolozzi et al., 2011) is a humanoid robot that has a vision system comprised of two event cameras. The iCub robot is supported, in software, by the Yet Another Robot Platform (YARP) middleware (Metta et al., 2006), upon which the iCub low-level and application-level modules have matured using standard cameras, and also utilized other freely available algorithms (e.g., using OpenCV). However, due to the asynchronous nature of the event-stream, and its fundamental differences from 2D frame sequences (see **Figure 1**), traditional computer vision algorithms and image processing frameworks cannot be directly applied.

This paper introduces the event-driven software libraries and infrastructure that is built upon YARP and integrates with the iCub robot. The library takes advantage of the YARP framework, which enables the distributed processing of events within multiple interchangeable modules spread across multiple networked machines. Modules include pre-processing utilities, visualization, low-level event-driven vision processing algorithms (e.g., corner detection), and robot behavior applications. These modules can be run and used by anyone for purely vision-based tasks, without the need for an iCub robot by using: pre-recorded datasets, a "stand-alone" camera with a compatible FPGA, a "stand-alone" camera with the compatible USB connection, or by contributing a custom camera interface to the open-source library. As the processing is modular, the exact method of event acquisition is transparent to the remainder of the library. This paper also describes several iCub applications that have been built upon the ED cameras and library and highlights some recent experimental results. We begin with a brief description of the current state-of-the-art in ED vision for robotics.

2. EVENT-DRIVEN VISION FOR ROBOTS

Recent work using event cameras show promising results for fast, low-latency robotic vision. The latency of an event-based visual attention was two order less than frame-based one (Rea et al., 2013). Recognition of playing-card suit was achieved as a deck was flicked through (30 ms exposure) (Serrano-Gotarredona and Linares-Barranco, 2015). Detection of a moving ball by a moving robot was achieved at rates of over 500 Hz (Glover and Bartolozzi, 2016). Visual tracking of features was shown at a rate higher than standard cameras (Vasco et al., 2016a) and also features position could be updated "between frames" of a standard camera (Kueng et al., 2016).

The extreme low-latency of event cameras enabled fast close-loop control (e.g., inverse pendulum balancing (Conradt et al., 2009) and goal keeping with 3 ms reaction time and only 4% CPU utilization (Delbruck and Lang, 2013)). High-frequency visual feedback (>1 kHz) enabled stable manipulator control at micrometer scale (Ni et al., 2012). On-board pose estimation during flips and rolls of a quadrotor has been shown to be plausible using event-driven vision (Mueggler et al., 2015). Finally, robotic navigation and mapping systems include a real-time 2-DOF SLAM system for a mobile robot (Hoffmann et al., 2013), and 6-DOF parallel tracking and mapping algorithms (Kim et al., 2016; Rebecq et al., 2016).

Some of the above experiments used the Java-based jAER (Delbruck, 2008); however, Java is typically less suited to on-line robotics due to computational overheads. jAER is also designed to process events from a camera directly connected to a single machine; however, robotics platforms have come to rely on a middleware that distributes processing over a computer network. A middleware allows the modular connection of sensors, algorithms and controls, which are shared within the robotics community to more quickly advance the state-of-the-art. Perhaps the most well known is the Robot Operating System (ROS), in which some support for event cameras has been made available.[1] In this paper, we present the open-source libraries for event camera integration with the YARP middleware that is used on iCub.

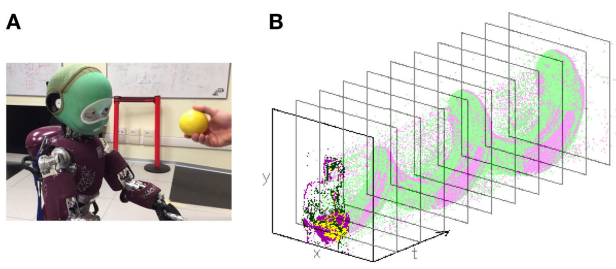


FIGURE 1 | The **(A)** iCub robot performing ball tracking and gazing toward the ball position and **(B)** the corresponding stream of events over time superimposed with approximate frame-captures for a hypothetical 10 Hz frame-based camera.

3. THE EVENT-DRIVEN LIBRARY

ED cameras encode information as a stream of asynchronous events with sub-μs resolution. When a pixel detects an illumination change beyond a threshold, it emits a digital pulse that can be assigned a timestamp and pixel address (using Address Event Representation (AER) (Mortara, 1998)) by a clock-based digital interface (e.g., FPGA or microcontroller). The entire visual information is, therefore, encoded within the relative timing and pixel position between events. An example event-stream is shown in **Figure 1**.

This event-driven library is designed to read events from the cameras, interface to communications for distributed processing, and provide event-based visual processing algorithms toward low-latency robotic vision. The library is written in C++, uses the `ev` namespace, and is integrated with the YARP middleware.

3.1. Representing an Event

The basic element representing an event is a `ev::vEvent`, which only stores the timestamp, i.e., *when* an event occurred. The

[1] github.com/uzh-rpg/rpg_dvs_ros.

information about *what* occurred is instead stored in the member variables of classes that are inherited from a `ev::vEvent`, see **Figure 2**. Events produced by the event cameras are called `ev::AddressEvent`, which consist of pixel location (x,y) and pixel polarity (p: darker/lighter) in addition to the camera channel (c: left/right). Algorithmic processing of events can be used to append additional information to an event, such as adding the velocity from an optical-flow algorithm. Currently used additional event-types include optical-flow events (`ev::FlowEvent`), class-labeled events (`ev::LabelledAE`), and events with a spatial distribution (`ev::GaussianAE`).

An instantiated `ev::vEvent` is wrapped in a C++11 `shared_ptr` such that memory is automatically managed, and events can be referenced in multiple threaded environments without duplicated memory allocation. The event-driven library provides a set of wrapper functions to ensure the `shared_ptrs` are correctly handled (see **Listing 1**).

These event-types can be easily extended through inheritance, and by defining the required additional member variables. Packet encoding and decoding methods are also required for transmission (described below). The framework is designed to be fully future compatible with the integration of different event-driven sensors (e.g., tactile and audio) by extending the base `ev::vEvent` class.

3.2. Event-Packets in YARP

On the iCub robot, a Linux driver reads the events from the camera FPGA interface and the `zynqGrabber` module exposes the data on a YARP port. A packet of events is sent in a `ev::vBottle` (a specialized type of `yarp::os::Bottle`) such that the bit-coding of the AER is preserved: to retain data-compression and compatibility with other AER-based hardware. A module that receives a `ev::vBottle` can decode the AER and instantiate a `ev::vEvent` easily, as event decoding is provided by each

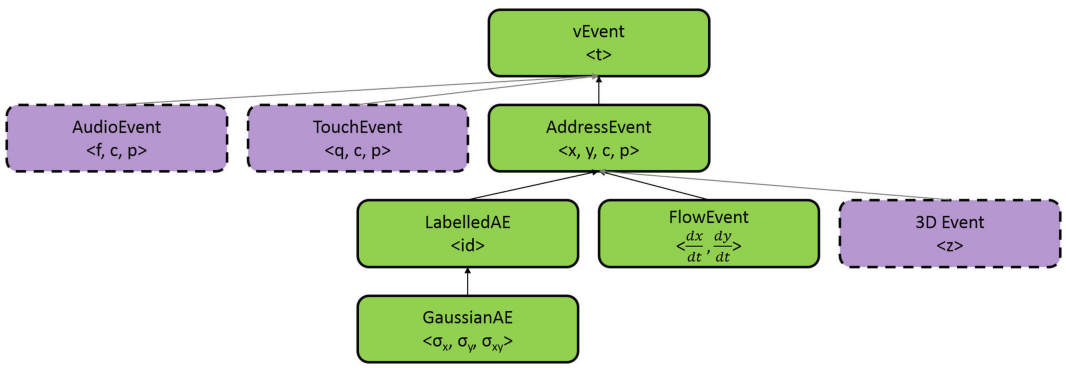


FIGURE 2 | Event-types and inheritance, purple/dashed boxes show possible additions to the library to support the integration of other sensory modalities and information from additional computing modules.

```
using namespace ev;

//create a new event in the centre of the ATIS sensor
event<AddressEvent> v1 = make_event<AddressEvent>();
v1->stamp = 0; v1->x = 152; v1->y = 120;

//upgrade the event to a labelledAE with a label of 1 (we make use of the
   keyword: auto)
auto v2 = make_event<labelledAE>(v1);
v2->id = 1;

//recast the event dynamically (using AE shorthand for AddressEvent)
auto v3 = as_event<AE>(v2);

//or recast statically (faster but less safe)
auto v4 = is_event<AE>(v2);
```

LISTING 1 | Instantiating events using shared pointer wrappers and dynamic casting. The outcome of the code-snippet will be the allocation of `v1` as an `ev::AddressEvent` and (an identical) `v2` as a `ev::labelledAE`, while `v3` and `v4` will be pointers to `v2`, but interpreted as `ev::AddressEvents`.

event class. Encoding/decoding typically involves bit-shifts and a typecast to interpret a specific range of bits as the correct data type. The decoded events are stored in a `ev::vQueue` which wraps a `std::deque<event<vEvent>>`. The procedure to obtain the event-stream is, therefore, transparent to the processing module. Reading `ev::vBottle` from a port is typically done using callback functionality (i.e., only where data is present) as the event-stream is asynchronous. An example code-snippet is provided **in Listing 2**.

Events can be saved and loaded from a file using the standard tools in YARP as an event-packet is fully interpretable as a standard `yarp::os::Bottle`. Therefore, it is easy to save a dataset using the `yarpdatadumper` and replay it using the `yarpdataplayer`. This is done externally to the event-driven library, simply by connecting the event-stream to/from the aforementioned modules using YARP connections.

3.3. Structuring the Event-Stream

The desired approach to ED processing is to perform a small, lightweight computation as each event is received; however, a single event (a single pixel) does not provide sufficient information on its own for many complex visual algorithms. Often it is necessary to store a sequence of events in order to extract useful information from their spatiotemporal structure. The type of structure used depends on the conditions, limitations and assumptions of the task or algorithm. For example, the length (in time) of a *ev::Temporal Window* can be tuned to respond to a target object moving at a certain velocity, but may fail if the target's velocity cannot be constrained. A *ev::Fixed Surface* of *N* events will be invariant to the speed of an object, but can fail if the target size and shape are unknown, a *ev::Surface* can access a spatial region-of-interest faster than a *ev::Temporal Window*, as long as the temporal order of events is not important. The event-driven library includes a range of methods to organize and structure the event-stream; an example code-snippet that combines port-callback functionality, event-packet decoding and event data structuring is shown in **Listing 2**.

In a distributed processing network, network latency, packet loss, and module synchronization become relevant issues, especially when it is desirable to take advantage of the intrinsic low-latency of the sensor. Processing needs to be performed in real-time to ensure robot behavior is decided from an up-to-date estimation of the world. The ATIS cameras will produce ≈10 kV/s when a small object is moving in the field of view but will produce >1,000 kV/s if the camera itself is rotated quickly (e.g., when the iCub performs a saccade). These numbers double for a stereo camera set-up. Real-time constraints can be broken if processing algorithms are dependent on processing every single event and the processing power is not sufficient.

```
using namespace ev;

class vManager : public yarp::os::BufferedPort<ev::vBottle> {

   ev::vSurface surface;

   //define the callback function of a yarp::os::BufferedPort<ev::vBottle>
      in the derived class "vManager"
   void vManager::onRead(ev::vBottle &inputBottle)
   {
      //decode only AddressEvents into a ev::vQueue
      vQueue q = inputBottle.get<AE>();

      //iterate over the events in the packet
      for(vQueue::iterator qi = q.begin(); qi != q.end(); qi++)
      {
         //static_ptr_cast to an ev::AddressEvent
         auto v = is_event<AE>(*qi);
         surface.addEvent(v);
      }

      //get the list of events on a 3x3 surface around the most recent
         event added to the surface
      ev::vQueue surfaceList = surface.getSurf(1);
   }
};
```

LISTING 2 | An example class for reading, decoding, and structuring events. This code will produce a small "surface" of events decoded from the AER representation automatically using the `ev::vBottle::get()` command, and the `ev::vBottle` is read asynchronously as the packets arrive.

In the YARP event-driven library, a multi-threaded event structure is provided to de-couple the process of reading events into a data structure from that of running the algorithm. Modules are constructed such that the entire history of events is accounted for, but the processing algorithm runs only at the rate at which it maintains real-time operation. The result is that chunks of events are not *randomly* lost within the communication pipeline; instead the rate at which the algorithm can output a result is reduced under high event-load. Our algorithms still typically run at rates of 100 to 1,000 s of Hertz; higher than the frame-rate of a standard camera. Importantly, the algorithm update-rate is not bottlenecked by the sensor update-rate (e.g., a frame-based camera), and the update-rate can be increased by adding computational power. The library classes `ev::queueAllocator`, `ev::tWinThread` and `ev::hSurfThread` manage real-time operation, and examples can be found in the documentation.

"Event-by-event" processing is also always possible in the YARP event-driven library and can be used to enforce a deterministic result to evaluate algorithm performance off-line, without the need to consider real-time constraints.

3.4. Low-Level Processing

Processing modules take the raw AER data and extract useful, higher-level information. The output of the modules will be a stream of events augmented with the additional information, as in **Figure 2**. The modules currently available in the event-driven repository are:

- Optical Flow—an estimate of visual flow velocity is given by the rate at which the position of events change over time. Local velocity can be extracted by fitting planes to the resulting spatiotemporal manifolds (Benosman et al., 2014). The vFlow module converts the ED camera output `ev::AddressEvent` to `ev::FlowEvent`.
- Cluster Tracking—The movement of an object across the visual field of an ED camera produces a detailed, unbroken trace of events. This module tracks clusters of events that belong to the same trace (Valeiras et al., 2015). The cluster center and distribution statistics is output from the `vCluster` module as a `ev::GaussianAE` event.
- Corner Detection—using an event-driven Harris algorithm, the event-stream is filtered to leave only the events on the corners of objects (Vasco et al., 2016a). Corner events provide unique features that can be tracked over time. Compared to a traditional camera, the ED corner algorithm requires less processing, as shown in **Figure 3B**. Corner events are represented by `ev::LabelledAE`s.
- Circle Detection—detection of circular shapes in the event-stream can be performed using an ED Hough transform. As the camera moves on a robot, many background events clutter the detection algorithm. The `vCircle` module reduces the false positive detections by incorporating optical-flow information (Glover and Bartolozzi, 2016). The detection results are shown in **Figure 3A**. Circle events are described by `ev::GaussianAE`s.
- Particle filtering—the particle filter achieves tracking that is robust to variations in the speed of the target, by also sampling within the temporal dimension (Glover and Bartolozzi, 2017). Ball tracking is implemented and the results are shown in **Figure 3A**.

The library also includes additional tools for:

- Camera Calibration—the intrinsic parameters of the camera can be estimated using a static fiducial and standard visual geometry techniques.
- Pre-processing—this module can apply a salt-and-pepper filter, flipping horizontal/vertical axes, applying camera distortion removal, and splitting a combined stereo event-stream into a left stream and a right stream.
- Visualization—the event-stream is asynchronous and does not inherently form "images" that can be viewed in the same way as a traditional camera. The `vFramer` uses a `ev::TemporalWindow` to accumulate events over time and produce a visualization of the event-stream. Different drawing methods exist for different event-types, which can be overlaid onto a single image (as shown in **Figure 1**).

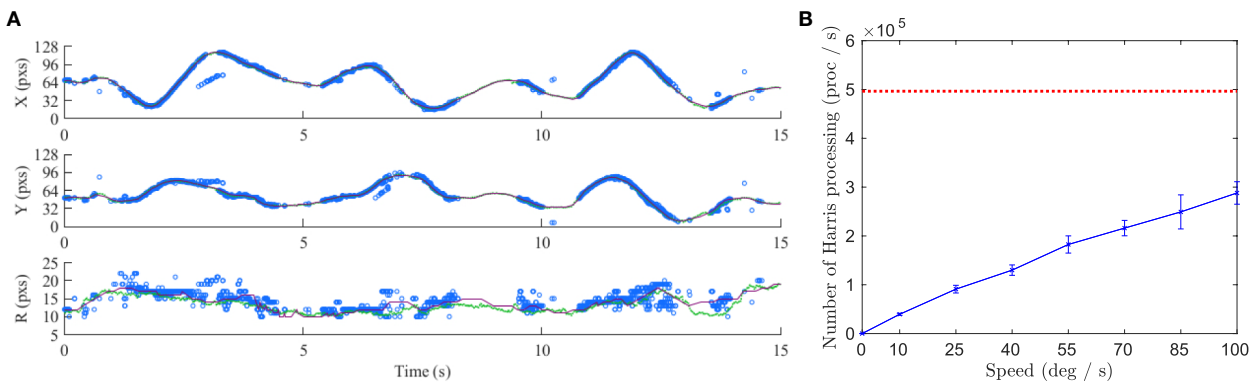


FIGURE 3 | The **(A)** tracking/detection accuracy of `ev::vCircle` (blue) and `ev::vParticleFiler` (green) compared to ground truth (black). Both algorithms can be used to control the iCub in gaze and grasping demonstrations. The **(B)** computation performance comparison between `ev::vCorner` and hypothetical frame-based Harris corner detection (red dashed line). A lower computational requirement is beneficial to resource contained on-line robotics. These figures can be generated using the scripts found in the datasets section.

4. DEMONSTRATIONS, CODE, AND DATASETS

The neuromorphic iCub and event-driven library have been used for several studies and robot demonstrations that can be run using `yarpmanager`. The modules are designed such that the robot begins performing the task once all required modules are running and the port connections have been made. Detailed instructions on how to run the demonstrations are provided in the online documentation[2] available with the code[3] on GitHub. An `xml` file is provided for each application to correctly launch and connect modules in `yarpmanager`. Known issues with the applications can also be found online. An overview of some of the applications is given below:

- **Ball Gazing and Grasping**—The module `vCircle` (desribed more in Glover and Bartolozzi (2016)) or `vParticleFilter` (described more in Glover and Bartolozzi (2017)) can be used to produce events describing the visual position of a ball, e.g., see **Figure 3A**. The `vGazeDemo` uses the `iKinGazeCtrl` (Roncone et al., 2016) to calculate the 3D position of the ball and the focus of the iCub's gaze can be directed to the location using the head and eye motors. Alternatively, the output of the ball position can be sent to the classic `DemoRedBall`[4] application to have the robot also move the arm to grasp the ball.
- **Stereo Vergence**—Automatic control of stereo vergence of the iCub to focus on an object within the field of view was implemented using biologically inspired methods (Vasco et al., 2016b). The `vVergence` application accepts stereo ev::AddressEvents and moves the vergence to minimize the response of stereo Gabor filters.
- **Attention and Micro-saccade**—A simple, yet effective, attention module is demonstrated that only requires the presence of events to perform a saccade to gaze at an external stimulus. If the event-rate is instead below a threshold, the `autosaccade` application generates small eye movements to visualize the static scene.

The documentation includes a project overview, instructions to run demonstration applications, descriptions and parameters of processing modules, and class and function descriptions. The code is only dependent on YARP and uses the `iCubContrib` to install the project in a manner compatible with YARP and iCub environment.

Datasets of event-driven data can be found in the tutorials section of the online documentation. The datasets consist of the event-streams used in several of the experiments presented in this paper. The datasets enable the processing of event-driven algorithms if a physical camera is not available.

5. CONCLUSION

This paper presents the YARP-integrated event-driven library, specifically toward enabling ED robotics using a robot middleware. The data structures, multi-threaded approach and algorithm design are aimed toward real-time operation under a wide range of conditions and in uncontrolled environment, toward robust robotic behavior. The paper has presented the YARP interface, the low-level vision algorithms, and the applications on the iCub robot.

Event cameras are now available as an add-on plug-in and new iCub robots can potentially come equipped with neuromorphic hardware; alongside traditional cameras, or as the sole form of vision. Alternatively, the software package can be used through a USB interface to the ATIS camera, or through off-line datasets. The contribution of alternative camera interfaces is possible (and welcome) as the processing modules are transparent to the source of the events, and the package is provided as an open-source project.

AUTHOR CONTRIBUTIONS

All authors contributed to the writing and proofing of the article, as well as documentation of the code. CB was the major contributor to the hardware interfaces and AG was the major contributor to the libraries and applications. VV and MI contributed to modules and applications.

ACKNOWLEDGMENTS

The authors would like to thank Ugo Pattacini, Charles Clerq, and Francesco Rea for early contributions to the event-driven libraries, and Francesco Diotalevi, Marco Maggiali, and Andrea Mura for hardware and FPGA development, and for the integration of event cameras on the iCub.

FUNDING

This research has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 231467 (eMorph) and no. 284553 (SICODE).

[2] http://robotology.github.io/event-driven/doxygen/doc/html/index.html.
[3] https://github.com/robotology/event-driven.
[4] https://github.com/robotology/icub-basic-demos.

REFERENCES

Bartolozzi, C., Rea, F., Clercq, C., Hofstätter, M., Fasnacht, D., Indiveri, G., et al. (2011). "Embedded neuromorphic vision for humanoid robots," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)* (Colorado Springs, CO: IEEE), 129–135.

Benosman, R., Clercq, C., Lagorce, X., Ieng, S.-H., and Bartolozzi, C. (2014). Event-based visual flow. *IEEE Trans. Neural Netw. Learn. Syst.* 25, 407–417. doi:10.1109/TNNLS.2013.2273537

Conradt, J., Cook, M., Berner, R., Lichtsteiner, P., Douglas, R. J., and Delbruck, T. (2009). "A pencil balancing robot using a pair of AER dynamic vision sensors," in *IEEE International Symposium on Circuits and Systems* (Taipei, Taiwan), 781–784.

Delbruck, T. (2008). "Frame-free dynamic digital vision," in *Proceedings of International Symposium on Secure-Life Electronics, Advanced Electronics for Quality Life and Society* (Tokyo, Japan), 21–26.

Delbruck, T., and Lang, M. (2013). Robotic goalie with 3 ms reaction time at 4% CPU load using event-based dynamic vision sensor. *Front. Neurosci.* 7:223. doi:10.3389/fnins.2013.00223

Glover, A., and Bartolozzi, C. (2016). "Event-driven ball detection and gaze fixation in clutter," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Daejeon, South Korea), 2203–2208.

Glover, A., and Bartolozzi, C. (2017). "Robust visual tracking with a freely-moving event camera," in *IEEE International Conference on Intelligent Robots and Systems* (Vancouver, Canada: IEEE).

Hoffmann, R., Weikersdorfer, D., and Conradt, J. (2013). "Autonomous indoor exploration with an event-based visual SLAM system," in *European Conference on Mobile Robots, ECMR 2013 – Conference Proceedings* (Barcelona, Spain), 38–43.

Kim, H., Leutenegger, S., and Davison, A. J. (2016). "Real-time 3D reconstruction and 6-DoF tracking with and event camera," in *European Conference on Computer Vision*, Amsterdam, 349–364.

Kueng, B., Mueggler, E., Gallego, G., and Scaramuzza, D. (2016). "Low-latency visual odometry using event-based feature tracks," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Daejeon, South Korea).

Lichtsteiner, P., Posch, C., and Delbruck, T. (2008). An 128x128 120dB 15μs-latency temporal contrast vision sensor. *IEEE J. Solid State Circuits* 43, 566–576. doi:10.1109/JSSC.2007.914337

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 043–048. doi:10.5772/5761

Mortara, A. (1998). "A pulsed communication/computation framework for analog VLSI perceptive systems," in *Neuromorphic Systems Engineering*, ed. T. Lande (Norwell, MA: Kluwer Academic), 217–228.

Mueggler, E., Gallego, G., and Scaramuzza, D. (2015). "Continuous-time trajectory estimation for event-based vision sensors," in *Proceedings of Robotics: Science and Systems*, Rome. doi:10.15607/RSS.2015.XI.036

Ni, Z., Bolopion, A., Agnus, J., Benosman, R., and Régnier, S. (2012). Asynchronous event-based visual shape tracking for stable haptic feedback in microrobotics. *IEEE Trans. Robot.* 28, 1081–1089. doi:10.1109/TRO.2012.2198930

Posch, C., Matolin, D., and Wohlgenannt, R. (2011). A QVGA 143 dB dynamic range frame-free PWM image sensor with lossless pixel-level video compression and time-domain CDS. *IEEE J. Solid State Circuits* 46, 259–275. doi:10.1109/JSSC.2010.2085952

Rea, F., Metta, G., and Bartolozzi, C. (2013). Event-driven visual attention for the humanoid robot iCub. *Front. Neurosci.* 7:234. doi:10.3389/fnins.2013.00234

Rebecq, H., Horstschaefer, T., Gallego, G., and Scaramuzza, D. (2016). EVO: a geometric approach to event-based 6-DoF parallel tracking and mapping in real-time. *IEEE Robot. Autom. Lett.* 2, 593–600. doi:10.1109/LRA.2016.2645143

Roncone, A., Pattacini, U., Metta, G., and Natale, L. (2016). "A cartesian 6-DoF gaze controller for humanoid robots," in *Proceedings of Robotics: Science and Systems*, AnnArbor. doi:10.15607/RSS.2016.XII.022

Serrano-Gotarredona, T., and Linares-Barranco, B. (2015). Poker-DVS and MNIST-DVS. Their history, how they were made, and other details. *Front. Neurosci.* 9:481. doi:10.3389/fnins.2015.00481

Valeiras, D. R., Lagorce, X., Clady, X., Bartolozzi, C., Ieng, S.-H., and Benosman, R. (2015). "An asynchronous neuromorphic event-driven visual part-based shape tracking," in *IEEE Transactions on Neural Networks and Learning Systems*, 1–15. Available at: http://ieeexplore.ieee.org/document/7063246/

Vasco, V., Glover, A., and Bartolozzi, C. (2016a). "Fast event-based Harris corner detection exploiting the advantages of event-driven cameras," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Daejeon, South Korea), 4144–4149.

Vasco, V., Glover, A., Tirupachuri, Y., Solari, F., Chessa, M., and Bartolozzi, C. (2016b). "Vergence control with a neuromorphic iCub," in *IEEE-RAS International Conference on Humanoid Robots (Humanoids)* (Cancun, Mexico: IEEE), 732–738.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

CODE
published: 12 February 2018
doi: 10.3389/frobt.2018.00010

Speech Recognition for the iCub Platform

*Bertrand Higy[1,2], Alessio Mereta[3], Giorgio Metta[1] and Leonardo Badino[4]**

[1] iCub Facility, Istituto Italiano di Tecnologia, Genoa, Italy, [2] Università di Genova, Genoa, Italy, [3] Advanced Concepts Team, European Space Agency, Noordwijk, Netherlands, [4] Center for Translational Neurophysiology of Speech and Communication, Istituto Italiano di Tecnologia, Ferrara, Italy

This paper describes open source software (available at https://github.com/robotology/natural-speech) to build automatic speech recognition (ASR) systems and run them within the YARP platform. The toolkit is designed (i) to allow non-ASR experts to easily create their own ASR system and run it on iCub and (ii) to build deep learning-based models specifically addressing the main challenges an ASR system faces in the context of verbal human–iCub interactions. The toolkit mostly consists of Python, C++ code and shell scripts integrated in YARP. As additional contribution, a second codebase (written in Matlab) is provided for more expert ASR users who want to experiment with bio-inspired and developmental learning-inspired ASR systems. Specifically, we provide code for two distinct kinds of speech recognition: "articulatory" and "unsupervised" speech recognition. The first is largely inspired by influential neurobiological theories of speech perception which assume speech perception to be mediated by brain motor cortex activities. Our articulatory systems have been shown to outperform strong deep learning-based baselines. The second type of recognition systems, the "unsupervised" systems, do not use any supervised information (contrary to most ASR systems, including our articulatory systems). To some extent, they mimic an infant who has to discover the basic speech units of a language by herself. In addition, we provide resources consisting of pre-trained deep learning models for ASR, and a 2.5-h speech dataset of spoken commands, the VoCub dataset, which can be used to adapt an ASR system to the typical acoustic environments in which iCub operates.

Keywords: automatic speech recognition, yarp, tensorflow, code:python, code:matlab, code:C++

OPEN ACCESS

Edited by:
Lorenzo Jamone, Queen Mary University of London, United Kingdom

Reviewed by:
Tadahiro Taniguchi, Ritsumeikan University, Japan
Anna Pribilova, Slovak University of Technology in Bratislava, Slovakia

***Correspondence:**
Leonardo Badino
leonardo.badino@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *09 August 2017*
Accepted: *23 January 2018*
Published: *12 February 2018*

Citation:
Higy B, Mereta A, Metta G and Badino L (2018) Speech Recognition for the iCub Platform. Front. Robot. AI 5:10. doi: 10.3389/frobt.2018.00010

1. INTRODUCTION

Several applications use speech to give instructions to iCub, often relying on proprietary software. However, the robot operates in specific conditions where those systems may perform poorly. An open and easy-to-use system that would reliably recognize commands in this context would thus be a very desirable tool. We present here a first codebase, henceforth *iCubRec*, which has been built to provide such services to the community of iCub users. It allows to train and run state-of-the-art deep neural network (DNN)-based automatic speech recognition (ASR).

As an additional contribution, a second codebase, henceforth *bioRec*, allows to experiment with novel DNN-based recognition systems that share the same bio-inspired and developmental learning view that gave birth to iCub (Lungarella et al., 2003). *bioRec* is self-contained and independent of *iCubRec*, however its DNN-based acoustic models can effortlessly be used within *iCubRec*.

Finally, in addition to the code, we are also providing resources to facilitate the implementation of a command recognizer: (i) the VoCub dataset, a dataset of registered vocal commands and (ii) pre-trained Gaussian Mixture Model (GMM)- and DNN-based acoustic models to perform recognition.

Our code, as well as the resources, is released under GPLv3 license. The code is available at https://github.com/robotology/natural-speech (doi: 10.5281/zenodo.1064043).

2. iCubRec

2.1. Application and Utility

An ASR system for iCub typically operates in challenging conditions. We have identified three specific factors which we want the system to be robust to:

- noise; the robot often operates in noisy environments (e.g., noisy servers and computers running, concurrent speakers, the robot itself generating noise).
- accents; the teams working with iCub are international and the robot needs to recognize spoken commands uttered with a wide variety of foreign accents.
- distance and movement; distant speech recognition is an important research topic in ASR and has been the focus of many recent challenges (e.g., the Chime4 challenge[1]). When the speaker–microphone distance increases, the speech signal-to-noise ratio decreases and signal distortions due to reverberation (in indoor environments) increases. A non-fixed distance, due to a moving speaker and/or microphone, adds further complexity to the task.

Although deep learning has recently produced excellent results in ASR, it still suffers the training-testing mismatched conditions problem. Proprietary ASR systems may perform poorly in the aforementioned acoustic/speech conditions mainly because such conditions are not well covered by their training datasets. We have addressed this problem by building a dataset (VoCub dataset) that covers such conditions and by providing tools to easily adapt a DNN to it.

Other than robust, an ASR system for iCub should be easy-to-use, open, and modular. Usability is necessary to allow all iCub mindware developers, who mostly have no ASR background, to train and run ASR on iCub. For this reason, we provide pre-trained GMM- and DNN-based acoustic models that can be used out of the box with the existing code. At the same time, we want more advanced users to easily modify and adapt the code to their own needs. This can only be done if everything is open and well modularized.

2.2. Methods

To facilitate the understanding of the *iCubRec* module for non-ASR experts we provide here the definition of few basic ASR terms. A standard ASR system consists of 4 main parts: an acoustic feature extraction step which extracts spectral features from the input acoustic waveform; an acoustic model which relates the extracted features to sub-words (e.g., phonemes, such as consonants and vowels) and then words (i.e., computes the likelihood that vectors of features are generated by a candidate word); a language model, which is independent of the acoustic signals and incorporates prior knowledge about a specific language (e.g., the probability that the word "barks" follows the word "dog"); and a speech decoder which performs word recognition by computing the most probable sequence of words of the utterance, given: (a) the acoustic model; (b) the language model; (c) the dictionary, which consists of all words the system has to recognize along with their phoneme transcriptions. Acoustic modeling is usually done using a Hidden Markov Model (HMM) which is well suited for sequential data like speech. HMMs combine transition probabilities (i.e., $p(s_t \mid s_{t-1})$ where s_t is a phone label at time t) with observation probabilities (i.e., $p(o_t \mid s_t)$, where o_t is the input vector of acoustic feature at time t). The core difference between classical GMM-HMM vs. hybrid DNN-HMM acoustic models simply resides on whether GMMs or DNNs are used to compute the observation probabilities.

2.3. Code Description

iCubRec code is based on the Hidden Markov Model Toolkit (HTK) (Young et al., 2015). However, as the training capabilities for DNNs are still quite limited in HTK, we also consider the alternative possibility to train a network with Tensorflow (Abadi et al., 2015) and convert it to HTK format for use in decoding. Although in the later case the DNN is still restricted to the architectures recognized by HTK (for now, only feedforward networks with a limited set of activation functions), this gives more flexibility and control over the training process. Additionally, the use of Tensorflow allows to easily adapt a pre-trained DNN to new adaptation data.

The code consists of scripts for:

- acoustic model training with GMMs
- acoustic model training with DNNs
- speech decoding
- integration within YARP for online speech decoding.

iCubRec is a combination of Python 3, Perl and shell scripts, and was written for HTK 3.5 and Tensorflow 1.0.

2.3.1. GMM-Based Acoustic Modeling

Before the advent of DNNs, GMM-HMM systems were state-of-the-art for acoustic modeling in speech recognition. Although they are significantly outperformed by neural networks (Dahl et al., 2012; Seltzer et al., 2013), GMMs are still widely used if only to compute the phone labels/speech segments alignments needed to train a DNN (Dahl et al., 2012). The folder `gmm_training` provides a set of scripts to train GMM-HMMs using HTK. These scripts are based on Keith Vertanen's code (Vertanen, 2006) and allow to build models similar to the ones described by Woodland et al. (1994). The recipe is originally intended for TIMIT (Garofolo et al., 1993a) and Wall Street Journal (WSJ) (Garofolo et al., 1993b) datasets and has been adapted for the Chime4 challenge (Vincent et al., 2016) and VoCub datasets.

2.3.2. DNN-Based Acoustic Modeling

Once the speech signal has been aligned (presumably using GMM-HMMs), a DNN-based model can be trained. Two

[1] http://spandh.dcs.shef.ac.uk/chime_challenge/chime2016/.

alternatives are available: (i) using the scripts in `dnn_training/htk` to train a model with HTK or (ii) using the code under `dnn_training/tf` to train the net with Tensorflow. The scripts proposed here are currently restricted to TIMIT and WSJ, but support for additional datasets will be added soon.

2.3.3. Speech Decoding

With a model trained with HTK (GMM-based or DNN-based), it is then straightforward to perform recognition on a new utterance. The folder `offline_decoding` provides an example of decoding on pre-recorded data with HTK. Additionally, `export for_htk.py` shows how to easily extract the parameters of a net trained with Tensorflow and convert them into HTK format.

2.3.4. Integration with YARP

All the code presented so far is meant to train and test a system offline. `yarp_decoding` folder provides the modules necessary to use an existing model within YARP and perform online recognition. A streaming service based on yarp.js[2] allows to record sound from any device equipped with a microphone and a web browser. Two other modules are provided: `rctrld_yarphear_asr` which saves the recorded data in a file, and the `decoder` (based on HVite tool from HTK) for feature extraction and command decoding. The application `speechrec.xml` is available to easily run and connect all the modules.

2.4. Resources

2.4.1. The VoCub Dataset

Recording a dataset has two main advantages: (i) it allows to easily test the recognition system and to reliably estimate its performance in real conditions and (ii) can be used to adapt the system in order to reduce the training/testing mismatch problem. For this reasons, we have recorded examples of the commands we want to recognize within real-usage scenarios. That resulted in the VoCub dataset.[3]

The recordings consist of spoken English commands addressed to iCub. There are 103 unique commands (see **Table 1** for some examples), composed of 62 different words. We recorded 29 speakers, 16 males and 13 females, 28 of them are non-native English speakers. We finally obtained 118 recordings from each speaker: of the 103 unique commands, 88 were recorded once, and 15 twice (corresponding to sentences containing rare words). This results in about 2 h and 30 min of recording in total.

A split of the speakers into training, validation, and test sets is proposed with 21, 4, and 4 speakers per set, respectively. The files are organized with the following convention `setid/spkrid/spkrid_cond_recid.wav`, where:

- `setid` identifies the set: `tr` for training, `dt` for validation and `et` for testing.
- `spkrid` identifies the speaker: from `001` to `021` for training, `101` to `104` for validation and `201` to `204` for testing.
- `cond` identifies the condition (see below).
- `recid` identifies the record within the condition (starting from `0` and increasing).

The commands were recorded in two different conditions, a non-static (`cond` = 1) and a static condition (`cond` = 2), with an equal number of recorded utterances per condition.

In the static condition, the speaker sat in front of two screens where the sentences to read were displayed. In the non-static condition, the commands were provided to the subject verbally through a speech synthesis system, and the subject had to repeat them while performing a secondary manual task. This secondary task was designed to be simple enough to not impede the utterance repetition task, while requiring people to move around the robot. The distance between the speaker and the microphone in this last condition ranges from 50 cm to 3 m.

We also registered a set of additional sentences for the testing group (same structure but different vocabulary) to test the recognition system for new commands not seen during training. The sentences consist of 20 new commands, pronounced by each speaker of the test set twice: once in non-static condition (`cond` = 3) and once in static condition (`cond` = 4).

2.4.2. Trained Models

As not all the datasets used in our scripts are freely available, and in order to ease the use of our system, we provide pre-trained acoustic models that can be used out of the box. The `models/README.md` file contains links to download GMM-based models trained on WSJ, Chime4 and VoCub datasets, and DNN-based models trained on TIMIT and WSJ. Additional DNN-based models will be added in the future. Further details about the different models and the precise training procedure can be found in the same file.

2.5. Example of Use

A good demonstration of the capabilities of the code presented so far is given in the file `icubrec/DEMO.md`. In a few simple steps, the user is shown how to perform offline decoding on the VoCub dataset with a pre-trained model. This example is accessible to novice ASR users and does not require any proprietary dataset.

A more in-depth example is given in `icubrec/TUTORIAL.md`, which provides detailed instruction on how to train a full ASR system on the WSJ dataset. This tutorial goes through all the main steps: training of a GMM-based acoustic model, computation of the alignments, training of a DNN-based acoustic model using those alignments, and finally decoding of the test sentences.

[2] https://github.com/robotology/yarp.js.

[3] Freely available at https://robotology.github.io/natural-speech/vocub/.

TABLE 1 | Examples of the commands used in the VoCub dataset.

I will teach you a new object.
This is an octopus.
What is this?
Let me show you how to reach the car with your left arm.
Let me show you how to reach the turtle with your right arm.
There you go.
Grasp the ladybug.
Where is the car?
No, here it is.
See you soon.

3. bioRec

3.1. Application and Utility

Our module for bio- and cognitive science-inspired ASR is composed of two distinct parts serving different purposes: *Articulatory Phone Recognition* and *Unsupervised/Developmental ASR*.

3.1.1. Articulatory Phone Recognition

This part includes modules `phonerec` and `pce_phonerec`, which build *articulatory* phone recognition systems. A phone recognition system recognizes the sequence of phones of an utterance. It can roughly be identified as an ASR system without language model and dictionary. *Articulatory* phone recognition uses prior information about how the vocal tract moves when producing speech sounds. This *articulatory* view is strongly motivated by influential neurobiological theories of speech perception that assume a contribution of the brain motor cortex to speech perception (Pulvermüller and Fadiga, 2010) and have been shown to outperform strong DNN-based baselines where no prior articulatory information is used (see, e.g., Badino et al. (2016)).

3.1.2. Unsupervised/Developmental ASR

The second part of *bioRec*, `zerorchallenge`, builds "unsupervised" ASR systems. Most recognition systems, including the *articulatory* systems, are trained on supervised data, where training utterances are associated to phonetic transcriptions, and the inventory of phones is given. This learning setting is far easier than the learning setting of an infant who has to acquire her native language and has to discover the basic units of the language on her own. In order to better understand how an infant can acquire the phone inventory during development from raw "unsupervised" utterances, we have created "unsupervised" ASR systems that were submitted and evaluated at the 1st Zero Resource Speech Challenge (ZRS challenge) (Versteegh et al., 2015).

3.2. Methods

3.2.1. Articulatory Phone Recognition

The *articulatory* phone recognition module consists of 2 parts depending on how speech production information is represented:

- `phonerec`; speech production is represented in the form of actual measurements of vocal tract movements, collected through instruments such as the electromagnetic articulograph (Richmond et al., 2011);
- `pce_phonerec`; vocal tract movements are initially described by discrete linguistic features and actual measurements are not used.

`phonerec`: in this module, prior information of speech production is built by learning, during training, an acoustic-to-articulatory mapping that allows to recover vocal tract movements, i.e., reconstructed articulatory features (AFs), from the acoustic signal (Badino et al., 2012, 2016). The reconstructed AFs are then appended to the usual input acoustic vector of the DNN that computes phone state posterior probabilities, i.e., the acoustic model DNN (see **Figure 1**, which shows the simplest strategy). Additionally, our code allows to apply autoencoder (AE)-based transformations to the original AFs in order to improve performance. AEs are a special kind of DNN that attempts to reconstruct its input after encoding it, typically through a lossy encoding. More details and evaluation results can be found in Badino et al. (2016).

`pce_phonerec`: in this module, AFs are derived (through a DNN) from linguistic discrete features (referred to as phonetic

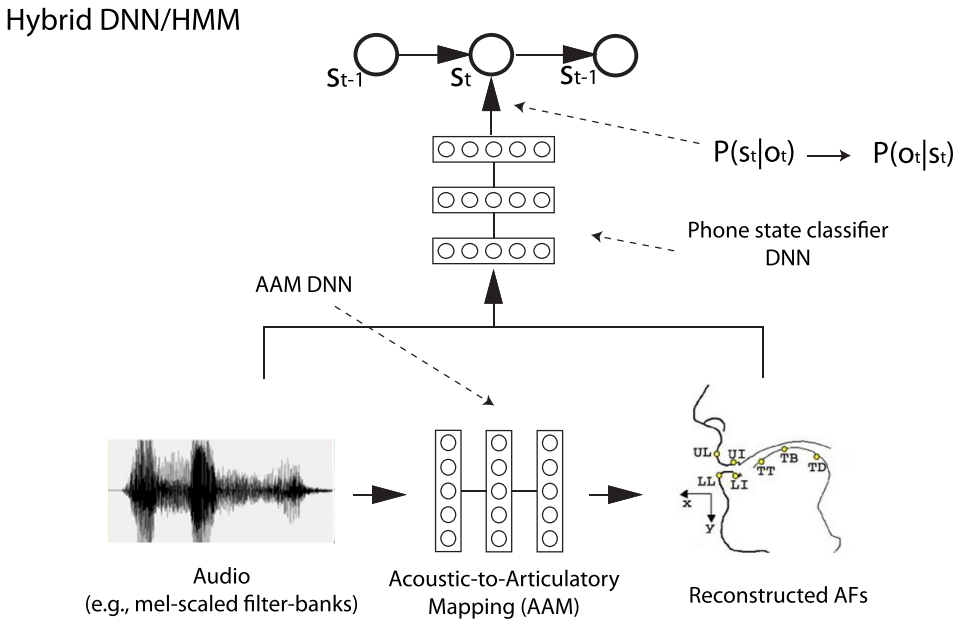


FIGURE 1 | An example of articulatory phone recognition. Here, the simplest strategy available in `phonerec` is shown. o_t is a vector of acoustic features, while s_t is a phone state.

context embedding). They are used as secondary target for the acoustic model DNN within a multi-task learning (MTL) strategy (Caruana, 1997). This strategy forces the DNN to learn a motor representation without the need for time-consuming collection of actual articulatory data. Our approach outperforms strong alternative MTL-based approaches (Badino, 2016).

3.2.2. Unsupervised/Developmental ASR

`zerorchallenge` is the module building the unsupervised/developmental ASR systems we submitted to Task1 of the ZRS challenge at Interspeech 2015 (Versteegh et al., 2015). The goal of the challenge was to compare systems that create new acoustic representations that can discriminate examples of minimal pairs, i.e., words differing only in one phoneme (e.g., "hat" vs. "had"), while identifying as a single entity different examples of a same word. Specifically, we focused on extracting discrete/symbolic representations, which equals to automatically discovering the inventory of (phone-like) sub-words of a language. Our core strategy is based on AEs (Badino et al., 2014), as shown in **Figure 2**. The provided scripts build 2 novel systems, one based on binarized AEs and one on Hidden Markov Model Encoders (HMM-Encoders) (Badino et al., 2015).

A binarized AE is an AE whose encoding layer nodes are binary. At each time step, it transforms a vector of real-valued acoustic features into a vector of binary units which in turn is associated to a positive integer corresponding to a discovered specific sub-word.

The HMM-Encoder combines an AE with a HMM.[4] An approach solely based on AEs ignores the sequential nature of speech and inter-sub-word dependencies. The HMM-Encoder was proposed to specifically address these potential weaknesses.

3.3. Code Description and Example of Use

All code is written in Matlab and uses the Parallel Processing Toolbox to allow fast DNN training with GPUs. All modules were tested in Matlab 2013a and 2015a.

[4]Our HMM training code is a modified version of code from K. Murphy's BayesianNet toolbox, available at https://github.com/bayesnet/bnt.

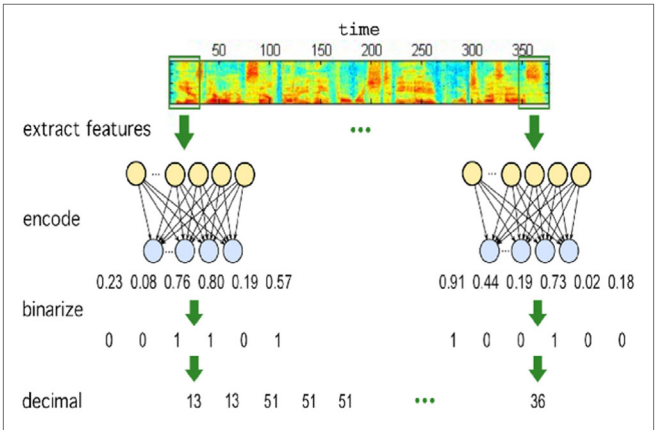


FIGURE 2 | Overview of the AE-based approach to sub-word learning.

3.3.1. Articulatory Phone Recognition

`phonerec`: the file `ploclassify.m` allows to train and test articulatory phone recognition systems. It requires the `inivar.m` configuration file where it is possible to define, e.g., the type of AFs through `cmotortype` (e.g., AE-transformed AFs or "plain" AFs), the hyperparameters of the acoustic model DNN (`parnet_classifier`), and of the acoustic-to-articulatory mapping DNN (`parnet_regress`).

The folder `demo` contains 2 examples to build and evaluate a baseline (`audio1_motor0_rec0`) and an articulatory phone recognition system (`audio1_motor3_rec1`) on the mngu0 dataset (Richmond et al., 2011). The dataset used here (available at https://zenodo.org/record/836692/files/bioRec_Resources.tar.gz, under `/bioRec_Resources/phonerec_mngu0/`) is a preprocessed version of the mngu0 dataset.

`pce_phonerec`: this articulatory phone recognition system is trained and evaluated by running `mtkpr_pce.m`. It can be compared with an alternative MTL-based strategy proposed by Microsoft researchers (Seltzer and Droppo, 2013), by running the script `mtkpr_baseline.m`. All systems are trained and tested on the TIMIT dataset, which unfortunately is not freely available. Training on different datasets would require some small dataset-dependent modifications to the look-up table used to extract discrete linguistic features from phone names.

We have created a Python + Tensorflow implementation the DNN training proposed in this module which will be soon available.

3.3.2. Unsupervised/Developmental ASR

We provide scripts that receive as input one of the datasets provided by the ZRS challenge, train one of the unsupervised ASR systems (on the training utterances), and return the testing utterances in a new discrete representation with a positive integer at each time step. We additionally provide the 3 datasets from the ZRS challenge already transformed to be processed by our scripts (available at https://zenodo.org/record/836692/files/bioRec_Resources.tar.gz, under `/bioRec_Resources/zerochallenge/`). The output format allows to evaluate the output file with the tools provided for the challenge (Versteegh et al., 2015).

3.3.3. Utilities

All utilities used by the `phonerec`, `pce_phonerec`, and `zerorchallenge` are in:

- `netutils`: contains functions to train and run DNNs, e.g., standard DNN training, Deep Belief Network-based DNN pretraining (Hinton et al., 2006), MTL training, DNN forward pass (i.e., to evaluate a DNN), deep autoencoder training, including training of some AEs we have recently proposed specifically for speech.
- `utils`: this folder contains all utilities that do not pertain to DNNs. These include: data loading and normalization, phone language models computation, Viterbi-based phone decoding, phone error rate computation, and analysis of error.

4. CONCLUSION

In this paper, we have described the codebase that allows to easily train deep neural network-based automatic speech

recognition systems and run them within YARP. As an additional contribution, we provide tools to experiment with recognition systems that are inspired by recent influential theories of speech perception and with systems that partly mimic the learning setting of an infant who has to learn the basic speech units of a language.

ETHICS STATEMENT

This study was carried out in accordance with the recommendations of the "Comitato Etico per la Sperimentazione con l'Essere Umano della ASL 3 di Genova" with written informed consent from all subjects. All subjects gave written informed consent in accordance with the Declaration of Helsinki. The protocol was approved by the "Comitato Etico per la Sperimentazione con l'Essere Umano della ASL 3 di Genova."

AUTHOR CONTRIBUTIONS

Conceived and designed the ASR systems: LB, GM, BH, and AM. Wrote the code: LB, BH, and AM. Wrote the paper: BH and LB.

FUNDING

The authors acknowledge the support of the European Commission project POETICON++ (grant agreement No. 288382) and ECOMODE (grant agreement No. 644096).

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at http://www.frontiersin.org/articles/10.3389/frobt.2018.00010/full#supplementary-material.

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. arXiv preprint arXiv:1603.04467.

Badino, L. (2016). "Phonetic context embeddings for DNN-HMM phone recognition," in *Proc. of Interspeech* (San Francisco, CA).

Badino, L., Canevari, C., Fadiga, L., and Metta, G. (2012). "Deep-level acoustic-to-articulatory mapping for DBN-HMM based phone recognition," in *Proc. of IEEE SLT* (Miami, FL).

Badino, L., Canevari, C., Fadiga, L., and Metta, G. (2014). "An auto-encoder based approach to unsupervised learning of subword units," in *Proc. of IEEE ICASSP* (Florence, Italy).

Badino, L., Canevari, C., Fadiga, L., and Metta, G. (2016). Integrating articulatory data in deep neural network-based acoustic modeling. *Comput. Speech Lang.* 36, 173–195. doi:10.1016/j.csl.2015.05.005

Badino, L., Mereta, A., and Rosasco, L. (2015). "Discovering discrete subword units with binarized autoencoders and hidden-Markov-model encoders," in *Proc. of Interspeech* (Dresden, Germany).

Caruana, R. (1997). Multitask learning. *Mach. Learn.* 28, 41–75. doi:10.1023/A:1007379606734

Dahl, G., Yu, D., Deng, L., and Acero, A. (2012). Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition. *IEEE Trans. Audio Speech Lang. Processing* 20, 30–42. doi:10.1109/TASL.2011.2134090

Garofolo, J. S., Lamel, L. F., Fisher, W. M., Fiscus, J. G., Pallett, D. S., Dahlgren, N. L., et al. (1993a). *TIMIT Acoustic-Phonetic Continuous Speech Corpus LDC93s1. Web Download*. Philadelphia: Linguistic Data Consortium.

Garofolo, J., Graff, D., Paul, D., and Pallett, D. (1993b). *CSR-I (WSJ0) Complete LDC93s6a. Web Download*. Philadelphia: Linguistic Data Consortium.

Hinton, G., Osindero, S., and Teh, Y. (2006). A fast learning algorithm for deep belief nets. *Neural Comput.* 18, 1527–1554. doi:10.1162/neco.2006.18.7.1527

Lungarella, M., Metta, G., Pfeifer, R., and Sandini, G. (2003). Developmental robotics: a survey. *Connect. Sci.* 15, 151–190. doi:10.1080/0954009031000165110

Pulvermüller, F., and Fadiga, L. (2010). Active perception: sensorimotor circuits as a cortical basis for language. *Nat. Rev. Neurosci.* 11, 351–360. doi:10.1038/nrn2811

Richmond, K., Hoole, P., and King, S. (2011). "Announcing the electromagnetic articulography (day 1) subset of the mngu0 articulatory corpus," in *Proc. of Interpseech* (Florence, Italy).

Seltzer, M., and Droppo, J. (2013). "Multi-task learning in deep neural networks for improved phoneme recognition," in *Proc. of ICASSP* (Vancouver, BC).

Seltzer, M., Yu, D., and Wan, Y. (2013). "An investigation of deep neural networks for noise robust speech recognition," in *Proc. of ICASSP* (Vancouver, BC).

Versteegh, M., Thiolliere, R., Schatz, T., Cao, X. N., Anguera, X., Jansen, A., et al. (2015). "The zero resource speech challenge 2015," in *Proc. of Interspeech* (Dresden, Germany).

Vertanen, K. (2006). *Baseline WSJ Acoustic Models for HTK and Sphinx: Training Recipes and Recognition Experiments. Technical Report*. Cambridge, UK: Cavendish Laboratory.

Vincent, E., Watanabe, S., Nugraha, A. A., Barker, J., and Marxer, R. (2016). An analysis of environment, microphone and data simulation mismatches in robust speech recognition. *Comput. Speech Lang.* 46, 535–557. doi:10.1016/j.csl.2016.11.005

Woodland, P., Odell, J., Valtchev, V., and Young, S. (1994). "Large vocabulary continuous speech recognition using HTK," in *IEEE International Conference on Acoustics, Speech, and Signal Processing, 1994. ICASSP-94*, Vol. ii (Adelaide, SA), II/125–II/128.

Young, S., Evermann, G., Gales, M., Hain, T., Kershaw, D., Liu, X., et al. (2015). *The HTK Book (for HTK Version 3.5)*. Cambridge University Engineering Department.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The handling editor declared a past co-authorship with one of the authors, GM.

frontiers in Robotics and AI

CODE
published: 16 February 2018
doi: 10.3389/frobt.2018.00005

YARP-ROS Inter-Operation in a 2D Navigation Task

Marco Randazzo, Andrea Ruzzenenti and Lorenzo Natale*

iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy

This paper presents some recent developments in YARP middleware, aimed to improve its integration with ROS. They include a new mechanism to read/write ROS transform frames and a new set of standard interfaces to intercommunicate with the ROS navigation stack. A novel set of YARP companion modules, which provide basic navigation functionalities for robots unable to run ROS, is also presented. These modules are optional, independent from each other, and they provide compatible functionalities to well-known packages available inside ROS framework. This paper also discusses how developers can customize their own hybrid YARP-ROS environment in the way it best suits their needs (e.g., the system can be configured to have a YARP application sending navigation commands to a ROS path planner, or vice versa). A number of available possibilities is presented through a set of chosen test cases applied to both real and simulated robots. Finally, example applications discussed in this paper are also made available to the community by providing snippets of code and links to source files hosted on github repository https://github.com/robotology.[1]

Keywords: YARP, autonomous navigation, SLAM, mobile robots, iCub, R1, ROS, C++ interfaces

OPEN ACCESS

Edited by:
Maxime Petit,
Imperial College London,
United Kingdom

Reviewed by:
Daniel Camilleri,
University of Sheffield,
United Kingdom
Ayse Kucukyilmaz,
University of Lincoln,
United Kingdom

****Correspondence:***
Marco Randazzo
marco.randazzo@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *20 August 2017*
Accepted: *16 January 2018*
Published: *16 February 2018*

Citation:
Randazzo M, Ruzzenenti A and Natale L (2018) YARP-ROS Inter-Operation in a 2D Navigation Task. Front. Robot. AI 5:5. doi: 10.3389/frobt.2018.00005

1. INTRODUCTION

YARP is an open-source robotics middleware, specifically designed to be modular, non-invasive, and flexible. It promotes software re-usability by means of abstract interfaces and modular software paradigms, and it allows to distribute computational tasks across a system by offering multi-platform network communication primitives (Fitzpatrick et al., 2014).

YARP development is historically correlated to the iCub robot (Metta et al., 2010; Natale et al., 2016), a child-sized humanoid platform for the study of cognitive robotics. In these years, the iCub community focused its attention on topics such as human–robot interaction, visual attention, machine learning, object manipulation, and grasping. Balancing a bipedal walking robot like iCub is a problem that has been addressed only recently by some research groups (Hu et al., 2016; Nava et al., 2016). This is the reason why a standard navigation interface was missing in YARP so far.

On the other side, ROS, an Ubuntu-based middleware developed around the PR2 wheeled robot, addressed the problem of making a mobile platform to navigate into a 2D environment from the very beginning (Quigley et al., 2009; Cousins, 2010). Over the past years, the ROS navigation stack has grown in comprehensiveness, wrapping or including bindings to basically all state-of-the-art algorithms and third-party libraries (Marder-Eppstein et al., 2010).

This paper has two goals. First, to provide the YARP community a way to re-use the massive amount of code that has been developed within the ROS community. Second, Yarp is a multi-platform framework which can run on Windows, Linux and MacOs, while ROS is currently limited

[1]http://doi.org/10.5281/zenodo.1116278.

to Ubuntu-based systems. Thus, Yarp can be used to interface applications belonging to the two different frameworks and running on different operating systems. This goal is accomplished through a set of dedicated YARP classes and interfaces, as shown in the following sections.

2. YARP/ROS INTERFACE

2.1. YARP Ports and ROS Topics

YARP inter-module communication is traditionally implemented through network objects called *ports*. In a typical usage scenario, a sender module opens an output port (identified by a symbolic name, registered onto a nameserver) and writes data to it. Analogously, a receiver module, which wants to perform a read operation, opens an input port with a different symbolic name. Sender and receiver are thus decoupled, and the user is responsible for making connections/disconnections between the two ports.

In ROS, inter-module communication is obtained through a *publisher/subscriber* paradigm, based on the concept of *topic*. The subscriber manifests its intention of receiving a specific stream of data by registering to a topic, without caring about the identity of the *node* (or nodes) that is actually publishing it. Connections are not managed by the user but by a central authority, called *ROS Master*, which also checks if publishers and receivers comply on the same data format. Indeed, ROS communication is strongly typed and it employs a set of standard formats defined in message (.msg) files.

The possibility to communicate natively with ROS has been recently integrated into YARP. Special classes such as `yarp::os::Node`, `yarp::os::Publisher`, and `yarp::os::Subscriber` have been introduced to allow a user to handle ROS topics. Additionally, a specialized converter, namely *yarpidl_rosmsg*, was developed to automatically generate C++ header files from ROS.msg files and to allow the usage of ROS data types inside YARP.

An example of a YARP module directly publishing data onto a ROS topic, without linking any external ROS library, is shown in Section I in Supplementary Material.

2.2. TransformServer and TransformClient

Tf is a ROS package which allows a distributed system to keep track of multiple coordinate frames over time. For example, a module may be able to compute and publish the transformation from reference frame /*a* to reference frame /*b* while a different module may be able to publish the transformation from frame /*b* to frame /*c*. By subscribing to the /*tf topic*, a third module can retrieve the broadcasted transforms and compute the resulting transformation from /*a* to /*c*.

This mechanism is pervasive in all ROS. Remarkable application examples are *move-it* (a motion planning framework for mobile manipulation), *Rviz* (a 3D visualization tool), and the *ROS navigation stack*. In this latter case, tf is typically used to keep track of the estimated robot position with respect to an odometry reference frame or to a map origin. Thus, it is clear that it is not possible to obtain a complete YARP-ROS integration without implementing a mechanism that is able to handle ROS frame transforms in YARP.

To overcome this limitation, we developed a YARP device called *transformServer*. *TransformServer* collects and stores frame transforms by subscribing to /*tf* and /*tf_static* topics and makes these information available to a YARP *transformClient* instance

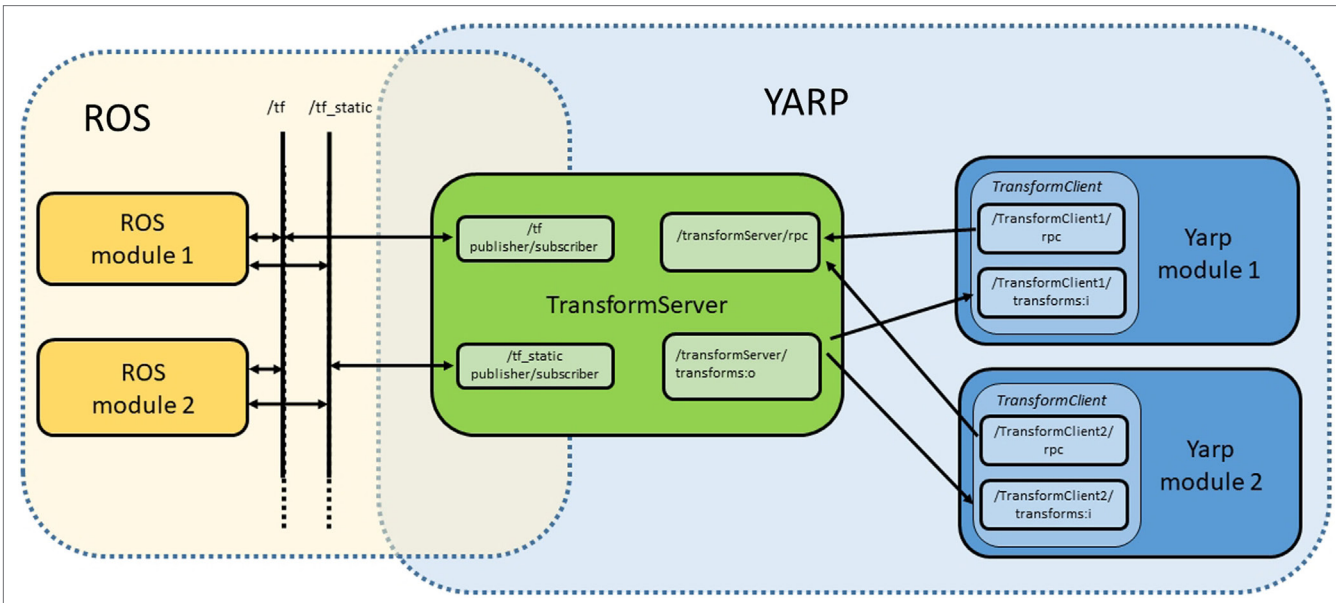


FIGURE 1 | Typical scenario in which multiple YARP modules, each of them instantiating its own `yarp::dev::transfomClient`, communicate with a single `yarp::dev::transformServer`. The latter is responsible for synchronizing YARP transforms with ROS data, publishing and subscribing to /*tf* and /*tf_static* topics.

inside a user module (**Figure 1**). TransformClient is an entity which implements the `yarp::dev::IFrameTransform` interface (see Sections II and III in Supplementary Material). Available methods allow the user to query the server about the registered YARP and ROS transforms, to perform kinematic computations, and to register on the server new transforms computed by YARP modules.

3. YARP CLASSES AND INTERFACES FOR NAVIGATION

This section presents the new YARP classes and interfaces specifically designed for managing maps and controlling a robot during a navigation task. Detailed description of available methods and usage examples are shown in Supplementary Material.

3.1. MapGrid2D

The class `yarp::dev::MapGrid2D` is the main YARP class used to store map data. Similar to ROS occupancy grid message (*nav_msgs/OccupancyGrid.msg*), data are organized in square cells of fixed size (e.g., 0.05 m × 0.05 m), each of them storing the probability of being occupied by a fixed obstacle (e.g., a wall). This information is typically used to localize the robot in an environment previously mapped by a SLAM algorithm. In addition to this property, map cells are also provided with an additional flag (Section IV in Supplementary Material), which can be used to control the robot behavior. For example, a user can choose to set keep-out areas, which should be avoided by the robot when it computes its path, or critical areas in which the robot should stop when an obstacle is encountered (instead of finding an alternate path). Finally, MapGrid2D is equipped with methods to save/load maps both in YARP and in a ROS compatible format.

3.2. Map2DLocation

A `yarp::dev::Map2DLocation` is a support class used to store user location information. A location is composed of the location name, the map name to which the location refers to, and the (x,y,θ) coordinates w.r.t. the map origin. Locations are typically stored together with maps in a *map2DServer* (see Section 4.1) so that a user can invoke the navigation APIs using the location name instead of the corresponding coordinates. Locations are also used by *map2DServer* to define interconnection points between multiple YARP maps.

3.3. IMap2D

`yarp::dev::IMap2D` is a pure virtual interface dedicated to the management of `MapGrid2D` and `Map2DLocation` entities. A *Map2DServer* (Section 4.1) implements methods of this interface to satisfy the requests from a *Map2DClient*. The complete listing of the methods belonging to `yarp::dev::IMap2D` as well as an application example is shown in Sections V and VI in Supplementary Material.

3.4. INavigation2D

`yarp::dev::INavigation2D` is a pure virtual interface shared between all client/server modules, which performs navigation tasks. The most classical usage in a user application requires the instantiation of a `yarp::dev::INavigation2DClient` to send navigation commands to the robot (e.g., "go to the entrance room"). On the other side, the server counterpart, which can be any module implementing the same `yarp::dev::INavigation2D` interface (e.g., robotPathPlanner, see Section 4.6), receives the goal command and computes the path required by the robot to reach the goal.

INavigation2D contains methods to start, pause, and resume navigation tasks, both in absolute (with respect to the map reference frame) or in relative coordinates (with respect to the robot reference frame) (Section VII Supplementary Material). Additionally, it allows the user to assign names to the current robot position and to important locations on the map. These names might be used instead of absolute coordinates when commanding a goal to the robot. Finally, the user can query the current status of the navigation task. The enum returned by the method `INavigation2D::getNavigationStatus()` can be used by the client application to know when the goal has been reached or if a problem occurred (Section VIII in Supplementary Material).

4. YARP MODULES AND TOOLS FOR NAVIGATION

This section describes the YARP modules and tools which constitute the core of the YARP navigation suite. They are provided inside `robotology/yarp` and `robotology/navigation` github repositories. A comparison between these YARP tools and similar ones provided by ROS is reported in **Table 1**.

4.1. Map2DServer

Map2DServer implements the methods of the YARP interface `yarp::dev::IMap2D`, and it allows a client application (such as the navigation module) to store and retrieve maps (`yarp::dev::MapGrid2D`) from memory. It can be initialized at startup by a map collection file which contains an index of all map files to be used in the session. It must be noticed that this module only behaves as a storage, and it contains neither information about the current robot position nor the name of the map in which the robot finds itself. These tasks are performed by other modules (e.g., *localizationServer*, Section 4.4) which interact with the map2Dserver when they need to obtain map data. Finally, this module implements some methods of the `yarp::dev::INavigation2D` interface, allowing to store/retrieve user notable locations (`yarp::dev::Map2DLocation`) on a map.

4.2. BaseControl

BaseControl is the core YARP module used to control a mobile robot. It receives cartesian velocity commands ($\dot{x}$, $\dot{y}$, $\dot{\theta}$) either from a navigation module or from a joystick device, and it computes the corresponding actuators actions required to achieve them. *BaseControl* is also responsible for computing robot odometry, i.e., estimating the robot position in the world using measured motions of robot actuators. Computed data are published on a YARP port both as a vector (x,y,θ) and, via

TABLE 1 | Similarities and correspondences between YARP and ROS modules with similar functionalities.

YARP	ROS	Notes
Map2DServer	map_server	map_server offers a single map via ROS latched topic/map. Map2DServer acts a storage for multiple maps and user-defined locations
BaseControl	–	In ROS, there is no equivalent module. Each kind of robot exposes its own specific control interface
Mapper2D	gmapping	gmapping performs loop closure detection and simultaneous localization and mapping. Mapper2D allows to set not only the occupancy value of the cell but also the YARP map flag
LocalizationServer	–	LocalizationServer does not have a direct correspondence in ROS. It acts as a bridge for a ROS localization module like Adaptive Montecarlo Localization (AMCL) adding the support for YARP map collections (not directly supported in ROS)
–	AMCL	YARP navigation suite currently does not provide any localization system for mobile robots. A YARP user may use a ROS module such as AMCL to estimate the robot position against a known map or use its own localization system
RobotGoto	move_base-base_local_planner	The two modules have similar functionalities although ROS base_local_planner supports multiple algorithms (e.g., Trajectory Rollout and Dynamic Window Approach) while RobotGoto artificial potential fields approach is more tailored to work together with YARP RobotPathPlanner
RobotPathPlanner	move_base-global_planner	The two modules have similar functionalities and use comparable algorithms

transformClient, as a transform between the origin of the odometry system (*/odom*) and the robot (*/mobile_base*). This allows a ROS module to interface with the robot by subscribing to the /*tf* topic.

4.3. Mapper2D

Mapper2D is a simple YARP module which registers laser scans to build an occupancy-based map. The module is not equipped with a loop closure detector, nor with an internal localization algorithm; thus, it is not suitable to perform stand-alone SLAM tasks. Instead, it is designed to receive accurate localization data from an external source (e.g., a Google Tango device) either via YARP port or via *transformClient*.

4.4. LocalizationServer

LocalizationServer is an auxiliary tool which acts as the server side of a *Navigation2DClient* for the `INavigation2D::getCurrentPosition()` and `INavigation2D::setInitialPose()` methods. *Robotology/navigation* repository does not provide a default localization system for a mobile robot. A YARP user may thus choose to employ a YARP-based localization system (such as *Robust-View-Graph-SLAM*), or a ROS-based one (e.g., *AMCL*, *RTAB-Map*, *Tango-ROS-Streamer*). In this latter case, *LocalizationServer* acts as a bridge between the ROS world (which is single map) and the YARP world (which is multi-map). When the user sets an initial position to initialize the localization algorithm, it specifies a `yarp::dev::Map2DLocation` which is translated to a string (the map name, handled by the *Map2DServer*) and a (x,y,θ) vector. This latter is sent with a *geometry_msgs/PoseWithCovarianceStamped* message to the ROS localization module as the estimated robot pose with respect to the origin frame of the current map.

4.5. RobotGoto

This module computes the cartesian velocities ($\dot{x}$, $\dot{y}$, $\dot{\theta}$) of the mobile base required to reach the commanded goal, given the current robot position (provided through a *transformClient*) and a set of parameters that controls the trajectory generation (e.g., differential drive or holonomic robot kinematics, heading and goal tolerance, etc.).

RobotGoto does not use any map information, except for the local occupancy grid which is continuously updated according to sensor data. An artificial potential field algorithm is employed to allow the robot to avoid obstacles obstructing the path to the goal. Depending on the configuration parameters, if a deadlock is detected, navigation may be paused (waiting a human to remove the obstacle) or aborted. In this latter case, the high-level path planner is notified by a specific `yarp::dev::INavigation-2D::NavigationStatusEnum`, as shown in Section VIII and Figure S1 in Supplementary Material.

4.6. RobotPathPlanner

This module is responsible for generating the navigation waypoints to be pursued by a local navigation module (e.g., *robotGoto*). By implementing the `INavigation2D` interface, *robotPathPlanner* acts as the server counterpart of a *Navigation2DClient* instantiated by a user module. For example, when the user calls the `INavigation2D::gotoAbsolutePosition()` method to command the robot to reach a new goal, *robotPathPlanner* becomes in charge of performing the navigation task, notifying the user about its current status (e.g., in progress, goal reached, etc.).

The algorithm acts as follows. *RobotPathPlanner* retrieves from a *Map2DServer* instance the current map of the area. A valid path from the current robot location to the goal is computed using A* algorithm. If the path does not exists, navigation is aborted. Otherwise the path, initially defined as a vector of map cells, is transformed into a sequence of navigation waypoints. To be accepted, these waypoints must satisfy some user-defined parameters (e.g., minimum distance between the cells etc.). Waypoints are then put in a queue and sent one by one to a local navigation algorithms (such as *robotGoto*) which will pursue them.

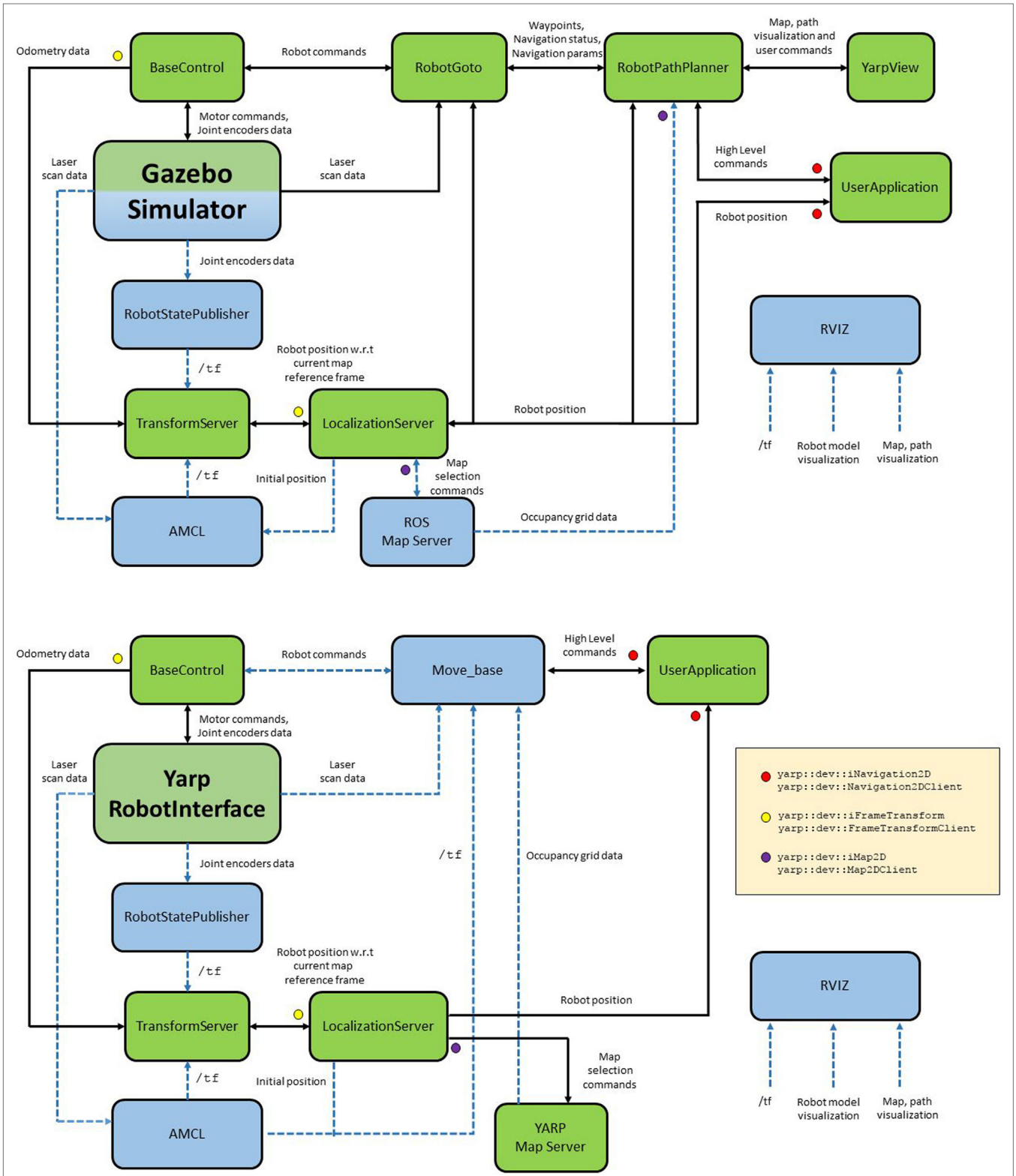


FIGURE 2 | Two realistic application scenarios, in which different combinations of YARP (green) and ROS (blue) modules are employed. Solid lines represent YARP port connections. Dashed lines represent ROS topic connections (Rviz connections are omitted for diagram clearness). Colored markers indicate the YARP interfaces employed to interconnect the various client/server modules. Gazebo simulator is represented as a hybrid YARP/ROS module because its modular design allows to execute plugins belonging to both frameworks (Mingo Hoffman et al., 2014).

RobotPathPlanner is also responsible for processing the YARP flags assigned to particular areas of the map. These flags may belong to two different categories. Those which alter the navigation trajectory (such as keep-out areas) are directly processed by the module during the trajectory generation phase. Instead, flags which alter the robot behavior (e.g., areas in which the robot must proceed at a different speed or interrupt the navigation if an obstacle is detected on the path) are not directly processed. Indeed, since they affect the behavior of the local navigation task, a proper set of commands is generated and sent to *RobotGoto* to modify the default navigation parameters.

Finally, *RobotPathPlanner* is able to show the computed robot trajectory by means of the standard YARP graphical visualization tool *yarpview* and, additionally, to receive navigation commands from it (dragging an arrow on the map will be interpreted as goal command).

5. NAVIGATION INTEGRATION AND EXAMPLES

YARP and ROS may inter-operate in several ways to attain a navigation task. Different possibilities range from using a full YARP-based framework to using the complete ROS navigation stack. In between there exist a number of possible combinations: as shown in previous sections, most of the YARP components can be replaced by a ROS equivalent or vice versa, depending on the user needs and preferences.

Figure 2 shows two illustrative scenarios. The first example refers to a simulated wheeled robot in *Gazebo*, a generic, multi-robot, physics simulator. The navigation task is carried out by robotGoto/robotPathPlanner modules. Since ROS *map_server* is used, robotPathPlanner employs only the occupancy grid information and no YARP map flags are available.

The second example refers to a real wheeled robot (i.e. R1 (Parmiggiani et al., 2017)) controlled by *yarpRobotInterface*, the core YARP application which manages the low-level hardware control. In this case, navigation task is carried out by ROS navigation stack encapsulated inside *move_base* node.

It must be noticed that, in both scenarios, the final end-user is a YARP application which instantiate a `yarp::dev::INavigation2DClient`. Section IX in Supplementary Material shows a simple application which controls the robot to reach a location stored into the map server, unaware of which framework and control modules are employed underneath. The included sequence diagram (Figure S2 in Section X in Supplementary Material) shows the timing and the messages exchanged between the clients opened by the example and the connected external modules (i.e., LocalizationServer, Map2DServer, robotPathPlanner).

Finally, a set of examples of increasing complexity is included in the github repository (Figure S3 in Section X in Supplementary Material), as well as some skeleton applications which the user can exploit to develop its own navigation modules.

6. CONCLUSION AND FUTURE WORK

In this paper, we showed latest developments to improve YARP interoperability with ROS. These improvements allow a robotics developer to use YARP middleware without giving up popular and convenient ROS features, such as the /tf package. By introducing a brand new set of standard interfaces, such as `yarp::dev::IMap2D` and `yarp::dev::INavigation2D`, YARP is now capable of performing a 2D navigation task, natively or interacting with ROS.

Future work will be aimed to further improve YARP-ROS integration. YARP transformServer is currently unable to interpolate/extrapolate frames over time, an advanced feature that is instead available in the ROS /tf package, which allows users to ask for the pose of a frame at a specific time instant, in the past or even in the future. Additionally, YARP is currently unable to manage octomaps or other 3D data types. Their introduction is thus a required step to allow foot planning of a bipedal robot on a highly structured terrain.

AUTHOR CONTRIBUTIONS

MR: development of YARP interfaces and classes for navigation; development of the navigation modules belonging to https://github.com/robotology/navigation repository; and experiments with real and simulated robots. AR: development of transformServer/transformClient, development of automatic tests for frameTransform and navigation interfaces; and experiments with real and simulated robots. LN: development of YARP framework and scientific supervision.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at http://www.frontiersin.org/articles/10.3389/frobt.2018.00005/full#supplementary-material.

REFERENCES

Cousins, S. (2010). Ros on the pr2. *IEEE Robot. Autom. Mag.* 17, 23–25. doi:10.1109/MRA.2010.938502

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Software Eng. Robot.* 5, 42–49. Available at: https://joser.unibg.it/index.php?journal=joser&page=article&op=view&path%5B%5D=69

Hu, Y., Eljaik, J., Stein, K., Nori, F., and Mombaur, K. (2016). "Walking of the iCub humanoid robot in different scenarios: implementation and performance analysis," in *IEEE-RAS 16th International Conference on Humanoid Robots (Humanoids)* (Cancun, Mexico), 690–696.

Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., and Konolige, K. (2010). "The office marathon: robust navigation in an indoor office environment," in *International Conference on Robotics and Automation*, Anchorage, AK.

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi:10.1016/j.neunet.2010.08.010

Mingo Hoffman, E., Traversaro, S., Rocchi, A., Ferrati, M., Settimi, A., Romano, F., et al. (2014). *Yarp Based Plugins for Gazebo Simulator*. Springer International Publishing, 333–346. Available at: https://link.springer.com/chapter/10.1007/978-3-319-13823-7_29

Natale, L., Paikan, A., Randazzo, M., and Domenichelli, D. E. (2016). The iCub software architecture: evolution and lessons learned. *Front. Robot. AI* 3:24. doi:10.3389/frobt.2016.00024

Nava, G., Romano, F., Nori, F., and Pucci, D. (2016). "Stability analysis and design of momentum-based controllers for humanoid robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Daejeon, South Korea), 680–687.

Parmiggiani, A., Fiorio, L., Scalzo, A., Sureshbabu, A. V., Randazzo, M., Maggiali, M., et al. (2017). "The design and validation of the R1 personal humanoid," in *International Conference on Intelligent Robots (IROS)*, Vancouver, BC.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, Kobe, Japan.

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

CODE
published: 12 March 2018
doi: 10.3389/frobt.2018.00022

iCub-HRI: A Software Framework for Complex Human–Robot Interaction Scenarios on the iCub Humanoid Robot

Tobias Fischer[1], Jordi-Ysard Puigbò[2,3], Daniel Camilleri[4], Phuong D. H. Nguyen[5], Clément Moulin-Frier[2], Stéphane Lallée[2], Giorgio Metta[5], Tony J. Prescott[4], Yiannis Demiris[1] and Paul F. M. J. Verschure[2,3,6]*

[1] *Personal Robotics Laboratory, Electrical and Electronic Engineering Department, Imperial College, London, United Kingdom,* [2] *Synthetic Perceptive Emotive and Cognitive Systems Group (SPECS), Universitat Pompeu Fabra, Barcelona, Spain,* [3] *Institute for Bioengineering of Catalonia (IBEC), The Barcelona Institute of Science and Technology, Barcelona, Spain,* [4] *Department of Computer Science, University of Sheffield, Sheffield, United Kingdom,* [5] *iCub Facility, Istituto Italiano di Tecnologia, Genova, Italy,* [6] *ICREA-Institució Catalana de Recerca i Estudis Avançats, Barcelona, Spain*

OPEN ACCESS

Edited by:
Lorenzo Jamone, Queen Mary University of London, United Kingdom

Reviewed by:
Amit Kumar Pandey, SoftBank Robotics, France
Torbjorn Semb Dahl, Plymouth University, United Kingdom

****Correspondence:***
Tobias Fischer
t.fischer@imperial.ac.uk

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *31 May 2017*
Accepted: *21 February 2018*
Published: *12 March 2018*

Citation:
Fischer T, Puigbò J-Y, Camilleri D, Nguyen PDH, Moulin-Frier C, Lallée S, Metta G, Prescott TJ, Demiris Y and Verschure PFMJ (2018) iCub-HRI: A Software Framework for Complex Human–Robot Interaction Scenarios on the iCub Humanoid Robot. Front. Robot. AI 5:22. doi: 10.3389/frobt.2018.00022

Generating complex, human-like behavior in a humanoid robot like the iCub requires the integration of a wide range of open source components and a scalable cognitive architecture. Hence, we present the iCub-HRI library which provides convenience wrappers for components related to perception (object recognition, agent tracking, speech recognition, and touch detection), object manipulation (basic and complex motor actions), and social interaction (speech synthesis and joint attention) exposed as a C++ library with bindings for Java (allowing to use iCub-HRI within Matlab) and Python. In addition to previously integrated components, the library allows for simple extension to new components and rapid prototyping by adapting to changes in interfaces between components. We also provide a set of modules which make use of the library, such as a high-level knowledge acquisition module and an action recognition module. The proposed architecture has been successfully employed for a complex human–robot interaction scenario involving the acquisition of language capabilities, execution of goal-oriented behavior and expression of a verbal narrative of the robot's experience in the world. Accompanying this paper is a tutorial which allows a subset of this interaction to be reproduced. The architecture is aimed at researchers familiarizing themselves with the iCub ecosystem, as well as expert users, and we expect the library to be widely used in the iCub community.

Keywords: robotics, iCub humanoid, human–robot interaction, YARP, software architecture, code:C++, code:Python, code:Java

1. INTRODUCTION AND BACKGROUND

The iCub is an advanced humanoid robot, which is equipped with multiple sensors: encoders in all its 53 joints, force/torque sensors, tactile sensors integrated in the artificial skin, and eye cameras (Metta et al., 2010). They allow for a coherent understanding of body configuration, motor capabilities, and the environment as well as an ability to show facial expressions, which makes it an ideal platform for studies of human–robot interaction and cognition.

The research community around the iCub humanoid robot is very active, with a large number of papers published every year. The source code leading to these publications is often made available to the public, which allows for the replication of the results and use of the code as a starting platform to tackle new research questions. However, despite YARP (Fitzpatrick et al., 2006) being typically used as the underlying middleware in these works, it remains challenging to combine these efforts in a coherent manner.

Here, we present iCub-HRI, which integrates several components for perception, object manipulation, and social interaction using two parts: (1) The iCub-HRI library, which facilitates the use of the aforementioned components by providing easy to use classes with suitable default parameters (called *Subsystems*) and a shared knowledge database as means to represent knowledge which is employed across all components. (2) Modules which supply the shared knowledge database with input, as well as some modules tailored for human–robot interaction scenarios.

1.1. Background and Related Works

iCub-HRI has its origins in the Experimental Functional Android Assistant (EFAA) project,[1] where most of the library was developed and employed in several works (e.g., Lallée et al. (2013, 2015), Petit et al. (2013), and Lallée and Verschure (2015)). EFAA targeted the development of a cognitive architecture to realize effective and psychologically plausible human–robot dyadic interaction. The code was then extended and matured further in the What You Say Is What You Did (WYSIWYD) project,[2] and more papers based on iCub-HRI were published (e.g., Fischer and Demiris (2016), Martinez-Hernandez et al. (2016), Petit et al. (2016), Puigbò et al. (2016), and Moulin-Frier et al. (2017)). WYSIWYD aimed at realizing robot human level language capabilities by augmenting this cognitive architecture with mechanisms for language acquisition, composition, and expression. The cognitive architecture in both projects is based on and elaborates the Distributed Adaptive Control theory of mind and brain (DAC, see for reviews Verschure (2012, 2016) and Section 4.3).

While reviewing robotics middlewares is out of the scope for this paper (we refer to Elkady and Sobh (2012) for an overview), we briefly introduce several other proposals detailing software frameworks for various robotics platforms. Natale et al. (2016) summarize recent developments of the iCub's software architecture, including the compatibility with the Robot Operating System (ROS) and the introduction of a new testing framework. They find that ROS is being adopted rapidly by more and more robot developers, and indeed, there are several papers introducing human–robot interaction-related frameworks based on ROS. For example, Jang et al. (2015) propose a ROS-based framework where modules concerned with low-level control and service logic are separated from modules concerned with social behaviors. Lane et al. (2012) present a bundle of ROS modules which allows the extension of existing projects for speech recognition, natural language understanding, and basic gesture recognition as well as gaze tracking. A toolkit which allows the evaluation of human–robot interactions in virtual reality environments and subsequent deployment on a real robot was presented by Krupke et al. (2017). The robot behavior toolkit (Huang and Mutlu, 2012) includes a ROS module which is based on the findings within the social sciences. While the authors conducted a large-scale study with humans, the evaluation was based on simulated sensor data. Finally, Sarabia et al. (2011) present a framework allowing to perceive the actions and intentions of humans, and show its application in a social context where a robot imitates the dance movements of a human.

1.2. Design Principles

Here, we devise a set of guidelines and design principles which were adopted when coding the framework.

- *Adaptability and ease of use:* the framework should be easy to adapt by the community. Individual parts of the framework should only depend on other parts if necessary, and substituting components should be easy. Furthermore, all libraries and modules should be properly documented.
- *Provision of overall framework:* related to the previous goal, our aim is to provide an overall framework which can work "out of the box." Hence, our framework contains modules related to perception, action execution, and social interaction.
- *Extendibility:* it should be easy to extend the framework with new modules. Rather than tailoring existing modules to work with the iCub-HRI framework, it should be possible to write wrapper code for the integration.
- *Shared, centralized knowledge representation:* each module should have access to the same knowledge database, and the contained knowledge should follow a standardized format. Within iCub-HRI, we call this knowledge database the *working memory*, and the contents are *Entities* or derivatives thereof. The working memory is the default means of communicating among modules.
- *Open software:* the code is released open source and made publicly available. All dependencies must be available as open source software too.

2. THE iCub-HRI LIBRARY

Due to the support of distributed computation within the YARP middleware, there are typically many modules running simultaneously when conducting research on the iCub. Typically, data are exchanged using YARP's *Bottle* container, which can encapsulate data of arbitrary length and varying type. While this allows a high degree of flexibility, these containers are error prone due to the requirement of parsing the messages dynamically. This makes verification of compatibility and versioning when used across a large number of modules hard (Natale et al., 2016). Thus, within the iCub-HRI library, we introduce fixed data representations for knowledge (fully compatible with the *Bottle* container), similar to those used in ROS messages (Quigley et al., 2009) and the Interface Description Language

[1] http://efaa.upf.edu/.
[2] http://wysiwyd.upf.edu/.

(IDL) in YARP (Fitzpatrick et al., 2014). Contrary to ROS messages and IDLs, the same representations are used across all components of the iCub-HRI library. The representations and their exchange which is orchestrated by a working memory are detailed in Section 2.1.

The communication protocol with external modules is described within *Subsystems*. Each subsystem connects to a host (i.e., external module) and abstracts away the communication internals, as described in Section 2.2. Finally, the *icubClient* class is designed with additional convenience for end users in mind such that all subsystems and other higher level methods are available from within a single class.

2.1. Knowledge Representation and Exchange

The basic representation type is an *Entity*, which is specified by an *ID* and an associated *name*. The *ID* is used when storing and retrieving the entity from the working memory. Several entities can be linked together by the means of a *Relation*, for example, the human "Paul" (subject) "holds" (verb) "duck" (object). For further details on relations, we refer to Lallée and Verschure (2015).

Other knowledge representations inherit the basic properties and methods of *Entity* and extend them further. The *Object* class has additional properties representing the pose, size, presence, and saliency of an object (see Section 3.1 for details how these properties are acquired). The *Agent* class represents a human partner, which additionally to all properties of an *Object* also stores the positions of all body parts and a list of beliefs. Another commonly used representation is that of a *Bodypart*, which represents a part of the robot's body. A *Bodypart* also inherits all attributes of an *Object*, and additionally contains the related joint number, tactile patch identifier, and corresponding body part of the human. Zambelli et al. (2016) have used these representations to anchor self-learned representations to those of a human interacting with the robot.

These representations must be shared across different modules (for example, between perceptual modules and the more abstract reactive layer as described later in this section), and we designed the *OPCClient* class to automate the exchange of representations with the working memory of the iCub ecosystem (Objects Properties Collector; OPC) (Lallée and Verschure, 2015). The OPC is an ontology-based knowledge representation system which is grounded on the need of humans and other social animals to interact in a physical, multi-agent world (see Lallée and Verschure (2015)). In this direction, the role of such knowledge representation should be to structure and distribute information to different modules in an asynchronous (on-demand) and centralized way. The design is inspired by the *repository pattern* known from software engineering (Evans, 2004), and its usage is very similar to the centralised version control software Apache Subversion (known as SVN).[3] For storage and retrieval, the *OPCClient* provides methods such as "checkout" to poll representations from the shared memory, "update" to update existing representations, and "commit" to overwrite representations in the memory with the local version of the module. Altogether, this implementation provides a shared, centralized knowledge representation (following our design principle outlined in Section 1.2), enabling asynchronous access to the information in a way similar to how brains work.

2.2. Subsystems

A *Subsystem* provides a wrapper between the representations used by external components and the ones used within iCub-HRI, which compares to the façade software engineering pattern (Gamma et al., 1994). This has several advantages, including that the complexity of remote procedure calls is hidden from the user and that formerly "incompatible" components can now be used within the same project. Within this paper, we provide a brief list of the most commonly used interfaces of these subsystems, and we provide a complete list in the documentation on GitHub.[4]

This is especially evident in the subsystems for the *Actions Rendering Engine* (ARE; follow up work on Pattacini et al. (2010))[5] and *KARMA* (Tikhanoff et al., 2015)[6] object manipulation libraries, which are typically used by the iCub community to issue high-level motor commands. If directly called, they require the provision of complex parameters. Contrary, using iCub-HRI, one simply specifies the desired action and the name of the object to be manipulated, as further demonstrated in Section 4.1.

Other important subsystems are those for speech recognition and synthesis. Both are convenience wrappers for the functionality offered in the "speech" repository of the iCub ecosystem. The speech synthesizer allows for speech production from text using a single command "say()," with the only parameter being the sentence to be spoken, while being agnostic to the underlying synthesizer (Acapela,[7] eSpeak,[8] Festival,[9] and SVOX Pico[10] are supported). The speech recognizer relies on the Microsoft Speech API,[11] which allows recognition and extraction of words from spoken utterance given a grammar file (using the command "recogFromGrammarLoop()").

The functionality of the different subsystems is aggregated in the *icubClient* class, which allows using the different subsystems from within a single class instance. A configuration file is used to specify which subsystems a module requires, such that no unnecessary resources are bound. Adding new subsystems is straightforward and we provide a tutorial to do so.[12]

[3] https://subversion.apache.org/.

[4] https://robotology.github.io/icub-hri/ → iCub-HRI libraries → Subsystems.

[5] The following interfaces are provided by the ARE subsystem: (1) "home()" to put the robot or a specified part in the home position, (2) "take()" to reach and grasp an object, (3) "push()" to laterally push an object, (4) "point()" to an object, (5) "expect()" to extend the hand and wait for an object, (6) "drop()" an object which is currently held, (7) "wave()" the robot's hands, (8) "look()" at an object, and (9) "track()" a moving object.

[6] The following interfaces are provided by the KARMA subsystem: (1) "pushKarmaLeft()" and "pushKarmaRight()" to push an object to the left/right side with a specified target position, (2) "pushKarmaFront()" to push an object forwards, and (3) "pullKarmaBack()" to bring an object closer to the robot.

[7] http://www.acapela-group.com.

[8] http://espeak.sourceforge.net/.

[9] http://www.cstr.ed.ac.uk/projects/festival/.

[10] https://github.com/robotology/speech/tree/master/svox-speech.

[11] https://msdn.microsoft.com/en-us/library/ee125663(v=vs.85).aspx.

[12] https://robotology.github.io/icub-hri/ → Tutorials → Create a new Subsystem.

3. iCub-HRI MODULES

The modules accompanying the iCub-HRI library can be grouped into four main areas: 1. perception, 2. action, 3. social interaction, and 4. miscellaneous tools. All modules have access to the knowledge introduced in the previous section (as they use the iCub-HRI library) and none of them is required to run; i.e., one can choose which subset of modules to run for each experiment, if any.

3.1. Perception Modules

3.1.1. Agent Detector

The *agentDetector* module is responsible for detecting and tracking a human partner using a RGB-D camera mounted behind the robot. It converts the joint positions detected by the RGB-D camera in the reference frame of the iCub and continuously updates the joint positions of the human partner in the working memory.

3.1.2. Default Speech Recognition

The *Ears* module allows for recognition of speech utterances from the human when no other module is trying to recognize speech. It takes the role of a central component to redirect the command extracted from the recognized sentence to the appropriate module, while still allowing other modules to access the speech recognition subsystem directly if needed.

3.1.3. Object Recognition

The object recognition module within iCub-HRI is based on the *interactive object learning* (*IOL*) pipeline (Pasquale et al., 2015). Given the two input images of the iCub's eyes, the scene is first segmented into the background and the different objects. Each object is then classified and stereo vision (Fanello et al., 2014) is used to localize the objects. We rely on superquadric models to estimate the size and pose of objects (Vezzani et al., 2017), and we use the OpenCV object tracker (Kalal et al., 2012) to track them even if they are manipulated by the human.

3.1.4. Saliency

The module *PASAR* (Mathews et al., 2012) detects the appearance and disappearance of objects, and the saliency of an object is increased proportionally to its acceleration. This also allows simple detection of pointing actions by measuring the proximity of the human's hand with each of the objects and increasing the saliency with inverse proportion to the distance.

3.1.5. Face and Action Recognition

To recognize faces and actions performed on objects, we use the *Synthetic Sensory Memory* module (Martinez-Hernandez et al., 2016). It uses Gaussian Process Latent Variable Models (Damianou et al., 2011) to train classifiers for faces and actions, which can then be loaded during interaction to perform real-time classification.

3.2. Action Modules

3.2.1. Face Tracking

The face tracking module detects the face of a human based on Haar cascades implemented in OpenCV (Viola and Jones, 2001) and uses the velocity control of the iCub to follow the face. This module can be used in human–robot interaction scenarios for increased vividness of the robot.

3.2.2. Babbling

The *babbling* module allows the issue of pseudo random (sinusoids) commands to the iCub (either individual or several joints). It has been used to learn forward and inverse models for the iCub (Zambelli and Demiris, 2017), as well as to learn correspondences between the robot's body parts and that of the human (Zambelli et al., 2016). Within the scope of this paper, it is mainly used for body part learning, as described in Section 4.2.

3.3. Social Interaction Modules

3.3.1. Proactive Tagging

The proactive tagging module can be used to acquire the names of objects (robot), body parts, and human partners. As this module plays a central role in the knowledge acquisition tutorial, it is further detailed in the corresponding Section 4.2.

3.3.2. Reactive Layer

The reactive layer implements drive reduction mechanisms for self-regulating the robot's behavior. A drive is defined as a control loop that triggers appropriate behaviors whenever an associated internal state variable goes out of its homeostatic range. These drives present a way to self-regulate value in a dynamic and autonomous way (Sanchez-Fibla et al., 2010). This has been shown to positively influence the acceptance of the human-robot interaction by naive users (Vouloutsi et al., 2014; Lallée and Verschure, 2015).

In the social robotic context, we provide two examples of drives that allow the robot to balance knowledge acquisition and expression in an autonomous way. The *drive for knowledge acquisition* maintains a curiosity-driven exploration of the environment by proactively requesting information from a human about the present entities (e.g., their name). The *drive for knowledge expression* regulates how the iCub expresses the acquired knowledge through synchronized speech, pointing actions and gaze. It informs the human about the robot's current state of knowledge and thus maintains the interaction.

3.4. Tools

Several tools provide preprocessing functionalities for the other modules or interact with other modules of the iCub ecosystem so they can be easily used within iCub-HRI. The *guiUpdater* translates the representations of iCub-HRI to those used within the *iCubGui*. More specifically, it allows the display of location for objects and agents stored within the working memory along with certain properties, such as their color and name. The *opcPopulator* can be used to spawn new entities in simulation and control their parameters. This allows testing new functionalities in a controlled environment, without the noise encountered when using the real robot. We further provide a *touchDetector* that connects to the iCub's artificial skin, and clusters taxels belonging to the same body part. Finally, the *referenceFrameHandler* provides functionalities similar to that of the transform

library (TF; Foote, 2013), i.e., transforming a pose from one frame (e.g., that of the RGB-D camera) to another (e.g., that of the iCub root).

4. USING iCub-HRI

There is a variety of use cases for iCub-HRI. We first show the ease of use of iCub-HRI in a representative example related to the object manipulation subsystem. We then introduce a tutorial which demonstrates the interplay of various components in the context of human–robot interaction. Subsequently, we briefly describe how an extended version of this tutorial has been used to tackle the symbol grounding problem in the DAC-h3 framework (Moulin-Frier et al., 2017). This is followed by a description of the implications of this library for technical and non-technical users alike. Finally, we discuss the platform independence and dependencies of iCub-HRI and provide links to the documentation and repository.

4.1. Example Usage of the Object Manipulation Subsystems

The GitHub repository contains a range of examples, including examples of using the *KARMA* and *ARE* subsystems to manipulate objects, i.e., grasping, pushing, or pulling them, in C++, Python, and Matlab. Some examples use *yarp::sig::Vector* instances to specify the target location (important for users looking to employ iCub-HRI as a lightweight library), while others rely on the *Object* class introduced earlier (providing a seamless integration with the contained object recognition module). Listing 1 shows an example which uses the iCub-HRI library to pull an object using the *KARMA Subsystem*, while Listing 2 contains code directly communicating with *KARMA*, which is much less intuitive and likely distracts from the actually desired code related to the human–robot interaction.

LISTING 1 | Pushing an object using iCub-HRI is straightforward and requires the provision of just two parameters: the object to be pushed and the desired target position.

```
#include <yarp/os/all.h>
#include <icubhri/clients/icubClient.h>

int main() {
    yarp::os::Network yarp;

    icubhri::ICubClient iCub("KARMA_Simple");
    if(!iCub.connect()) {return -1;}//connect to
    subsystems

    std::string objectName = "octopus";//as recognized by
    object recognition
    double targetPositionX = -0.45;

    bool ok = iCub.pushKarmaFront(objectName,
    targetPositionX);
    yInfo() << (ok ? "Success": "Failed");

    return 0;
}
```

LISTING 2 | Pushing an object communicating directly with KARMA. Besides being less readable, this code is also more error prone as the Bottle's components need to be provided with the right type and in the right order. Furthermore, many more parameters are involved.

```
#include <yarp/os/all.h>
#include <yarp/sig/all.h>

yarp::sig::Vector getPos(std::string name) {
    //communicate with object recognition module to obtain
    object position
    //this is not shown for brevity
}

int main() {
    yarp::os::Network yarp;

    yarp::os::RpcClient toKarma; toKarma.open("/example/
    toKarma");
    yarp::os::Network::connect(toKarma.getName(), "/
    karmaMotor/rpc");

    yarp::sig::Vector pos = getPos("octopus");
    double targetPositionX = -0.45;
    double radius = fabs(pos[0] - targetPositionX);

    yarp::os::Bottle cmd, reply;
    cmd.addString("push");
    cmd.addDouble(pos[0]); cmd.addDouble(pos[1]); cmd.
    addDouble(pos[2]);
    cmd.addDouble(-90);//angle theta
    cmd.addDouble(radius);//distance to be pushed
    toKarma.write(cmd, reply);
    bool ok = (reply.get(0).
    asVocab() == yarp:os:Vocab:encode("ack"));
    yInfo() << (ok ? "Success": "Failed");

    return 0;
}
```

4.2. Knowledge Acquisition Tutorial

The robot can acquire knowledge in two different ways: proactively, where a decaying drive to acquire knowledge triggers the behavior to obtain the name of an object or body part, or reactively, where the knowledge acquisition follows a human command. The demo for this paper is centered around the proactive tagging module, which makes use of several subsystems and connects directly to several other modules. For example, it uses the speech recognition subsystem to acquire the names of entities (objects in the vicinity, partners, and body parts), the speech synthesis subsystem to enable the robot to verbally express itself (in order to ask for object names), and the *ARE* subsystem to point at objects and make them salient. Furthermore, it makes use of the functionalities provided by a number of other modules presented within the previous section, including *PASAR* to detect which object the partner is pointing to, the face recognition module to recognize the partner, and the *touchDetector* to identify which skin patch was being touched by the human. An overview of the interaction between the modules is shown in **Figure 1**. All further details, including the necessary set-up, configuration files, modules to run, and supported interactions, are described in the dedicated tutorial. We provide a set of videos of this experiment which

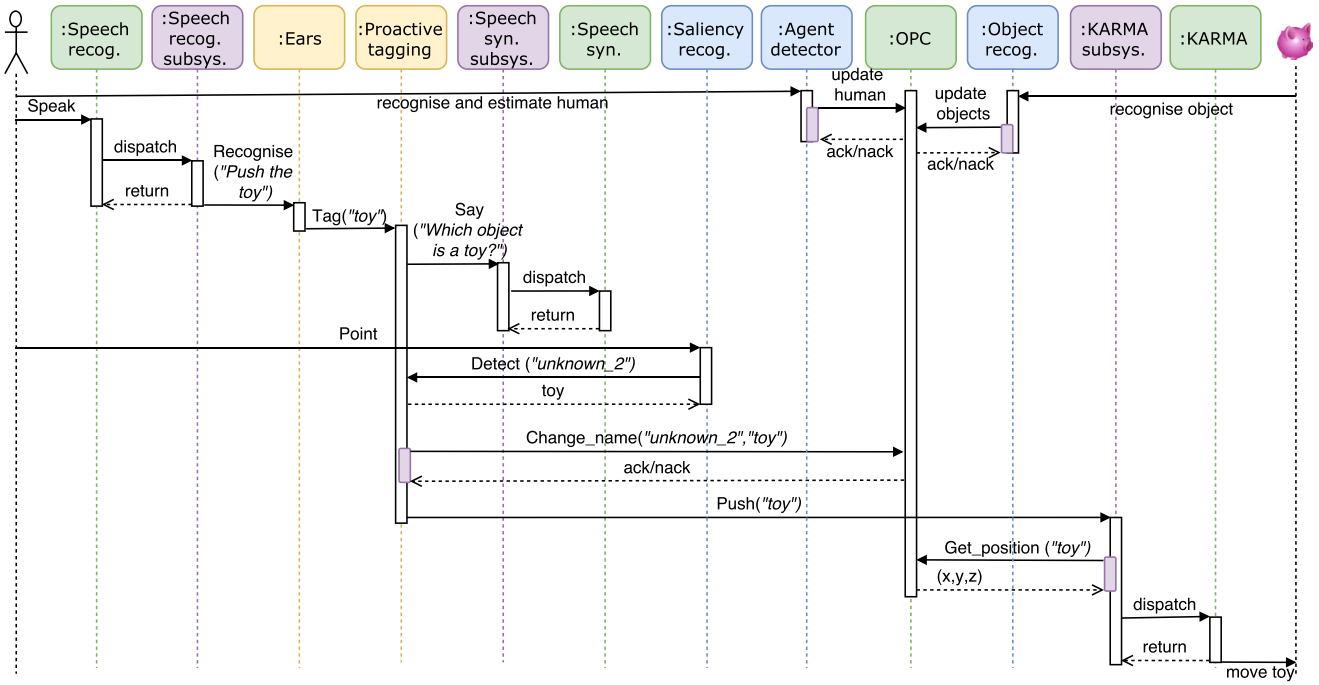


FIGURE 1 | Temporal UML diagram for an interaction where a human gives a speech command to the iCub to push an object which is currently unknown to the robot. The diagram depicts the involved modules and subsystems, and shows the information flow. After converting the speech command in an action plan, the robot first asks the human to indicate the desired object, and subsequently pushes that object. The knowledge database is continuously being updated by the agent detector and object recognition system throughout the interaction, and the object name is updated after the human indicated the object by pointing to it. In our GitHub repository, we provide another diagram for the case that a drive threshold is hit, which triggers the behavior to tag an unknown object autonomously.

demonstrates the robustness of the framework in various environments.[13]

4.3. Usage within DAC-h3 Framework

An extended version of the knowledge acquisition tutorial has been used to solve the symbol grounding problem, acquire language capabilities, execute goal-oriented behavior, and express a verbal narrative of the robot's experience in the world, using the DAC-h3 framework (Moulin-Frier et al., 2017). The work of Moulin-Frier et al. (2017) also demonstrates that the software framework presented in this paper can be readily used to study human–robot interaction experiments with naive subjects.

From the engineering perspective, the library and modules of iCub-HRI have been embedded in the Distributed Adaptive Control architecture (DAC, mentioned in the Introduction). The DAC architecture proposes that the brain can be seen as a multi-layered control structure consisting of (1) the body (with its sensors, needs and effectors), (2) the reactive layer for reflexive predefined control, (3) the adaptive layer for state acquisition and model-free reinforcement learning, and (4) the contextual layer which acquires model-based goal-oriented policies. Across these layers, we can distinguish columns of systems that processes states of the environment, the self and action as depicted in **Figure 2**.

[13]https://github.com/robotology/wysiwyd—"Demonstration 4," up to the fourth minute of the video.

The implementation of iCub-HRI with its subsystems and working memory make it particularly suitable in any scenario where module integration is driven by a complex multi-layered control architecture, with heterogeneous modules communicating within and between the different control layers.

4.4. More Applications and Use Cases

The central advantage of iCub-HRI is that the library bypasses the requirement for obtaining a working knowledge of the operation of a large range of modules during the normal operation of the iCub and their interaction before starting to develop one's specific application on top of these modules. Furthermore, iCub-HRI's modular subsystem architecture means that one can easily integrate applications developed on top of iCub-HRI to further abstract and accelerate the development of robotics applications.

The underlying design principles of iCub-HRI (see Section 1.2) and the high-level abstractions of the robot's basic input and output systems like speech, vision, and motor control allow a wide, varied range of use cases. For users with a non-technical background, it significantly reduces the learning curve to exploit the iCub robotic platform, with potential applications such as robotic art, research into the societal effects of robotics, investigations into human–robot collaboration and human–robot interaction studies investigating the psychological effects of such an interaction. For users more familiar with the iCub, the flexibility of the library allows them to focus on the core of their applications, where iCub-HRI provides a bridge to quickly integrate

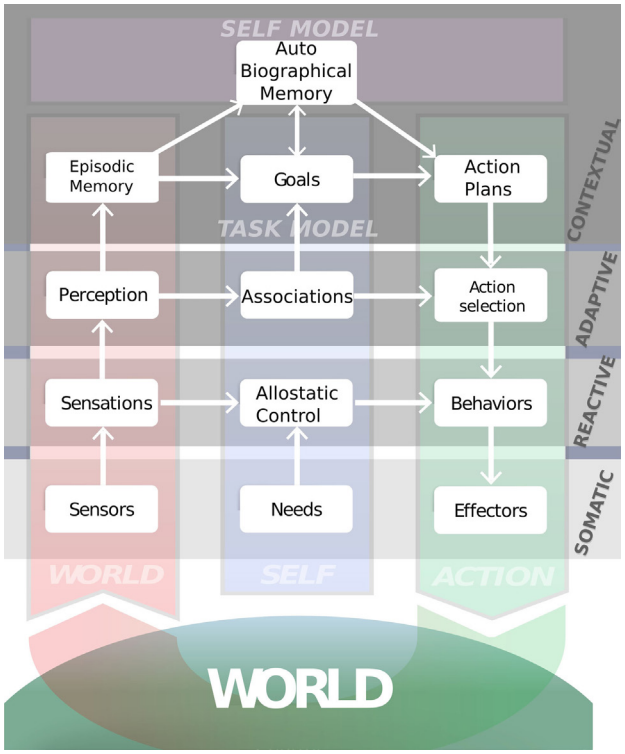


FIGURE 2 | iCub-HRI serves as underlying software framework for the depicted DAC-h3 cognitive architecture (see text for more detail). The usage within DAC-h3 has shown that several design principles were successfully implemented: iCub-HRI was easy to adapt and was extended with several other modules. Furthermore, the user study presented by Moulin-Frier et al. (2017) was directly based on the knowledge acquisition tutorial presented in Section 4.2.

these applications with the sensory, motor, and affective systems of the robot. This reduces the implementation effort which leads to faster developments, and allows for accelerated prototyping of embodied artificial intelligence applications.

4.5. Platform Independence

This paper specifically aims to provide a software framework to be used on the iCub humanoid robot. However, due to the modular design of the framework, certain components could be used on other robotic platforms as well, as they do not directly interface with the iCub's sensors and/or actuators and are hence robot agnostic. The following components are platform independent and can directly be used on other robots[14]:

- Working memory (Section 2.1).
- Perception modules: agent detector (Section 3.1.1), speech recognition (Section 3.1.2), saliency (Section 3.1.4), as well as face and action recognition (Section 3.1.5).
- Reactive layer (Section 3.3.2); the actions executed by the drives can be easily adjusted in a configuration file.
- Tools (Section 3.4): the *opcPopulator* as well as the *referenceFrameHandler*.

All other components are tailored for the iCub and would need to be re-implemented or substituted with alternatives on another platform.

4.6. Dependencies

The only hard dependencies of iCub-HRI are a C++ 11 compatible compiler and YARP. Due to the aspiration to combine various components within a single architecture, there are a number of soft dependencies: *OpenCV*, *IOL*, and *superquadric-model* for object tracking, *kinect-wrapper* to track the human partner, the *speech* repository for speech synthesis and recognition, as well as (a modified version of) *KARMA* for object manipulation. All dependencies are released under free software licenses, specifically LGPLv2.1 for YARP, BSD-3-Clause in case of OpenCV and GPLv2 for all other dependencies.

The installation of these components is further detailed in the iCub-HRI repository and we provide a Python script to easily keep all dependencies up-to-date. It is also possible to download or compile a Docker image which contains all required and optional libraries pre-installed and configured.

4.7. Download, Licensing, and Compatibility

The code is available for download on the designated GitHub repository[15] alongside the documentation (including class diagrams) and tutorials. It is released under the free software license GPLv2. The build status is continuously monitored on Windows, Linux, and macOS. The code itself can be considered stable and has been in adapted from the code which was used in the EFAA and WYSIWYD projects for several years.

5. CONCLUSION AND FUTURE WORK

We presented iCub-HRI, a software framework which integrates various components available within the iCub ecosystem and makes them easily accessible by the means of method calls. iCub-HRI can be used in various ways, from a very lightweight library up toan integrated platform for studies on human–robot interaction. While it is tailored for the iCub humanoid robot, many parts are platform independent and can be used on other robotic platforms as well. We provide a full documentation and various tutorials, allowing researchers to easily adapt iCub-HRI for their purposes.

One limitation of the presented framework is that while it facilitates communication between different modules, it does not have any means of manipulating the execution of individual modules. This is a disadvantage in case of, e.g., monitoring real-time constraints, which cannot be guaranteed on the framework level but only within individual components (this is the case for the low-level Cartesian controller employed by ARE and KARMA (Pattacini et al., 2010)). Furthermore, as a

[14]Provided they run on YARP, or can be interfaced with YARP through, e.g., the YARP-ROS interoperation.

[15]https://github.com/robotology/icub-hri.

central memory is being employed, there is a delay of the information flow from one module to another. Another limitation of this work is that no test data are being provided. Providing a proper test-suite is beyond the scope of this research, as it would need to write test cases for several tens of modules (many of them being external), and their communication handled by over 100 YARP ports. Writing suitable test cases using the testing framework presented by Natale et al. (2016) is an interesting research idea which we would like to tackle in future works.

A key point for the future adaptation of iCub-HRI will be the integration of new components from within the iCub ecosystem as well as state of the art software from related disciplines. For example, we intend to replace the current object tracking functionality with a state of the art object tracker (Choi et al., 2017); and to embed the reaching-with-avoidance framework (Nguyen et al., 2016; Roncone et al., 2016) for safer robot actions.

ETHICS STATEMENT

The research protocol was approved by the Parc de Salut MAR—Clinical Research Ethics Committee.

AUTHOR CONTRIBUTIONS

TF, CM-F, DC, and J-YP drafted the initial version of the paper. TF, SL, and PN designed and implemented the iCub-HRI library. TF, J-YP, DC, PN, CM-F, and SL designed and implemented the iCub-HRI modules. TF, PN, J-YP, and SL documented the code and wrote tutorials. TF, J-YP, DC, PN, and CM-F conceived and performed the knowledge acquisition demonstration. CM-F, J-YP, and PV designed the DAC-h3 cognitive architecture. PV conceived and coordinated the EFAA and WYSIWYD projects including the proactive tagging benchmark. GM, TP, PV, and YD created the idea, were significantly involved in reviewing manuscript drafts, and supervised the project.

ACKNOWLEDGMENTS

The authors would like to thank all participants of the WYSIWYD project who contributed to the code which made writing this paper feasible.

FUNDING

The research leading to these results has received funding under the European Union's Seventh Framework Programme (FP/2007-2013)/ERC Grant Agreement n. FP7-ICT-612139 (WYSIWYD—What You Say Is What You Did) and FP7-ICT-270490 (EFAA—The Experimental Functional Android Assistant). PN was supported by a Marie Curie Early Stage Researcher Fellowship (H2020-MSCA-ITA, SECURE 642667). PV was supported by the ERC advanced grant 341196 (cDAC—Role of Consciousness in Adaptive Behavior).

REFERENCES

Choi, J., Chang, H. J., Yun, S., Fischer, T., Demiris, Y., and Choi, J. Y. (2017). "Attentional correlation filter network for adaptive visual tracking," in *IEEE Conference on Computer Vision and Pattern Recognition* (Honolulu, HI), 4807–4816.

Damianou, A., Titsias, M. K., and Lawrence, N. D. (2011). "Variational Gaussian process dynamical systems," in *Advances in Neural Information Processing Systems* (Granada, Spain), 2510–2518.

Elkady, A., and Sobh, T. (2012). Robotics middleware: a comprehensive literature survey and attribute-based bibliography. *J. Robot.* 2012, 1–15. doi:10.1155/2012/959013

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley Professional.

Fanello, S., Pattacini, U., Gori, I., Tikhanoff, V., Randazzo, M., Roncone, A., et al. (2014). "3D stereo estimation and fully automated learning of eye-hand coordination in humanoid robots," in *IEEE-RAS International Conference on Humanoid Robots* (Madrid, Spain), 1028–1035.

Fischer, T., and Demiris, Y. (2016). "Markerless perspective taking for humanoid robots in unconstrained environments," in *IEEE International Conference on Robotics and Automation* (Stockholm, Sweden), 3309–3316.

Fitzpatrick, P., Ceseracciu, E., Domenichelli, D. E., Paikan, A., Metta, G., and Natale, L. (2014). A middle way for robotics middleware. *J. Softw. Eng. Robot.* 5, 42–49. doi:10.6092/JOSER_2014_05_02_p42

Fitzpatrick, P., Metta, G., and Natale, L. (2006). YARP: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3, 43–48. doi:10.5772/5761

Foote, T. (2013). "tf: the transform library," in *IEEE Conference on Technologies for Practical Robot Applications* (Woburn, MA, USA).

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object Oriented Software*. Boston, MA: Addison-Wesley.

Huang, C. M., and Mutlu, B. (2012). "Robot behavior toolkit: generating effective social behaviors for robots," in *ACM/IEEE International Conference on Human-Robot Interaction* (Boston, MA), 25–32.

Jang, M., Kim, J., and Ahn, B. K. (2015). "A software framework design for social human-robot interaction," in *International Conference on Ubiquitous Robots and Ambient Intelligence* (Goyang, South Korea), 411–412.

Kalal, Z., Mikolajczyk, K., and Matas, J. (2012). Tracking-learning-detection. *IEEE Trans. Pattern Anal. Mach. Intell.* 34, 1409–1422. doi:10.1109/TPAMI.2011.239

Krupke, D., Starke, S., Einig, L., Steinicke, F., and Zhang, J. (2017). "Prototyping of immersive HRI scenarios," in *International Conference on Climbing and Walking Robots and the Support Technologies for Mobile Machines* (Porto, Portugal), 537–544.

Lallée, S., Hamann, K., Steinwender, J., Warneken, F., Martienz, U., Barron-Gonzales, H., et al. (2013). "Cooperative human robot interaction systems: IV. Communication of shared plans with Naïve humans using gaze and speech," in *IEEE International Conference on Intelligent Robots and Systems* (Tokyo, Japan), 129–136.

Lallée, S., and Verschure, P. (2015). How? Why? What? Where? When? Who? Grounding ontology in the actions of a situated social agent. *Robotics* 4, 169–193. doi:10.3390/robotics4020169

Lallée, S., Vouloutsi, V., Blancas, M., Grechuta, K., Puigbo, J.-Y., Sarda, M., et al. (2015). Towards the synthetic self: making others perceive me as an other. *Paladyn J. Behav. Robot.* 6, 136–164. doi:10.1515/pjbr-2015-0010

Lane, I., Prasad, V., Sinha, G., Umuhoza, A., Luo, S., Chandrashekaran, A., et al. (2012). "HRItk: the human-robot interaction ToolKit rapid development of speech-centric interactive systems in ROS," in *NAACL-HLT Workshop on Future Directions and Needs in the Spoken Dialog Community: Tools and Data* (Montreal, Canada), 41–44.

Martinez-Hernandez, U., Damianou, A., Camilleri, D., Boorman, L. W., Lawrence, N., and Prescott, T. J. (2016). "An integrated probabilistic framework for robot perception, learning and memory," in *IEEE International Conference on Robotics and Biomimetics* (Qingdao, China), 1796–1801.

Mathews, Z., i Badia, S. B., and Verschure, P. F. M. J. (2012). PASAR: an integrated model of prediction, anticipation, sensation, attention and response

for artificial sensorimotor systems. *Inf. Sci.* 186, 1–19. doi:10.1016/j.ins.2011.09.042

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi:10.1016/j.neunet.2010.08.010

Moulin-Frier, C., Fischer, T., Petit, M., Pointeau, G., Puigbo, J.-Y., Pattacini, U., et al. (2017). DAC-h3: a proactive robot cognitive architecture to acquire and express knowledge about the world and the self. *IEEE Trans. Cogn. Dev. Syst.* doi:10.1109/TCDS.2017.2754143

Natale, L., Paikan, A., Randazzo, M., and Domenichelli, D. E. (2016). The iCub software architecture: evolution and lessons learned. *Front. Robot. AI* 3:24. doi:10.3389/frobt.2016.00024

Nguyen, P. D., Hoffmann, M., Pattacini, U., and Metta, G. (2016). "A fast heuristic Cartesian space motion planning algorithm for many-DoF robotic manipulators in dynamic environments," in *IEEE-RAS International Conference on Humanoid Robots* (Cancun, Mexico), 884–891.

Pasquale, G., Ciliberto, C., Odone, F., Rosasco, L., and Natale, L. (2015). "Teaching iCub to recognize objects using deep convolutional neural networks," in *Workshop on Machine Learning for Interactive Systems* (Lille, France), 21–25.

Pattacini, U., Nori, F., Natale, L., Metta, G., and Sandini, G. (2010). "An experimental evaluation of a novel minimum-jerk Cartesian controller for humanoid robots," in *IEEE/RSJ International Conference on Intelligent Robots and Systems* (Taipei, Taiwan), 1668–1674.

Petit, M., Fischer, T., and Demiris, Y. (2016). Lifelong augmentation of multi-modal streaming autobiographical memories. *IEEE Trans. Cogn. Dev. Syst.* 8, 201–213. doi:10.1109/TAMD.2015.2507439

Petit, M., Lallee, S., Boucher, J.-D., Pointeau, G., Cheminade, P., Ognibene, D., et al. (2013). The coordinating role of language in real-time multimodal learning of cooperative tasks. *IEEE Trans. Auton. Ment. Dev.* 5, 3–17. doi:10.1109/TAMD.2012.2209880

Puigbò, J.-Y., Moulin-Frier, C., and Verschure, P. F. (2016). "Towards self-controlled robots through distributed adaptive control," in *Conference on Biomimetic and Biohybrid Systems* (Edinburgh, Scotland), 490–497.

Quigley, M., Conley, K., Gerkey, B. P., Faust, J., Foote, T., Leibs, J., et al. (2009). "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, (Kobe, Japan).

Roncone, A., Hoffmann, M., Pattacini, U., Fadiga, L., and Metta, G. (2016). Peripersonal space and margin of safety around the body: learning visuo-tactile associations in a humanoid robot with artificial skin. *PLoS ONE* 11:e0163713. doi:10.1371/journal.pone.0163713

Sanchez-Fibla, M., Bernardet, U., Wasserman, E., Pelc, T., Mintz, M., Jackson, J. C., et al. (2010). Allostatic control for robot behavior regulation: a comparative rodent-robot study. *Adv. Complex Syst.* 13, 377–403. doi:10.1142/S0219525910002621

Sarabia, M., Ros, R., and Demiris, Y. (2011). "Towards an open-source social middleware for humanoid robots," in *IEEE-RAS International Conference on Humanoid Robots* (Bled, Slovenia), 670–675.

Tikhanoff, V., Pattacini, U., Natale, L., and Metta, G. (2015). "Exploring affordances and tool use on the iCub," in *IEEE-RAS International Conference on Humanoid Robots* (Atlanta, GA, USA), 130–137.

Verschure, P. F. M. J. (2012). Distributed adaptive control: a theory of the mind, brain, body nexus. *Biol. Inspired Cogn. Arch.* 1, 55–72. doi:10.1016/j.bica.2012.04.005

Verschure, P. F. M. J. (2016). Synthetic consciousness: the distributed adaptive control perspective. *Philos. Trans. R. Soc. Lond. B Biol. Sci.* 371, 263–275. doi:10.1098/rstb.2015.0448

Vezzani, G., Pattacini, U., and Natale, L. (2017). "A grasping approach based on superquadric models," in *IEEE International Conference on Robotics and Automation* (Singapore, Singapore), 1579–1586.

Viola, P., and Jones, M. (2001). "Rapid object detection using a boosted cascade of simple features," in *IEEE Conference on Computer Vision and Pattern Recognition* (Kauai, HI), I-511–I-518.

Vouloutsi, V., Grechuta, K., Lallée, S., and Verschure, P. F. (2014). "The influence of behavioral complexity on robot perception," in *Conference on Biomimetic and Biohybrid Systems* (Milan, Italy), 332–343.

Zambelli, M., and Demiris, Y. (2017). Online multimodal ensemble learning using self-learnt sensorimotor representations. *IEEE Trans. Cogn. Dev. Syst.* 9, 113–126. doi:10.1109/TCDS.2016.2624705

Zambelli, M., Fischer, T., Petit, M., Chang, H. J., Cully, A., and Demiris, Y. (2016). "Towards anchoring self-learned representations to those of other agents," in *Workshop on Bio-Inspired Social Robot Learning in Home Scenarios at IEEE/RSJ International Conference on Intelligent Robots and Systems*, (Daejeon, Korea).

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer, TD, declared a shared affiliation, though no other collaboration, with one of the authors, GM, to the handling editor.

frontiers in Robotics and AI

CODE
published: 29 March 2018
doi: 10.3389/frobt.2018.00024

Optimization-Based Controllers for Robotics Applications (OCRA): The Case of iCub's Whole-Body Control

*G. Jorhabib Eljaik, Ryan Lober, Antoine Hoarau and Vincent Padois**

Sorbonne Université, CNRS UMR 7222, Institut des Systèmes Intelligents et de Robotique, ISIR, Paris, France

OCRA stands for Optimization-based Control for Robotics Applications. It consists of a set of platform-independent libraries which facilitates the development of optimization-based controllers for articulated robots. Hierarchical, weighted, and hybrid control strategies can easily be implemented using these tools. The generic interfaces provided by OCRA allow different robots to use the exact same controllers. OCRA also allows users to specify high-level objectives via tasks. These tasks provide an intuitive way of generating complex behaviors and can be specified in XML format. To illustrate the use of OCRA, an implementation of interest to this research topic for the humanoid robot iCub is presented. OCRA stands for Optimization-based Control for Robotics Applications. It consists of a set of platform-independent libraries which facilitates the development of optimization-based controllers for articulated robots. Hierarchical, weighted, and hybrid control strategies can easily be implemented using these tools. The generic interfaces provided by OCRA allow different robots to use the exact same controllers. OCRA also allows users to specify high-level objectives via tasks. These tasks provide an intuitive way of generating complex behaviors and can be specified in XML format. To illustrate the use of OCRA, an implementation of interest to this research topic for the humanoid robot iCub is presented.

Keywords: whole-body controller, iCub, optimization, tasks, hierarchical, code:c++

OPEN ACCESS

Edited by:
Ugo Pattacini,
Fondazione Istituto Italiano di Technologia, Italy

Reviewed by:
Carlo Ciliberto,
University College London, United Kingdom
Matej Hoffmann,
Czech Technical University in Prague, Czechia

***Correspondence:**
Vincent Padois
vincent.padois@sorbonne-universite.fr

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *04 August 2017*
Accepted: *28 February 2018*
Published: *29 March 2018*

Citation:
Eljaik GJ, Lober R, Hoarau A and Padois V (2018) Optimization-Based Controllers for Robotics Applications (OCRA): The Case of iCub's Whole-Body Control. Front. Robot. AI 5:24. doi: 10.3389/frobt.2018.00024

1. INTRODUCTION

Whole-body control (WBC) is a research direction in robotics, where humanoids are faced with the problem of executing multiple tasks simultaneously. As stated by the IEEE Technical Committee on Whole-Body Control:

> A control system that is specifically designed to guarantee the execution of a single task, even if it uses all the joints of a robot, cannot be considered WBC.

This is indeed the core of the software introduced in this work, but it goes further by drawing additional requirements from the identification of typical concerns in the control of articulated robots, such as (1) standardization of the problem formulation, which is done in the form of an optimization problem; (2) flexibility in the solver choice; (3) independence of tasks from the problem formulation with user-friendly ways to introduce them; (4) addition of constraints, contact modeling and support for both fixed and floating-base robots. OCRA draws its origins from these design requirements. It stands for Optimization-based Control for Robotics Applications and consists of a set of

platform-independent libraries which facilitates the development of optimization-based controllers. It builds on top of ORC which was originally a framework developed by CEA-List,[1] later used at the Institute of Intelligent Systems and Robotics (ISIR) to develop whole-body controllers with simulations on XDE (Salini et al., 2013).

Examples of software addressing similar problems include the Stack of Tasks (SOT) (Mansard et al., 2009), OpenSOT (Rocchi et al., 2015), and CoDyCo[2] controllers (Nori et al., 2015). Nevertheless, they either lack the level of desired flexibility or do not meet the proposed design requirements. SOT and OpenSOT use strictly hiearchical methods, and while OpenSOT is intended for torque-controlled robots similar to OCRA, SOT originally targets velocity-controlled robots. When it comes to solvers, OpenSOT relies solely on QPOases while SOT's controller and solver are tight together.

Another software that has been used in the formulation of this type of controllers is Roboptim (2016). It is, however, an optimization framework for robotics and it is up to the user to formulate the control problem, workout the prioritization strategy and address the different components to achieve a whole-body controller.

CoDyCo's controllers on the other hand, although aimed at WBC, are tailored to be task-specific and do not constitute a WBC library.

OCRA has been designed to exploit a client–server paradigm, where the *server* is responsible for running the whole-body controller, send control inputs to the robot and host user-defined tasks, while the *client* is built by the user according to their needs on task servoing, planning, or higher-level control.

OCRA contributes to the building of the iCub mindware through the implementation of an *iCub server* along with communication utilities for the construction of clients. It facilitates the creation of a vast type of whole-body behaviors, with special attention to interaction. It also addresses the needs of different types of users, from the advanced one who desires to implement particular low-level control laws, to the more practical one who prefers to state at the metatask-level.

In Section 2, a generic overview of the main design requirements and features of OCRA, along with a list of software dependencies is presented. Section 3 introduces the main concepts involved in optimization-based control which allow the reader to have a deeper insight in the inner workings of the software. Concepts such as tasks, constraints, quadratic programming based control (and motivations for its use), prioritization strategies, and optimization solver are covered. Section 4 spans OCRA's structure, shedding light on its libraries and the main classes they are composed of as well as how these were used for iCub implementations. The same section continues with a more in-depth description of the iCub server and a generic client through sequence diagrams, as well as a brief explanation on how to automatically build a template client. Finally, Section 5 draws final conclusions.

[1] http://www-list.cea.fr/en/.

[2] European Project Whole-body Compliant Dynamical Contacts in Cognitive Humanoids.

2. OCRA

OCRA is a set of libraries and tools for the implementation of QP-based whole-body controllers for torque/force-controlled articulated robots. Robots like the humanoid iCub or the KUKA Light Weight Robot (LWR) manipulators (floating/fixed base) can be controlled using this open source software. In particular, for the iCub, the set of necessary libraries is implemented and distributed.

One main design requirement from OCRA's inception is that (1) it should be heavily task-oriented. This means, that a user can specify a set of tasks to be performed by the robot, e.g., *follow a CoM trajectory, while maintaining balance and make one hand follow another trajectory* and (2) the specifications of these tasks have to be easy to provide. This is achieved through an XML file that we call the *tasks set*.

Features that make OCRA flexible include: the possibility to choose between different types of tasks and their prioritization strategies; two different optimization solvers; various types of constraints and the tools to create a client–server architecture, where the *server* runs a reactive controller with the tasks and constraints, and one or more *clients* perform the computation of the right instantaneous tasks values through local trajectory controllers (e.g., PIDs), motion planning, model predictive control, or any higher-level control schemes.

The required dependencies of this software are given in **Table 1**.

3. OPTIMIZATION-BASED CONTROL

Traditionally, redundancy resolutions for robotic control problems find analytical solutions by ensuring that lower-priority tasks are executed in the null-space of higher-priority tasks. In prioritized inverse kinematics, acceleration or torque based control, the jacobian of low-priority tasks is projected onto the null-space of higher-priority ones (Khatib, 1987; Sentis and Khatib, 2006; Peters et al., 2008). Inequality constraints are, however, difficult to deal with in these approaches. They are usually transformed into avoidance tasks, which try to prevent the robot from hitting the original constraint (Khatib, 1986; Padois et al., 2007). This type of active avoidance (passive or active) method is doomed to fail as the number of constraints is necessarily higher than the number of DOF ($2n$ joints limits for an n DOF robot) and it thus requires to make decision reactively about which avoidance tasks should be used in order to guarantee the respect of all constraints while still

TABLE 1 | Required dependencies table for `ocra` and `ocra-icub`.

Dependency	*Minimum version*	*ocra*	*ocra-icub*
YARP	2.3	✓	✓
Eigen	3.2	✓	✓
orocos_kdl	1.2	✓	✓
iDynTree	0.4.0		✓
yarpWholeBodyInterface	0.35		✓
Boost	1.64	✓	✓
CMake	2.8.11	✓	✓
TinyXML	2.6.2	✓	
YCM	0.4.0		✓

For the sake of clarity, it is not shown that `ocra` *is naturally a dependency of* `ocra-icub`.

achieving the operational tasks in the most efficient way possible (Padois, 2016).

OCRA resorts to convex optimization for the formulation of the whole-body controller, as it has been stated multiple times before this point. The controller is written as a linearly constrained quadratic multi-objective optimization problem where strict or soft hierarchies are used to express the priorities between the tasks. *Linearly constrained* due to the constraints being strictly linear (or linearized if not), *quadratic* because each objective is the quadratic error of a task and *multi-objective* because multiple tasks are combined. The result of this optimization are the optimal actuation inputs to the system (i.e., joint torques) given the set of prioritized tasks to be performed and the constraints that have to be respected. Among these constraint, this optimization problem includes inequality constraints, coming from control input saturations or any other variable which should never cross certain limits. Under these conditions, the solution space can be proved convex and finding the optimal solution to the whole-body control problem is equivalent to finding the set of active constraints. In fact, methods in which optimization is avoided end up using algorithms that pretty much search for this active set, not explicitly and in a suboptimal way. It is then indisputable that the strong background in convex optimization outruns analytical methods used to heuristically activate constraints.

The primary concern of this section is to present the necessary equations and relationships to understand the critical aspects of the types of controllers which can be developed with OCRA. Generally speaking, an optimization-based controller formulates the control problem as one of minimizing control objective functions while respecting the control constraints. Specifically, the problem is formulated as a convex linearly constrained QP using the second-order rigid body dynamics of the robot. Therefore, the control objectives (Tasks) are expressed as either accelerations, torques, or wrenches, allowing for complex dynamic interactions with the environment, and the control constraints are expressed directly in the QP as linear equalities and inequalities.

3.1. Tasks

Tasks allow users to decompose complex whole-body behaviors into atomic control objectives, which can be planned by a user or automatically with planners. Here, a task represents a control objective for the robot, and more specifically, an error between some desired task value and the current value of the task in terms of the control variable. These tasks are expressed as the squared norm of these errors in either accelerations, torques, or wrenches and can be expressed in both joint and operational-space. In Section 3.4, the expression of these tasks in terms of the control variables is provided, but **Table 2**, below, shows their standard formulations.

In **Table 2**, $\boldsymbol{\nu}$ and $\dot{\boldsymbol{\nu}}$ are the generalized velocities and accelerations of the robot. They can be more or less directly related to the derivatives of the generalized coordinates $\boldsymbol{q}$. Indeed, for robots whose root link can float freely in Cartesian space, e.g., humanoids, it is necessary to consider the pose of the root link w.r.t. the world reference frame. The primary method for doing so is to account for the root link pose directly in the generalized coordinates, $\boldsymbol{q}$, of the robot (Sentis and Khatib, 2005; Mistry et al., 2010). The terms J and $\dot{J}$ are link Jacobians and their derivatives. The variable ${}^e\boldsymbol{\omega}$ represents an external wrench, and $\boldsymbol{\tau}$, the system torques, while $\ddot{\boldsymbol{\xi}}$ is operational-space acceleration. The corresponding *desired* values of each term in **Table 2** should not be confused with the raw trajectory given by the user (subscript *ref*). These set-points are used as inputs to a task-level PD controller in the case of operational-space acceleration tasks and a PI in the case of wrench (${}^e\boldsymbol{\omega}$) tasks, such that:

TABLE 2 | Different types of tasks.

Task	Definition
Operational-space acceleration	$T\left(\ddot{\boldsymbol{\xi}}^{\text{des}}\right) = \left\|J(\boldsymbol{q})\dot{\boldsymbol{\nu}} + \dot{J}(\boldsymbol{q},\boldsymbol{\nu})\boldsymbol{\nu} - \ddot{\boldsymbol{\xi}}^{\text{des}}\right\|$
Joint-space acceleration	$T\left(\dot{\boldsymbol{\nu}}^{\text{des}}\right) = \left\|\dot{\boldsymbol{\nu}} - \dot{\boldsymbol{\nu}}^{\text{des}}\right\|$
Operational-space wrench	$T\left({}^e\boldsymbol{\omega}^{\text{des}}\right) = \left\|{}^e\boldsymbol{\omega} - {}^e\boldsymbol{\omega}^{\text{des}}\right\|$
Joint torque	$T\left(\boldsymbol{\tau}^{\text{des}}\right) = \left\|\boldsymbol{\tau} - \boldsymbol{\tau}^{\text{des}}\right\|$

Superscript "des" stands for desired.

$$\ddot{\boldsymbol{\xi}}^{\text{des}}(t+\Delta t) = \ddot{\boldsymbol{\xi}}^{\text{ref}}(t+\Delta t) + K_p\boldsymbol{\epsilon}(t) + K_d\dot{\boldsymbol{\epsilon}}(t), \quad (1)$$

$${}^e\boldsymbol{\omega}^{\text{des}}(t+\Delta t) = {}^e\boldsymbol{\omega}^{\text{ref}}(t+\Delta t) + K_p\boldsymbol{\epsilon}(t) + K_i\int\boldsymbol{\epsilon}(t)dt, \quad (2)$$

where $\ddot{\boldsymbol{\xi}}^{\text{ref}}$ and ${}^e\boldsymbol{\omega}^{\text{ref}}$ are feedforward terms, while $\boldsymbol{\epsilon}$ and $\dot{\boldsymbol{\epsilon}}$ are pose error and its derivative (these being representation dependant). K_p, K_d, and K_i are proportional, derivative, and integral gains and by default, $K_d = 2\sqrt{K_p}$. Task servoing is necessary to compensate for drift and tracking errors associated with using second-order control techniques. Additionally, it is often the case that only position values are specified by the user, and these must be converted to accelerations—task servoing provides this service. For joint-space accelerations the servoing is done in similar fashion as for $\ddot{\boldsymbol{\xi}}^{\text{des}}$.

3.2. Constraints

As with all real world control problems, there are limits to what the system being controlled can do. For example, the control input is typically bounded, which for robots with revolute joints means that the torque which can be generated by the actuators is limited to plus or minus some value. Likewise, the joints themselves generally have limited operating ranges for various mechanical reasons. In addition to these common limiting factors, it may be reasonable to maintain the robot in some region of its state space that will ease control, e.g., avoid slipage of the contact points or avoid contact with the environment.

In **Table 3**, the $\bullet_{\min}$ and $\bullet_{\max}$ values represent the lower and upper limits of a variable. The term $C_{c_j}{}^e\boldsymbol{\omega}_j \leq 0$ represents the linearized friction cone constraint for a point contact, and ${}^eJ(q)\dot{\boldsymbol{\nu}} + {}^e\dot{J}(q,\boldsymbol{\nu})\boldsymbol{\nu} = 0$, its coupled "no motion" constraint, which ensures that the contact does not move. For details on these constraint expressions and the way to express them through linearization as functions of joint torques or generalized acceleration, the reader is directed to Salini et al. (2011). In addition to these nearly universal robotic constraints, particular care must be taken to ensure that the motions generated by the controller respect the system dynamics, i.e., the equations of motion.

TABLE 3 | Possible constraints in OCRA.

General constraint	*Equation*
Actuator limits	$\boldsymbol{\tau}_{\min} \leq \boldsymbol{\tau} \leq \boldsymbol{\tau}_{\max}$
Joint position limits	$\boldsymbol{q}_{\min} \leq \boldsymbol{q} \leq \boldsymbol{q}_{\max}$
Joint velocity limits	$\dot{\boldsymbol{\nu}}_{\min} \leq \dot{\boldsymbol{\nu}} \leq \dot{\boldsymbol{\nu}}_{\max}$
Contact constraints	$C_{c_j}{}^{e}\boldsymbol{\omega}_j \leq 0$
	${}^{e}J(\boldsymbol{q})\dot{\boldsymbol{\nu}} + {}^{e}\dot{J}(\boldsymbol{q},\boldsymbol{\nu})\boldsymbol{\nu} = 0$

3.3. Dynamics

The principle constraint of the controllers in OCRA is that of the system dynamics. This means that any solution found must be dynamically feasible, and consequently, respect the equations of motion,

$$M(\boldsymbol{q})\dot{\boldsymbol{\nu}} + \underbrace{C(\boldsymbol{q},\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{g}(\boldsymbol{q})}_{\boldsymbol{n}(\boldsymbol{q},\boldsymbol{\nu})} = S^{\top}\boldsymbol{\tau} + {}^{e}J^{\top}(\boldsymbol{q})\,{}^{e}\boldsymbol{\omega} \tag{3}$$

$$M(\boldsymbol{q})\dot{\boldsymbol{\nu}} + \boldsymbol{n}(\boldsymbol{q},\boldsymbol{\nu}) = S^{\top}\boldsymbol{\tau} + {}^{e}J^{\top}(\boldsymbol{q})\,{}^{e}\boldsymbol{\omega}. \tag{4}$$

In (3), $M(\boldsymbol{q})$ is the generalized mass matrix, $C(\boldsymbol{q}, \boldsymbol{\nu})\boldsymbol{\nu}$ and $\boldsymbol{g}(\boldsymbol{q})$ are the Coriolis-centrifugal and gravitational terms, S is a selection matrix indicating the actuated degrees of freedom, ${}^{e}\boldsymbol{\omega}$ is the concatenation of the external contact wrenches, and ${}^{e}J$ their concatenated Jacobians. Grouping $C(\boldsymbol{q}, \boldsymbol{\nu})\boldsymbol{\nu}$ and $\boldsymbol{g}(\boldsymbol{q})$ together into $\boldsymbol{n}(\boldsymbol{q}, \boldsymbol{\nu})$, we can simplify the equations to (4). Additionally, the variables $\dot{\boldsymbol{\nu}}$, τ, and ${}^{e}\boldsymbol{\omega}$, can be grouped into the same vector,

$$\boldsymbol{x} = \begin{bmatrix} \dot{\boldsymbol{\nu}} \\ \boldsymbol{\tau} \\ {}^{e}\boldsymbol{\omega}, \end{bmatrix} \tag{5}$$

forming the *control variable*, and allowing (4) to be rewritten as,

$$\underbrace{\begin{bmatrix} -M(\boldsymbol{q}) & S^{\top} & {}^{e}J^{\top}(\boldsymbol{q}) \end{bmatrix}}_{A}\boldsymbol{x} = \underbrace{\boldsymbol{n}(\boldsymbol{q},\boldsymbol{\nu})}_{b}. \tag{6}$$

Equation 6 provides an affine equality constraint, $A\boldsymbol{x} = \boldsymbol{b}$, which can be used to ensure that the minimization of the control objectives respects the system dynamics.

3.4. Quadratic Programming Based Control

Given the control objectives defined by the task errors from Section 3.1, the control constraints from Section 3.2, and the optimization variable defined by (5), we can now form a generic, single task, optimization-based whole-body control problem as,

$$\begin{aligned} \min_{\boldsymbol{x}} \quad & T_i(\boldsymbol{x}) \\ \text{s.t.} \quad & G\boldsymbol{x} \leq \boldsymbol{h} \\ & A\boldsymbol{x} = \boldsymbol{b}, \end{aligned} \tag{7}$$

where the objective function, $T_i(\boldsymbol{x})$, is the task error, representing for example, the squared error between a desired acceleration or wrench and the system's (see Section 3.1). The inequality constraints, generically represented by, $G\boldsymbol{x} \leq \boldsymbol{h}$, contain the concatenation of all of the affine inequalities defined in **Table 3**, while the affine equality constraints, shown by $A\boldsymbol{x} = \boldsymbol{b}$, obligatorily contain the equation of motion constraints from (6), and possibly the coupled "no motion" constraints of any contacts which might be active.

The form of this problem will be referred to throughout this work as the *full problem*, which is also the default formulation used in OCRA. The user can choose to work with the *reduced problem*, in which the dynamics are not explicit in the constraints, but projected onto the different control objectives, and with the optimization variable, $\boldsymbol{x}$, in this case, consisting of the control inputs, $\boldsymbol{\tau}$, and external wrenches ${}^{e}\boldsymbol{\omega}$, i.e., $\boldsymbol{x} = [\boldsymbol{\tau}^{\top} \quad {}^{e}\boldsymbol{\omega}^{\top}]^{\top}$. The reduced problem has the advantage of having less optimization variables, which can improve the solution time as shown in Section 3.5 of Salini (2012), at the expense of complicating the writing of the tasks and constraints in terms of the optimization variable. The inclusion of the generalized joint accelerations, $\dot{\boldsymbol{\nu}}$, in the full problem, yields clarity and simplicity when writing the cost functions and the constraints on the joint velocities, acceleration and joint limits.

3.5. Prioritization Strategies

Up to this point, only one task objective function is considered in the whole-body controller in Section 3.4. If multiple task objective functions are combined (using operations that preserve convexity) in the resolution of the control problem, then they can be performed simultaneously. In these cases, it is important to select a strategy for the resolution of the optimization problem. The strategy will in turn, determine how tasks interact/interfere with one another. The two prevailing methods for dealing with multiple tasks are hierarchical (Saab et al., 2013; Escande et al., 2014) implemented as WOCRA and weighted prioritization (Bouyarmane and Kheddar, 2011; Salini et al., 2011) implemented as HOCRA. A hybrid scheme can also be used providing the best of the former two methods (Liu et al., 2016).

4. SOFTWARE

4.1. Structure

4.1.1. OCRA Libraries

The main concepts introduced in previous sections are materialized in the different interfaces, abstract, and concrete classes OCRA is composed of. These are encapsulated in four essential components or libraries. These are: `ocra-optim`, `ocra-control`, `ocra-coms`, and `ocra-utils`.

The first of these libraries, `ocra-optim`, defines the lowest-level data structures required to build an optimization problem such as variables, functions, and constraints, as well as the basic concept of a solver and prioritization strategies. **Table 4** shows the main classes in this library, their type, and a brief description.

The `ocra-control` library goes up one level of abstraction, containing all the classes necessary to build the model of a robot, implement a control law, account for the floating-base dynamics and build the different types of tasks, constraints and trajectories. The two main prioritization techniques described in Section 3.5 are, respectively, implemented through `HOCRA` and `WOCRA`. Again, the main classes in this library along with their brief description are collected in **Table 5**.

TABLE 4 | Main classes composing the `ocra-optim` library.

ocra-optim	
Main classes	**Features**
`Variable` abstract	Represents the mathematical concept of variable
`Function` concrete	Base for any type of function
`Constraint` concrete	Templated base class to build equality/inequalities constraints
`LinearizedCoulombFunction` concrete	Builds a discretized cone representing a Coulomb Friction cone
`Solver` abstract	Base class for optimization solvers
`CascadeQPSolver` implementation	Implements a hierarchical solver
`OneLevelSolver` abstract	Used for building solvers with one level of importannce to all tasks. It also contains specific implementations with `QuadProg++` and `QPOases`. This is the solver used in `wocra`

Blue labels indicate abstract classes that can be later implemented. Orange labels are assigned instead to concrete classes without particular inheritances, while green labels stand for implemented classes.

TABLE 5 | Main classes composing the `ocra-control` library.

ocra-control	
Main classes	**Features**
`Controller` abstract	Used to implement control laws
`Model` abstract	Provides dynamic and static terms from the equations of motion
`FullDynamicEquationFunction` abstract	Creates the dynamics equation as a linear function of the optimization variable
`ModelContacts` concrete	Concatenates the contact variables and Jacobians for a model
`ControlFrame` interface	Generic representation of a frame
`Feature` interface	Used by tasks to compute errors and Jacobians
`Task` concrete	–
`TaskBuilder` abstract	Builds task-specific features
`*TaskBuilder` implementation	Task-specific implementations of `TaskBuilder`. "*" is replaced by Com, FullPosture, Orientation, etc.
`TaskConstructionManager` abstract	–
`*LimitConstraint` implementation	(torque and joint limits)
`Trajectory` concrete	Helper class to build trajectories. These can be minimum jerk, linearly interpolated, gaussian processes or time-optimal
`WocraController` implementation	QP-based controller using a weighted prioritization strategy
`HocraController` implementation	QP-based controller using a hierarchical prioritization strategy

Blue labels indicate abstract classes that can be later implemented. Orange labels are assigned instead to concrete classes without particular inheritances. Red labels to interfaces and green labels to implementations.

The last two libraries are agnostic to the paradigm suggested by OCRA. That is, a client–server model. In order to implement it, the `ocra-coms` library is provided and comes with the generic classes to create a server and a client and to manage the communication between them. **Table 6** lists the main classes in this library along with their description.

TABLE 6 | Main classes composing the `ocra-coms` library.

ocra-coms	
Main classes	**Features**
`ControllerServer` abstract	Must be inherited to implement the server side
`ServerCommunications` concrete	Helps the server establish YARP-based communication with the client
`ClientCommunications` concrete	Helps the client establish YARP-based communication with the server
`ClientManager` concrete	Implements the functionalities of `YARP RFModule` on the client side. Holds the main client thread
`ControllerClient` abstract	Implements the functionalities of `YARP RateThread` on the client side. Main thread hosted by `ClientManager`
`TaskConnection` concrete	Used on the client side to connect and communicate with the tasks started by the server
`TrajectoryThread` concrete	Used to create trajectories on the client side

Blue labels indicate abstract classes that can be later implemented. Orange labels are assigned instead to concrete classes without particular inheritances.

TABLE 7 | Main classes composing the `ocra-icub` library.

ocra-icub	
Main classes	**Features**
`ModelInitializer` concrete client	Retrieves model configuration information from the server to create a local copy of the robot model
`OcraWbiModel` implementation client	Implements the abstract Model class from `ocra-control` for the iCub robot
`IcubControllerServer` implementation server	Implements `ControllerServer` for the iCub robot
`Module` implementation server	Module that launches the controller thread, parses controller options and the tasks set XML. Basically a `yarp:os:RFModule`
`Thread` implementation server	Main controller thread started. Created by `Module`, contains the controller, tasks manager, and solves the whole-body control problem

Orange labels mean concrete class without any particular inheritance. Green labels are for classes that implement some base class from the main OCRA libraries. Yellow labels stand indicate classes that are used to build a client, while gray labels are for those used to build a server.

Finally, the `ocra-utils` library as its name states, is a set of utilities to aid the other libraries: helpers to perform file operations, xml parsing, data structure conversions, errors descriptors, among others.

4.1.2. OCRA for iCub

The classes needed to *implement* a server for the iCub robot and a generic client are present in the `ocra-icub` library. As can be seen from the green implementation labels in **Table 7**, most of the main classes are implementations of base classes from `ocra-control` and `ocra-coms`. In the following section, two main detailed explanations are provided: how to use these classes to obtain a client–server architecture for iCub, and how objects of the different classes interact.

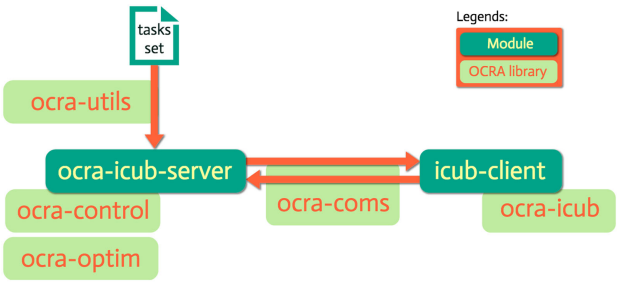


FIGURE 1 | A server (ocra-icub-server) and a client (icub-client) are here represented in dark green as YARP modules. In light green, we see the underlying OCRA libraries associated to their construction, as well as for the communication between them and the parsing of the tasks set.

Given the classes involved in the construction of this task-oriented, client-server paradigm for whole-body control, as well as the particular implementations for iCub, we present for the sake of clarity in **Figure 1** an illustration of a typical server–client architecture with the underlying OCRA libraries used to build each component. This section proceeds with a time-based illustration of the interaction logic between the different objects of our system in the form of *sequence diagrams* (IEEE, 2009) as shown in **Figures 2** and **3**. Given the amount of classes in the package, it might be difficult to see the global interaction among them along with the intended architecture. The next two sections attempt to clear this out by showing the inner interactions of both client and server, independently and between them.

4.1.3. iCub Server

Figure 2 depicts the sequence diagram for the `ocra-icub-server`. The user starts by executing the server from terminal issuing the command `ocra-icub-server [options]` (1).

The default options are specified in its initialization file `ocra-icub-server.ini` or hardcoded in the source code. After the execution of the server, an object of type `ResourceFinder` is created, which is responsible for the parsing of the former *options*. Right after, a yarp `RFModule` is created (3) and started (4), whose first task will be to configure the server (6), ask the `ResourceFinder` to find the desired type of controller (7), i.e., WOCRA or HOCRA, the solver to be used, i.e., QUADPROG or QPOASES, the XML file with the description of the tasks that the client will manipulate, etc. At this point, a `yarpWholeBodyInterface` object is created (8) and initialized. This class serves as an interface to the robot, and as such will allow us to set the control references obtained, as well as to obtain the state of the robot. Now the module is ready to create (12) and start (13) the main thread of the client.

Before entering the main loop of the thread, however, a couple of objects of interest are created. First, an object of type `IcubControllerServer` (14), which during initialization (16) will create the desired controller with its internal solver. At this phase, also communication ports are opened with standardized names that will be used by the cient for future connections. `IcubControllerServer` is then asked by the thread to update its internal model of the robot (17) and add the tasks specified by the user via XML (18). This process involves the creation (19) of an object of type `TaskConstructionManager` which will create one or multiple instances (20) of `TaskBuilder`, one per type of task found in the XML. These task objects will then get added to `iCubControllerServer` (21). Notice how the tasks are *living in the server*. The server will then ask the `yarpWholeBodyInterface` object to set the torque control mode on the robot (22) for it to accept torque references. The latter are computed every cycle of the `Thread` (24–27) by `iCubControllerServer`.

The server will be constantly controlling the robot to achieve default initial states of the specified tasks. As an example, if one task is of COM type, it controls the robot to keep it at its initial position, until a client connects to the server and tells it to do otherwise. Finally, if the user decides to stop the server (28), the sequence of object "destructions" is illustrated from (29) to (37).

4.1.4. Generic Client

A client's main goal is to connect to the server to provide reference trajectories to the tasks it hosts. Let us show through **Figure 3** the main interactions within a client and the type of communication it establishes with the server.

As done previously on the server side, we are going to follow the sequence diagram in an orderly fashion. First, notice how before the user can start a client, they need to start the server. This is evident by the sequence number (2) next to `example-client`. Thus, having a server properly started, the client is launched and the first thing it does is to get model information of the robot through the class `ModelInitializer`. This is the first interaction between the client and the server (4-5), after which a local model of the robot is built (6). Once the client has access to the robot model, the main client thread is created (7). This is of type `ControllerClient` which is a Yarp `RateThread`. The creation of the thread is followed by a `ClientCommunications` object (8), which creates and connects local ports to the server for inter-process communications. Its role will become clearer later on. The client thread is passed to a `ClientManager` object (10) which will handle the life-cycle of the thread and its configuration (11–12). The module subsequently starts (18) the client thread, which afters initialization will spawn a couple of objects of interest.

Given the tasks contained in the XML file (taskSet) and fed to the server, the client will create one or more `TaskConnection` objects (18) for each of those tasks that are to be manipulated. Although not depicted in the diagram, for the sake of clarity, these objects will open control ports that are then connected to their corresponding tasks on the server side (19). It is through these objects that the client will be able to send task-specific messages to get or set their state.

As it is often the case, the user might want to create reference trajectories (of even different types) for all or some of the tasks. To this end one or more objects of type `TrajectoryThread` are created (20). These, at the same time, will internally create `TaskConnection` objects again to set the references to the tasks on the server (21). The client thread can then start the trajectory threads (23) and run in the background until it receives new references (25–29).

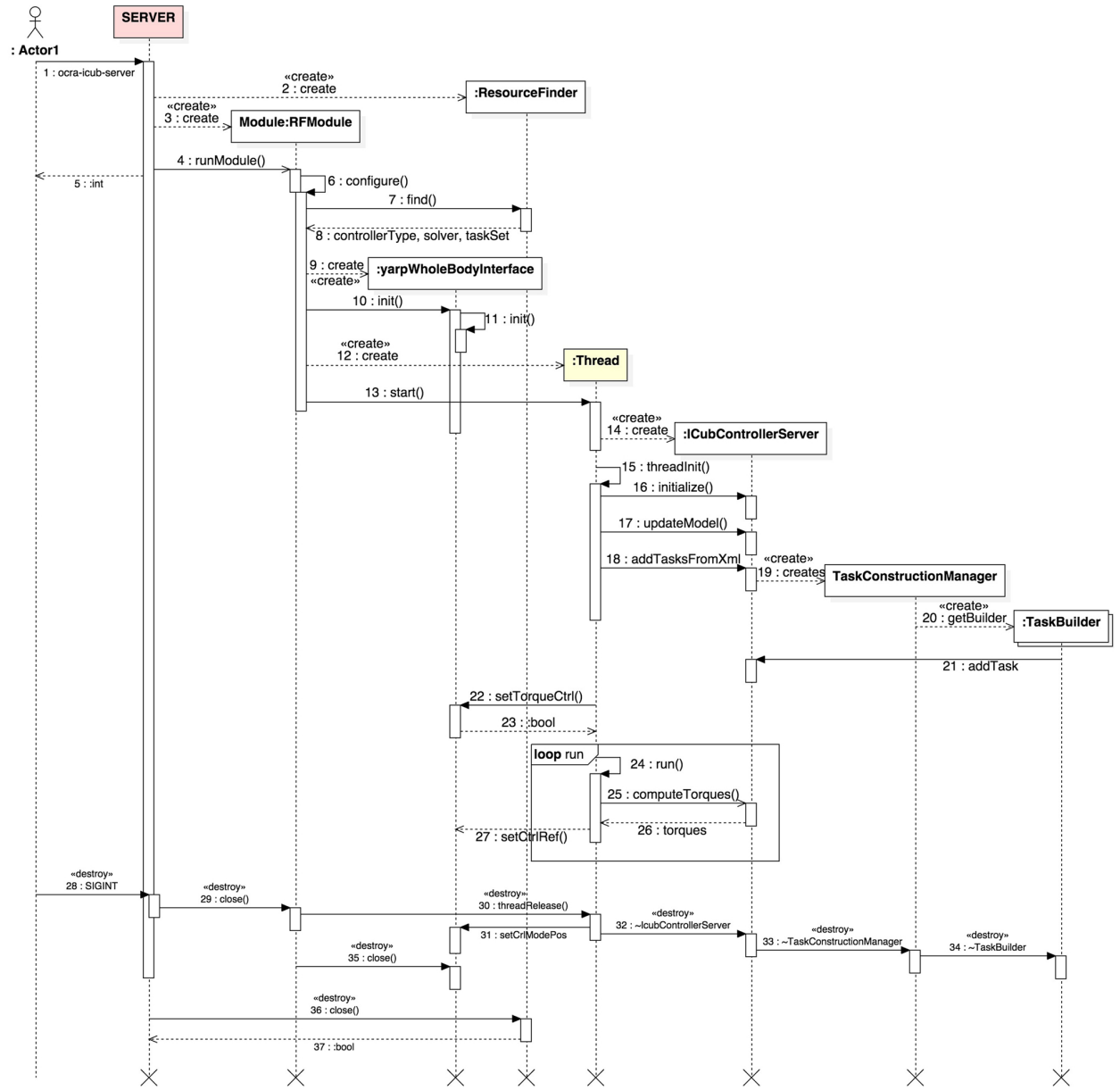


FIGURE 2 | UML sequence diagram displaying the typical interactions within the ocra-icub-server. The time evolution of interactions is followed from top to bottom, while messages passed among objects are found in the horizontal dimension. The light yellow background of some lifelines indicates that these are threads.

Now that the client has created task connections and trajectory threads, the client logic starts in the main thread (30–40). In this main loop, the client can:

- Get or set task-specific states through the `TaskConnection` objects (31–34).
- Add, remove or get tasks through the `ClientCommunications` object (35–38).
- Set references to tasks trajectories through the `TrajectoryThread` objects (39–40).

In order to stop the client, the user can send a `SIGINT` signal (`ctrl + c`) to kill the process and the sequence of "destructions" will be as in (43–53).

In Section 5.2, a link to a short tutorial can be found where it is explained how to launch a server and client.

4.1.5. Client Generator

Because each new iCub controller client requires the same basic setup, a helper tool has been developed to automatically scaffold out the minimum required code for a new client. Invoking `icub-client-generator [name-of-client]` from the command line will produce a directory called `name-of-client/`, with all of the minimum client requirements and a complete `CMake` build. One then needs only to edit the `name-of-client.cpp` file and add control logic. Therefore, anyone can write an iCub client in just a few minutes.

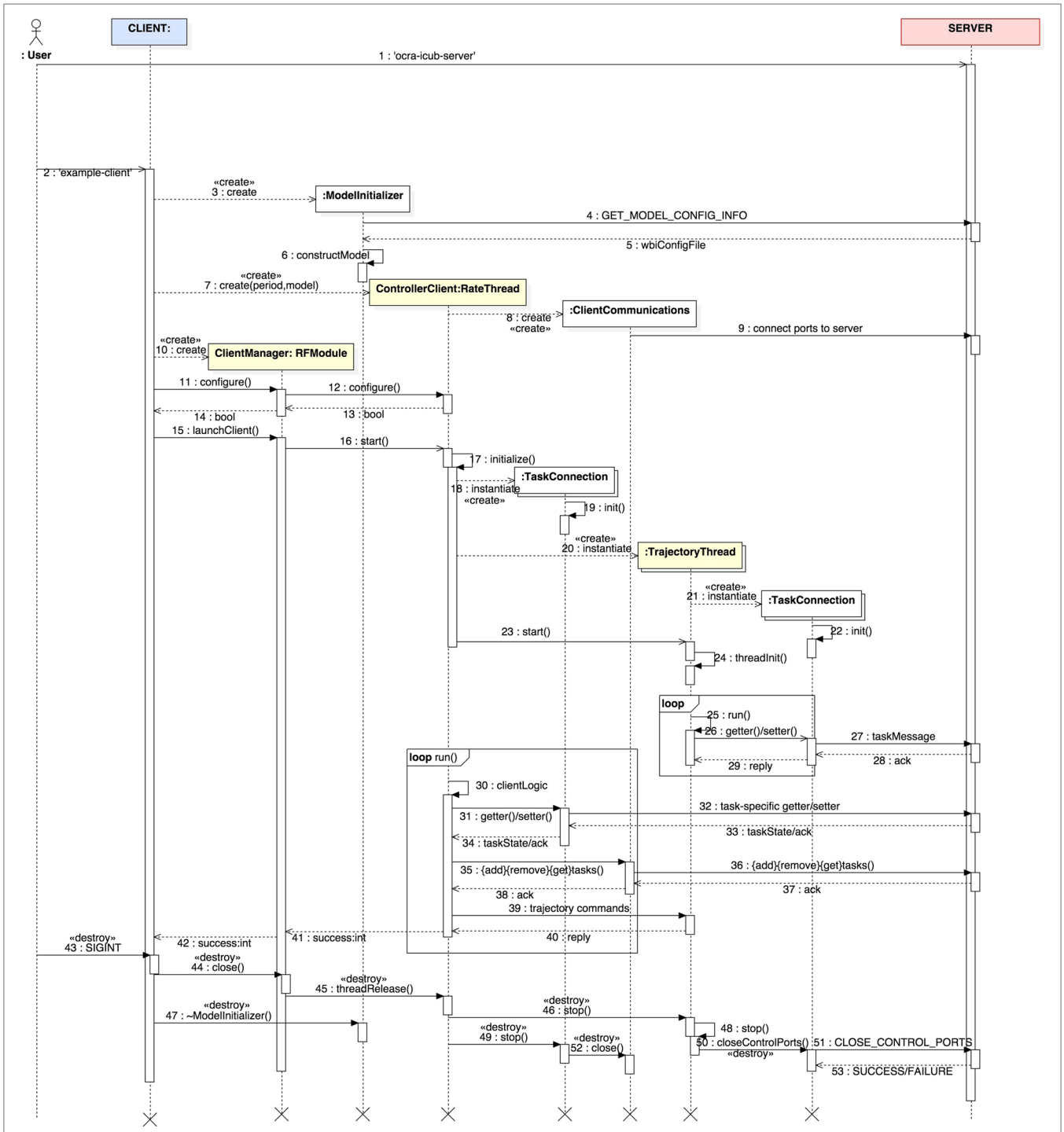


FIGURE 3 | UML sequence diagram displaying the typical interactions within a generic client. The time evolution of interactions is followed from top to bottom, while messages passed among objects are found in the horizontal dimension. The light yellow background of some lifelines indicate that these are threads.

5. CONCLUSION

The development of intelligent and autonomous robots entails many challenges, one of which is robust and flexible controllers. The overall goal of any control software should be to abstract the control of redundant robots, such as the iCub, to higher and higher levels of logic in order to facilitate the generation of complex overall behaviors—behaviors, which should ultimately render the robot useful. Whole-body control was born from these requirements and lays forth the design criteria for OCRA presented in Section 1. Through its various abstract and concrete classes, and server–client structure, OCRA attempts to provide a solution

which meets these needs but also balances ease of use with flexibility. The design of OCRA allows users to interact with and customize the control problem at virtually any level from the real-time computation of joint torques to high-level controller clients. This wide array of usability means that OCRA is suitable for any user from control experts to control novices. We believe that this is an important step toward improving the usability of such software because the learning curve should be simple for those who only want a functioning controller, but the software should also be flexible enough to allow users to experiments with fundamental concepts.

At the low-level, this is accomplished by abstracting the various aspects of the control problem and providing concrete implementations for the most commonly reused concepts. Users interested in low-level control concepts can, therefore, experiment with customizing the abstract interface classes to their own needs, or simply construct novel controllers using the concrete class implementations. Higher-level usage on the other hand, is easy to get started with, thanks to the server–client architecture. If the robot has been properly interfaced with the OCRA controller server, then clients can be developed with little effort and most of all, no deep understanding of the internals of the server side. Various examples of the different manners in which one can interact with OCRA are presented in the Supplemental Data Section and validate the variety of ways OCRA can be used to study and develop autonomy.

Ultimately, OCRA should serve as the basis for increasingly complex logic, by robustly resolving progressively more complex layers of the control problem. The server–client architecture is just the beginning of this process and should be built upon by even high-levels of problem reasoning, to create greater and greater levels of robot autonomy.

AUTHOR CONTRIBUTIONS

GE, RL, and AH contributed to the development and integration of the proposed software framework. VP laid out the conceptual foundations of the main algorithms in this software. GE, RL, AH, and VP contributed to the writing of the associated paper, JE being the main contributor to the writing.

ACKNOWLEDGMENTS

The authors wish to acknowledge the contribution of CEA-List for providing access to the ORC framework as well as to the engineers/researchers whose work has led to OCRA: Darwin Lau, Mingxin Liu, Joseph Salini, Hak Sovannara, and Silvio Traversaro.

FUNDING

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreements No. 600716 (CoDyCo). This work has also been partially sponsored by the French government research program Investissements d'Avenir through the Robotex Equipment of Excellence (ANR-10-EQPX-44).

ONLINE MATERIAL

Website: https://ocra-recipes.github.io/web/.

OCRA Documentation: https://ocra-recipes.github.io/web/doxy-ocra-recipes/html/index.html.

OCRA iCub Documentation: https://ocra-recipes.github.io/web/doxy-ocra-wbi-plugins/html/index.html.

OCRA Source Code: https://github.com/ocra-recipes/ocra-recipes.

OCRA iCub Source Code: https://github.com/ocra-recipes/ocra-wbi-plugins.

Related publications: https://ocra-recipes.github.io/web/authors/.

Tutorials: https://ocra-recipes.github.io/web/icub/2016/11/26/using-ocra-with-icub.html.

REFERENCES

Bouyarmane, K., and Kheddar, A. (2011). "Using a multi-objective controller to synthesize simulated humanoid robot motion with changing contact configurations," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2011* (San Francisco, CA: IEEE), 4414–4419. doi:10.1109/IROS.2011.6094483

Escande, A., Mansard, N., and Wieber, P.-B. (2014). Hierarchical quadratic programming: fast online humanoid-robot motion generation. *Int. J. Rob. Res.* 33, 1006–1028. doi:10.1177/0278364914521306

IEEE. (2009). *1016-2009 – IEEE Standard for Information Technology–Systems Design–Software Design Descriptions* (IEEE). doi:10.1109/IEEESTD.2009.5167255

Khatib, O. (1986). Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Rob. Res.* 5, 90–98. doi:10.1177/027836498600500106

Khatib, O. (1987). A unified approach for motion and force control of robot manipulators: the operational space formulation. *IEEE J. Rob. Autom.* 3, 43–53. doi:10.1109/JRA.1987.1087068

Liu, M., Tan, Y., and Padois, V. (2016). Generalized hierarchical control. *Auton. Robots* 40, 17–31. doi:10.1007/s10514-015-9436-1

Mansard, N., Stasse, O., Evrard, P., and Kheddar, A. (2009). "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: the stack of tasks," in *International Conference on Advanced Robotics, 2009. ICAR 2009* (Munich: IEEE), 1–6.

Mistry, M., Buchli, J., and Schaal, S. (2010). "Inverse dynamics control of floating base systems using orthogonal decomposition," in *IEEE International Conference on Robotics and Automation* (Anchorage, AK: IEEE), 3406–3412. doi:10.1109/ROBOT.2010.5509646

Nori, F., Traversaro, S., Eljaik, J., Romano, F., Del Prete, A., and Pucci, D. (2015). iCub whole-body control through force regulation on rigid non-coplanar contacts. *Front. Rob. AI.* 2:6. doi:10.3389/frobt.2015.00006

Padois, V. (2016). *Control and Design of Robots With Tasks and Constraints in Mind*. Paris, France: Hdr, Université Pierre et Marie Curie (Paris 6).

Padois, V., Fourquet, J.-Y., and Chiron, P. (2007). Kinematic and dynamic model-based control of wheeled mobile manipulators: a unified framework for reactive approaches. *Robotica* 25, 157–173. doi:10.1017/S0263574707003360

Peters, J., Mistry, M., Udwadia, F., Nakanishi, J., and Schaal, S. (2008). A unifying framework for robot control with redundant dofs. *Auton. Robots* 24, 1–12. doi:10.1007/s10514-007-9051-x

Roboptim. (2016). *C++ Library for Numerical Optimization for Robotics*. Available at: http://roboptim.net/

Rocchi, A., Hoffman, E. M., Caldwell, D. G., and Tsagarakis, N. G. (2015). "Opensot: a whole-body control library for the compliant humanoid robot

coman," in *IEEE International Conference on Robotics and Automation (ICRA), 2015* (Seattle, WA: IEEE), 1093–1099. doi:10.1109/ICRA.2015.7140076

Saab, L., Ramos, O. E., Keith, F., Mansard, N., Soueres, P., and Fourquet, J.-Y. (2013). Dynamic whole-body motion generation under rigid contacts and other unilateral constraints. *IEEE Trans. Robot.* 29, 346–362. doi:10.1109/TRO.2012.2234351

Salini, J. (2012). *Dynamic Control for the Task/Posture Coordination of Humanoids: Toward Synthesis of Complex Activities*. Theses, Paris: Université Pierre et Marie Curie – Paris VI.

Salini, J., Ivaldi, S., Hak, S., and Padois, V. (2013). *ISIR Controller in the XDE Framework for the Control of Robots Based on LQP Solvers*. Available at: http://chronos.isir.upmc.fr/salini/XDE-ISIRController/documentation/html/index.html

Salini, J., Padois, V., and Bidaud, P. (2011). "Synthesis of complex humanoid whole-body behavior: a focus on sequencing and tasks transitions," in *IEEE International Conference on Robotics and Automation (ICRA), 2011* (Shanghai: IEEE), 1283–1290. doi:10.1109/ICRA.2011.5980202

Sentis, L., and Khatib, O. (2005). "Control of free-floating humanoid robots through task prioritization," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation, 2005. ICRA 2005* (Barcelona: IEEE), 1718–1723. doi:10.1109/ROBOT.2005.1570361

Sentis, L., and Khatib, O. (2006). "A whole-body control framework for humanoids operating in human environments," in *Proceedings 2006 IEEE International Conference on*, *Robotics and Automation, 2006. ICRA 2006* (Orlando, FL: IEEE), 2641–2648. doi:10.1109/ROBOT.2006.1642100

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

frontiers in Robotics and AI

CODE
published: 09 April 2018
doi: 10.3389/frobt.2018.00040

Design and Implementation of a YARP Device Driver Interface: The Depth-Sensor Case

Alberto Cardellino, A. Ruzzenenti and L. Natale*

iCub Facility, Istituto Italiano di Tecnologia, Genoa, Italy

This work illustrates the design phases leading to the development of a new YARP device interface along with its client/server implementation. In order to obtain a smoother integration and a more reliable software usability, while avoiding common errors during the design phases, a new interface is created in the YARP network when a new family of devices is introduced.

Keywords: hardware abstraction, client server architecture, software design, depth sensor, YARP

OPEN ACCESS

Edited by:
Maxime Petit,
Imperial College London,
United Kingdom

Reviewed by:
Hyung Jin Chang,
Imperial College London,
United Kingdom
Nuno Ferreira Duarte,
Instituto Superior Técnico,
Universidade de Lisboa, Portugal

****Correspondence:***
Alberto Cardellino
alberto.cardellino@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *21 August 2017*
Accepted: *22 March 2018*
Published: *09 April 2018*

Citation:
Cardellino A, Ruzzenenti A and Natale L (2018) Design and Implementation of a YARP Device Driver Interface: The Depth-Sensor Case. Front. Robot. AI 5:40. doi: 10.3389/frobt.2018.00040

1. INTRODUCTION

Depth sensors, such as the kinect (Zhang, 2012; Han et al., 2013), are very popular in the field of navigation for mobile robots. OpenNI2 framework (Aksoy et al., 2011; Rehem Neto et al., 2013), an open source SDK used for the development of 3D sensing middleware libraries and applications, is arising as a tentative standard for this type of devices, yet producers do not always comply with the specifications. In a typical application, data are acquired by a robot but processed and visualized on a remote machine. The device driver is in charge of acquiring data from the sensor while client and server handles the transfer, optimizing both portability and performance. In general terms, we believe that an effective solution to standardize data flow in a software framework is to provide the device interface together with its client/server implementation.

1.1. YARP Device Interface

YARP (Metta et al., 2006) is a middleware specifically designed for robotics with a strong focus on modularity, code re-usage, flexibility, and hw/sw abstraction. In order to achieve those goals, the use of interfaces is fundamental because they allow to abstract from a specific producer. YARP device driver interfaces are the ones devoted to generalize the capabilities and configurations of a specific set of similar devices.

An interface is a class composed only by pure virtual function, data type definitions, and it is the place where relevant measurement unit must be declared. The implementation of a new YARP device interface is realized in the development of three C++ objects: (i) the device driver which handles the real hardware, (ii) the network server which publish the data, and (iii) the network client used by the application to remotely access the device. The objects are shown in the **Figure 1A**.

Note that by mean of an interface, the user application can connect directly to the device driver bypassing the client/server architecture, as illustrated in **Figure 1B**. This is useful when higher efficiency and low latency are required.

1.2. RGBD Device Family

An RGBD sensor is a device equipped with a standard RGB color camera and a depth image source. The latter is producing a special image in which each pixel is providing the distance of closest object in view.

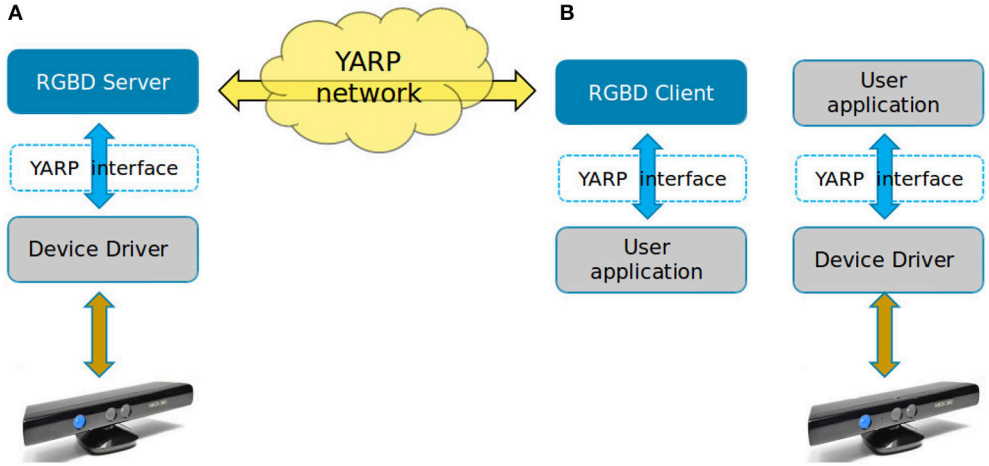


FIGURE 1 | How the application connects to an hardware device. **(A)** Via client/server. **(B)** Directly.

RGBD is the source data required to build a point cloud, but they have distinct characteristics. The depth sensor produces two separated image frames where the first one contains color component and the second one distance information. A point cloud instead is a specific data type where the *point* contains color and depth components altogether and optionally other related information like surface normals, curvatures, histogram, and so on.

While both RGB and depth frames shared the rectangular *width per height* structure, a point cloud is an unordered list of points of any size and shape. When dealing with this type of sensors, a number of information is required in order to correctly extract valuable data. Besides the image dimensions in terms of pixels and the frames themselves, other useful parameters are, the lens distortion model of RGB cameras and the measurement range and its accuracy for the depth sensor. The designing of the interface should thus include and provide all the previously mentioned data.

In this work, the concept of RGBD device is extended to include more cases than the physical sensor. All cases are shown in the **Figure 2**.

1.3. Common Design Patterns

The quickest approach for designing an Hardware Abstraction Layer (HAL) is proceeding bottom-up, starting from the hardware capabilities and generalizing them. This approach tends to fail when the device generates non-standard data or when the underlying hardware varies significantly from sample to sample.

On one hand, bottom-up generated interfaces are comprehensive of all device features whilst, on the other hand those interfaces tends to be too tailored on the first device they were built upon and difficult to be reused when the underlying assumptions change.

The other most followed lead is the top-down approach which is capable of providing a better abstraction, but usually it fails being comprehensive. In this case, low level details or configurations may be missing and users do not have access to all the required data.

2. DESIGN PROCESS

An example of well-structured software design is illustrated in the NEPOMUK project paper (Groza et al., 2007), described as an iterative process starting from the user's need, to design new code in order to seamlessly fit into an existing software environment. In order to overcome the before mentioned limitations, the design process has been widened to a bigger-picture, real use cases have been analyzed and generalized to extract relevant functionalities. The latter information has been employed to analyze the data flow and to design the resulting interface.

2.1. Identifying Data Flow and Device Capabilities

Typically, an RGBD device is capable of producing a color image and a depth image along with their respective parameters. The interfaces are thus required to describe and provide equivalent streams and parameters. YARP is often referred to as the robot information piping system, because one of its main functions is exchanging data between applications. Identifying the data to be shared and their properties helps designing better interfaces and client/server objects.

There are two main ways to exchange data in YARP, called streaming and RPC. All the information the device streams are sent to the client, whereas all the get/set requests originated by the client are RPCs. Data are sent trough the network via an object called *port*. A port is an abstraction of the operative system socket and it is able to send any type of data, different protocols can be used and multiple consumer can read from a single producer. We can identify the following desired data:

- Streaming
  - ⇒ The RGB image
  - ⇒ The depth image
- RPCs
  - ⇔ Info about streaming: e.g., how big is the image being published

- ⇔ Controls: e.g., increase the saturation/brightness and other camera parameters.
- ⇔ Info Visual: e.g., get the field of view of RGB/depth camera
- ⇔ Info about HW: e.g., this is a USB device

Each piece of information is required for both RGB and depth separately because they may differ in availability and values. Note that interfaces and data flow do not need to match. For example, a single interface may include both streaming and RPC data while a single RPC connection can handle requests from multiple interfaces.

2.2. Identifying Use Case Scenarios

The analysis result in four scenarios being comprehensive for all foreseen real world uses of a RGBD sensor, shown in **Figure 2**.

Note: Each sensor can be a real or simulated one, they will be handled in the same way.

Among existing applications, yarpview and camCalib are the most important ones the new device has to be compatible with. The first one is a GUI used to display images while the latter is used to compensate lens distortion.

2.3. Additional Constraints and Requirements

It is useful to explicit a few other characteristics the new interface and its implementation shall have in order to cope with the needs of an highly dynamic and innovative field like robotics.

2.3.1. Need of a Standard

Different devices may provide the same data using incompatible formats, for example the distance measure can be measured in meters, millimeter, or other units while the binary implementation can be an integer or a floating point number. Furthermore, a lens distortion model can be described using different set of parameters. In order for an high level application to run on different robots, it must be able to get all information at runtime and use them properly.

Relevant settings that are not available from OpenNI API have to be acquired from another source, for example distortion model can rarely be retrieved from the OpenNI API.

2.3.2. Unique Traits of RGBD Device

This work has to deal with the intrinsic complexity of a device composed by two different sensors with similar characteristics which may be unclear. A large amount of parameters are required to correctly identify the device properties.

2.3.3. Compliance With the YARP Ecosystem

This software is part of the YARP middleware, hence the new interface has to fit into existing code and to be as much intuitive as possible for both experienced and novice users. The RGB

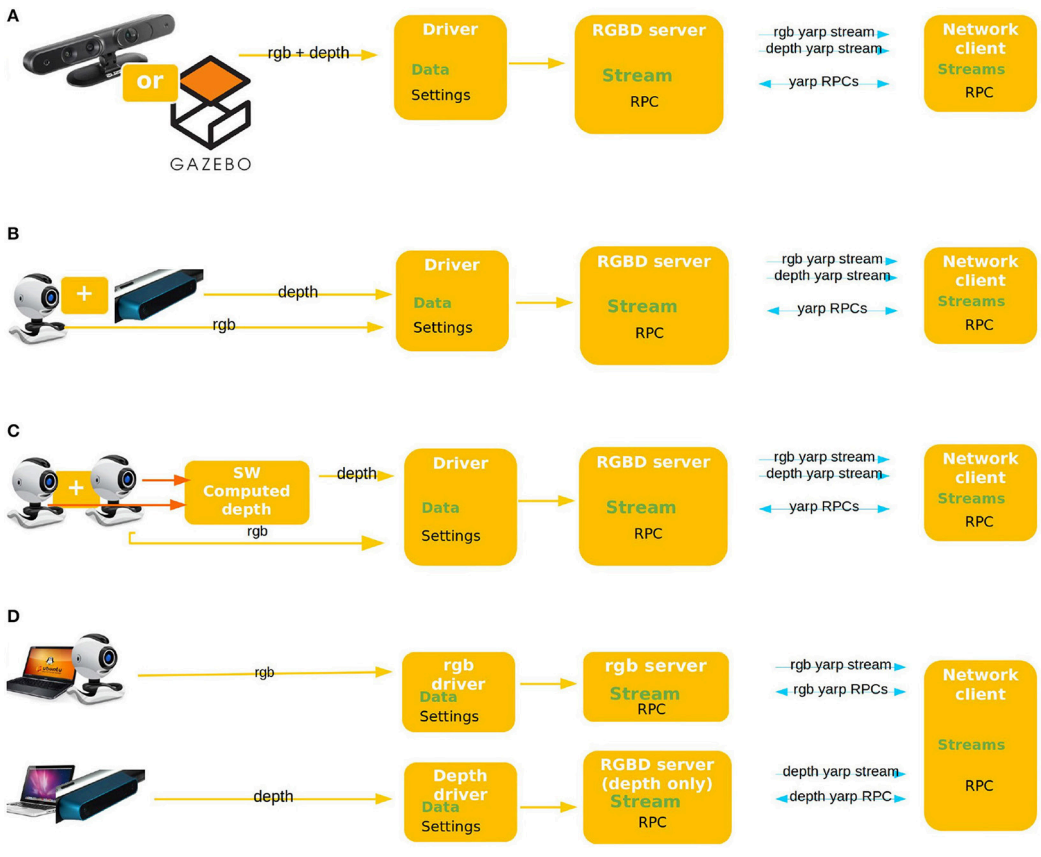


FIGURE 2 | Use case scenarios. **(A)** Proper RGBD sensor: MicroSoft Kinect, Asus XTION, or similar. **(B)** Two local yet separated sensors. **(C)** Stereo vision: two RGB sensor and a computed depth image. **(D)** Two separated sensors physically connected to two different machines.

sensor is by all means a standard camera and, as such, it provides options to configure color properties like saturation, brightness, exposure etc... The ability to change the camera parameters at runtime is widely used, so it must be accessible via the interface. The optimal solution is to allow any software currently working with standard camera to work also with RGB part of this device without any changes.

2.3.4. Modularity

YARP heavily leverages on modularity and code re-use, therefore the implementation of the depth sensor interface has to maximize these best practices. Furthermore, the client can read data from multiple sources while the server can broadcast them independently, as in use cases (**Figures 2B,D**).

2.3.5. Re-usability

Re-usability check has to be performed in two ways: first looking for compatible code to re-use into this project and second creating code that may be useful outside the scope of RGBD device for future use.

3. ADOPTED SOLUTIONS

The design process resulted in a series of design considerations and technical solutions adopted to best attain all the requirements. Those solutions can be divided into "abstract" design criteria and their relative "concrete" implementation.

All the requirements and solutions are summarized in the **Table 1**.

The resulting structure is general enough to cover all the use cases and flexible to allow both incremental implementation and update of existing software. The current state is already able to handle use case scenarios (**Figures 2A,C**) and can be easily extended to handle also cases (**Figures 2B,D**).

3.1. Design Criteria

3.1.1. Definition of a YARP Standard

YARP uses international measurement system for all units (except for angular degrees), therefore this convention has been enforced also in this interface where the unit for the depth measurement is set to be meters. The binary representations is the `float` to allow fraction of meters.

TABLE 1 | Requirements and proposed solutions for new YARP interface.

Requirement	Design criteria	Implementation solutions
Need of a standard	Definition of a YARP standard	API compensation
Compatibility with the YARP ecosystem	Re-use, not inherit	Separated data flow
Modularity	Isolation of capabilities	Separated data flows
Re-usability	Isolation of capabilities	Three levels decoupling
Unique traits of this device type	Isolation of capabilities	Capabilities composition

3.1.2. Re-use, Not Inherit

The interface `IFrameGrabberControls2` is an already existing YARP interface describing how to set RGB color sensor properties as saturation, brightness, exposure etc... A possible way to include these functionalities in the new interface would be to inherit from it, but this has some implications. The new interface will be tightly coupled to previous code and the maintenance will be more difficult. Any change to `IFrameGrabberControls2` will be propagated to the new interface and all devices using it. On the other hand, adding the same methods also in the new interface will generate duplicated code and confusion.

The best approach is to keep separated the two functionalities and have the device implementation to use them where required.

3.1.3. Isolation of Capabilities

Instead of defining an single interface covering all the device functionalities or data types, the best solution is to define an interface for each capability and then combine them into a bigger one where appropriate. This way each interface is smaller and cleaner, but most importantly each single interface can be re-used more easily in different contexts. New interfaces created for this device are the ones required to fill the gap between what's existing and what is required. They have been created separately for RGB and depth part of the device. A snippet of code is shown below.

```
class yarp::dev::IRgbVisualParams
{
   int  getRgbHeight();
   int  getRgbWidth();
   bool getRgbConfigurations(Vector<Config> &c);
   bool getRgbResolution(int &width, int &height);
   bool setRgbResolution(int width, int height);
   bool getRgbFOV(double &hFov, double &vFov);
   bool setRgbFOV(double hFov, double vFov);
   bool getRgbIntrinsicParam(Property &param);
   bool getRgbMirroring(bool &mirror);
   bool setRgbMirroring(bool mirror);
}
```

```
class yarp::dev::IDepthVisualParams
{
   int   getDepthHeight();
   int   getDepthWidth();
   bool  setDepthResolution(int width,int height);
   bool  getDepthFOV(double& hFov, double& vFov);
   bool  setDepthFOV(double hFov, double vFov);
   bool  getDepthIntrinsicParam(Property& param);
   double getDepthAccuracy();
   bool  setDepthAccuracy(double accuracy);
   bool  getDepthClipPlanes(double &near, &far);
   bool  setDepthClipPlanes(double near, far);
   bool  getDepthMirroring(bool& mirror);
   bool  setDepthMirroring(bool mirror);
}
```

Interface snapshot 1. Example of interface methods.
Documentation: http://www.yarp.it/classyarp_1_1dev_1_1IRgbVisualParams.html
http://www.yarp.it/classyarp_1_1dev_1_1IDepthVisualParams.html
Git repository: https://github.com/robotology/yarp/tree/master/src/libYARP_dev/include/yarp/dev

The two interfaces created are similar because the sensors have similar features, but each method has the RGB/Depth prefix to clearly state which sensor it is working with. This helps novice users to understand what the function is supposed to do and name clash between two sensors is avoided. There are some differences however due to the sensors nature, for example in the depth interface there are getter and setter methods for Accuracy and clip planes which has no meaning for a standard RGB camera.

3.2. Implementation Solutions

3.2.1. API Compensation

The information requested to be available are more then what's usually covered by the OpenNI2 API, hence another source of information is needed. This has been achieved by mean of a configuration file, subdivided in three main sections:

- General parameters: describe which device the YARP factory shall create and how to manage it.
- Settings: these parameters describe the user's desired initial configuration of the device. These values will be set at startup and if anyone fails, the device must be closed providing an error. All the settings are also available for remote control with getter/setter methods, therefore the configuration can be verified and changed remotely by the user's application at any time.
- Hardware description: the listed parameters are read only . Everything not available through device API can be listed here. These values will be available to remote applications via getter methods, but they cannot be set. This is also useful in case the device returns wrong values; the data from configuration file will be returned to the user instead.

3.2.2. Separated Data Flow

Defining how many sockets to create, the protocol to use etc... is a trade-off between optimization of resources and granularity of information. The more complex/custom the data is, the less application will be compatible with. On the other hand, creating many sockets to send small pieces of information is a waste of resources.

The choice implemented is to create two separated streams for RGB and depth images, to be back compatible with existing application using color images only. All the RPC requests instead can be handled by a single YARP port. There is no need in fact for the client to know all the server's capabilities. A client can implement only the subset of RPC it requires, therefore a existing client can freely work with a newer server using an extended set of messages.

Only the 4th use case scenario will require the client to have two separated RPC ports, as it requires to connect to two different servers to collect all the required informations.

3.2.3. Three Levels Decoupling

Network messages, client/server implementation and hardware device are separated between each-others. Usually when building a client/server pair in YARP there are two levels of decoupling: the first one is the YARP message which decouples the server from the remote client. The second one is the interface itself which decouples the server from the device driver and the user application from the network client.

The implementation of an interface in the server/client requires to write code devote to generate the YARP message and parse it in order to provide the service and generate proper response. Historically this job was always been implemented by the client/server classes themselves, but this may lead to duplicated code when more servers or clients uses the same interface. Therefore a new decoupling level has been introduced by implementing all the YARP message parsing into a specific class for each interface, the client/server will then use these classes to handle network communication.

This way, should a new server implement this interface, adding the message parsing will require only three lines of code:

1) Add interface inheritance

```
Server : public NewInterface
```

2) Instantiate a parser class

```
yarp::dev::Implement_Interface_Parser  rgbParser;
```

3) Configure the parser by giving access to the class

```
rgbParser.configure(NewInterfacePointer);
```

Interface snapshot 2. Example of usage, server side.

3.2.4. Capabilities Composition

Leveraging on the previously shown ideas "Isolation of Capabilities" and "Three Levels Decoupling," it follows that a device can incrementally add capabilities by inheriting from required interfaces and parsers. The whole RGBD interface will be the sum of RgbVisualParams and DepthVisualParams, plus the specific information which have meaning only when both sensors are available together.

```
class yarp::dev::IRGBDSensor : public
    IRgbVisualParams
public IDepthVisualParams
{
    bool getExtrinsicParam(Matrix &extrinsic) ;
        string getLastErrorMsg(Stamp *timeStamp);
    bool getRgbImage(FlexImage &rgbImage, Stamp
    *timeStamp);
    bool getDepthImage(ImageOf<PixelFloat>
    &depthImage, Stamp  *timeStamp);
    bool getImages(FlexImage &colorFrame, ImageOf
    <PixelFloat> &depthFrame, Stamp *colorStamp,
         Stamp *depthStamp);
    RGBDSensor_status getSensorStatus();
}
```

Interface snapshot 3. Extending capabilities by merging two interfaces into a bigger one.
Documentation: http://www.yarp.it/classyarp_1_1dev_1_1IRGBDSensor.html
Git repository: https://github.com/robotology/yarp/blob/master/src/libYARP_dev/include/yarp/dev/IRGBDSensor.h

Each interface contains methods to get the sensor intrinsic parameters, and since the RGBD interface includes the two sensors together, a method to get extrinsic parameters is included. The client/server for this device will create its own message

sender/parser by extending the ones implemented for each single interface as explained in "Three Levels Decoupling" section. Furthermore, previous RGB-only image server has been easily extended to implement also the RgbVisualParams interface by adding the parser.

4. CONCLUSION AND FUTURE WORK

The design process successfully generated a set of interfaces both flexible and comprehensive to handle all use cases identified and satisfy all additional requirements. The interface and C++ objects shown in this work have been used with three models of depth sensors from two different producers and with the simulated device available within Gazebo. The new server is well-integrated in the YARP framework, compatibility with existing applications has been achieved and former device drivers specific for RGB-only cameras have been extended to implement new functionality, hence user application can benefit from additional information.

The dataset acquisition pipeline shown in Pasquale et al. (2016) and used in Maiettini et al. (2017) was developed for the iCub robot using images acquired from stereo vision system and then, using the interfaces resulting from the work presented, the pipeline was easily integrated on the R1 robot, that mounts a RGBD sensor.

The implementation will be extended to cover also use scenarios (**Figures 2B,D**), also a synchronization mechanism for the two image streaming will be integrated in the client. The code can be verified using YARP test utilities or using simple example code. Instruction how to run tests are in the in the following github repository https://github.com/robotology/yarp/tree/master/example/dev/RGBD/README.md.

AUTHOR CONTRIBUTIONS

AC: main contributor, designed the interfaces, and client/server implementation; AR: contributed refining the interfaces, device driver implementation, testing; LN: supervisor.

REFERENCES

Aksoy, E. E., Abramov, A., Dörr, J., Ning, K., Dellen, B., and Würgötter, F. (2011). Learning the semantics of object-action relations by observation. *Int. J. Robot. Res.* 30, 1229–1249. doi: 10.1177/0278364911410459

Groza, T., Handschuh, S., Moeller, K., Grimnes, G., Sauermann, L., Minack, E., et al. (2007). "The nepomuk project–on the way to the social semantic desktop," in *Proceedings of I-Semantics' 07*, eds T. Pellegrini and S. Schaffert (Graz), 201–211.

Han, J., Shao, L., Xu, D., and Shotton, J. (2013). Enhanced computer vision with microsoft kinect sensor: a review. *IEEE Trans. Cybern.* 43, 1318–1334. doi: 10.1109/TCYB.2013.2265378

Maiettini, E., Pasquale, G., Rosasco, L., and Natale, L. (2017). "Interactive data collection for deep learning object detectors on humanoid robots," in *17th IEEE-RAS International Conference on Humanoid Robotics, Humanoids 2017* (Birmingham), 862–868.

Metta, G., Fitzpatrick, P., and Natale, L. (2006). Yarp: yet another robot platform. *Int. J. Adv. Robot. Syst.* 3:8. doi: 10.5772/5761

Pasquale, G., Mar, T., Ciliberto, C., Rosasco, L., and Natale, L. (2016). Enabling depth-driven visual attention on the iCub humanoid robot: instructions for use and new perspectives. *Front. Robot. AI* 3:35. doi: 10.3389/frobt.2016.00035

Rehem Neto, A. N., Saibel Santos, C. A., and de Carvalho, L. A. A. (2013). "Touch the air: an event-driven framework for interactive environments," in *Proceedings of the 19th Brazilian Symposium on Multimedia and the Web* (New York, NY: ACM), 73–80.

Zhang, Z. (2012). Microsoft kinect sensor and its effect. *IEEE MultiMedia* 19, 4–10. doi: 10.1109/MMUL.2012.24

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer HC, and handling Editor declared their shared affiliation.

frontiers in Robotics and AI

CODE
published: 12 June 2018
doi: 10.3389/frobt.2018.00046

Markerless Eye-Hand Kinematic Calibration on the iCub Humanoid Robot

Pedro Vicente [1], Lorenzo Jamone [1,2] and Alexandre Bernardino [1]*

[1] Institute for Systems and Robotics, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, [2] ARQ (Advanced Robotics at Queen Mary), School of Electronic Engineering and Computer Science, Queen Mary University of London, London, United Kingdom

Humanoid robots are resourceful platforms and can be used in diverse application scenarios. However, their high number of degrees of freedom (*i.e.*, moving arms, head and eyes) deteriorates the precision of eye-hand coordination. A good kinematic calibration is often difficult to achieve, due to several factors, *e.g.*, unmodeled deformations of the structure or backlash in the actuators. This is particularly challenging for very complex robots such as the iCub humanoid robot, which has 12 degrees of freedom and cable-driven actuation in the serial chain from the eyes to the hand. The exploitation of real-time robot sensing is of paramount importance to increase the accuracy of the coordination, for example, to realize precise grasping and manipulation tasks. In this code paper, we propose an online and markerless solution to the eye-hand kinematic calibration of the iCub humanoid robot. We have implemented a sequential Monte Carlo algorithm estimating kinematic calibration parameters (joint offsets) which improve the eye-hand coordination based on the proprioception and vision sensing of the robot. We have shown the usefulness of the developed code and its accuracy on simulation and real-world scenarios. The code is written in C++ and CUDA, where we exploit the GPU to increase the speed of the method. The code is made available online along with a Dataset for testing purposes.

Keywords: code:C++, humanoid robot, markerless, hand pose estimation, sequential monte carlo parameter estimation, kinematic calibration

OPEN ACCESS

Edited by:
Giorgio Metta, Fondazione Istituto Italiano di Technologia, Italy

Reviewed by:
Claudio Fantacci, Fondazione Istituto Italiano di Technologia, Italy
Hyung Jin Chang, Imperial College London, United Kingdom

****Correspondence:***
Pedro Vicente
pvicente@isr.tecnico.ulisboa.pt

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *21 August 2017*
Accepted: *06 April 2018*
Published: *12 June 2018*

Citation:
Vicente P, Jamone L and Bernardino A (2018) Markerless Eye-Hand Kinematic Calibration on the iCub Humanoid Robot. Front. Robot. AI 5:46. doi: 10.3389/frobt.2018.00046

1. INTRODUCTION AND RELATED WORK

An intelligent and autonomous robot must be robust to errors on its perceptual and motor systems to reach and grasp an object with great accuracy. The classical solution adopted by industrial robots rely on a precise calibration of the mechanics and sensing systems in controlled environments, where sub-millimeter accuracy can be achieved. However, a new emerging market is targeting consumer robots for collaboration with humans in more general scenarios. These robots cannot achieve high degrees of mechanical accuracy, due to (1) the use of lighter and flexible materials, compliant controllers for safe human-robot interaction, and (2) lower sensing precision due to varying environmental conditions. Indeed, humanoid robots, with complex kinematic chains, are among the most difficult platforms to calibrate and model properly with the precision required to reach and/or grasp objects. A small error in the beginning of the kinematic chain can generate a huge mismatch between the target location (usually coming from vision sensing) and the actual 6D end-effector pose.

Eye-hand calibration is a common problem in robotic systems that several authors tried to solve exploiting vision sensing [e.g., Gratal et al. (2011); Fanello et al. (2014); Garcia Cifuentes et al. (2017); Fantacci et al. (2017)][1].

2. PROPOSED SOLUTION

In this code paper, we propose a markerless hand pose estimation software for the iCub humanoid robot [Metta et al. (2010)] along with an eye-hand kinematic calibration. We exploit the 3D CAD model of the robot embedded in a game engine, which works as the robot's internal model. This tool is used to generate multiple hypotheses of the hand pose and compare them with the real visual perception. By using the information extracted from the robot motor encoders, we generate hypotheses of the hand pose and its appearance in the cameras, that are combined with the actual appearance of the hand in the real images, using particle filtering, a sequential Monte Carlo method. The best hypothesis of the 6D hand pose is used to estimate the corrective terms (joint offsets) to update the robot kinematic model. The visual based estimation of the hand pose is used as an input, together with the proprioception, to continuously calibrate (*i.e.*, update) the robot internal model. At the same time, the internal model is used to provide better hypotheses for the hand position in the camera images, therefore enhancing the robot perception. The two processes help each other, and the final outcome is that we can keep the internal model calibrated and obtain a good estimation of the hand pose, without using specialized visual markers on the hand.

The original research work [Vicente et al. (2016a) and Vicente et al. (2016b)] contains: (1) a complete motivation from the developmental psychology point of view and theoretical details of the estimation process, and (2) technical details on the interoperability between the several libraries and the GPGPU approach for an increased boost on the method speed, respectively.

The present manuscript is a companion and complementary code paper of the method presented in Vicente et al. (2016a). We will not describe with full details the theoretical perspective of our work, instead we will focus on the resulting software system connecting the code with the solution proposed in Vicente et al., 2016b. Moreover, the objective of this publication is to give a hands-on perspective on the implemented software which could be used and extended by the research community.

The source code is available at ***the official GitHub code repository***:

`https://github.com/vicentepedro/Online-Body-Schema-Adaptation`

and the documentation on the ***Online Documentation page***:

`http://vicentepedro.github.com/Online-Body-Schema-Adaptation`

We use a Sequential Monte Carlo parameter estimation method to estimate the calibration error β in the 7D robot's joint space corresponding to the kinematic chain going from each eye to the end-effector. Let us consider:

$$\theta = \theta^r + \beta \tag{1}$$

where θ^r are the real angles; θ are the measured angles; β are joint offsets representing calibration errors. Given an estimate of the joint offsets ($\hat{\boldsymbol{\beta}}$), a better end-effector's pose can be retrieved using the forward kinematics.

One of the proposed solutions for using Sequential Monte Carlo methods for parameter estimation[2] (*i.e.*, the parameters β in our problem), is to introduce an artificial dynamics, changing from a static transition model $\left(\beta_t = \beta_{t-1}\right)$ to a slowly time-varying one:

$$\beta_t = \beta_{t-1} + \mathrm{w}_t \tag{2}$$

where w_t is an artificial dynamic noise that decreases when t increases.

3. SOFTWARE DESIGN AND ARCHITECTURE PRINCIPLES

The software design and architecture for implementing the eye-hand kinematic calibration solution has the following requirements: (1) the software should be able to run in real-time since the objective is to calibrate the robot during a normal operating behaviour, and (2) it should be possible to run the algorithm in a distributed way, *i.e.*, run parts of the algorithm in several computers in order to increase computation power.

The authors decided to implement the code in `C++` in order to cope with the real-time constraint, and to exploit the YARP middleware [Metta et al. (2006)] to distribute the components of the algorithm in more than one machine.

The source code for these modules are available at ***the official GitHub code repository*** (check section 2).

The code is divided into three logical components: (1) the hand pose estimation (section 4.1), (2) the Robot's Internal Model generator (section 4.2), and (3) the likelihood assessment (section 4.3), which are implemented, respectively, at the following repository locations:

- `modules/handPoseEstimation`
  - `include/handPoseEstimationModule.h`
  - `src/handPoseEstimationMain.cpp`
  - `src/handPoseEstimationModule.cpp`
- `modules/internalmodel`

[1]For a more detailed review of the state of the art, please check the article Vicente et al. (2016a)

[2]See Kantas et al. (2009) for other solutions

- `icub-internalmodel-rightA-cam-Lisbon.exe`
- `icub-internalmodel-leftA-cam-Lisbon.exe`

- `modules/likelihodAssessment`
  - `src/Cuda_Gl.cu`
  - `src/likelihood.cpp`

The software architecture implementing the proposed eye-hand calibration solution can be seen in **Figure 1**. The first component - Hand Pose Estimation - is responsible for proposing multiple hypotheses according to the posterior distribution. We use a Sequential Monte Carlo parameter estimation method in our work [check Vicente et al. (2016a) Section 3.3 for further theoretical details]. The definitions of the functions presented in the architecture (**Figure 1**) can be found in the `.cpp` and `.h` files and will be explained in detail in Section 4.1. The Hand Pose Estimation is OS independent and can run in any computer with the YARP library installed.

The second component - Robot's Internal Model - generates hypotheses of the hand pose based on the 3D CAD model of the robot and was build using the game engine Unity®. There are two versions of the internal model on the repository. One for the right-hand (`rightA`) and another one for the left-hand (`leftA`). Our approach was to divide the two internal models since we have separated calibration parameters for the head-left-arm and for the head-right-hand kinematic chains. The Unity platform was chosen to develop the internal model of the robot since it is able to generate a high number of *frames per second* on the GPU even for complex graphics models. The scripting component of the Unity game engine was programmed in `C#`. The bindings of YARP for `C#` were used in order to facilitate the internal model generator to communicate with the other components of the system. This component is OS-dependent and only runs on Windows and the build version available on the repository does not require a paid license of Unity Pro.

Finally, the likelihood assessment is called inside the Robot's Internal Model as a Dynamic Link Library and exploits GPGPU programming to compare the real perception with the multiple generated hypotheses. The GPGPU programming, using the CUDA library [Nickolls et al. (2008)], allows the algorithm to run in quasi-real-time. The `.cpp` file contains the likelihood computation method, and the `.cu` the GPGPU program.

Our eye-hand calibration solution exploits vision sensing to reduce the error between the perception and the simulated hypotheses, the OpenCV library [Bradski (2000)] with CUDA enabled capabilities [Nickolls et al. (2008)] was chosen to exploit computer vision algorithms and run them in real-time.

The interoperability between the OpenCV, CUDA and OpenGL libraries was studied in Vicente et al. (2016b). In the particular case of the iCub humanoid robot [Metta et al. (2010)], and to suit within the YARP and iCub architectures, we encapsulated part of the code in an RFModule[3] class structure and use YARP buffered ports[4] and RPC services[5] for communications and user interface (Check section 5.2.3). The hand pose estimation module allows the user to send requests to the algorithm which follows an event-driven architecture: where for each new incoming information from the robot (cameras and encoders) a new iteration of the Sequential Monte Carlo parameter estimation is performed.

4. CODE DESCRIPTION

4.1. Hand Pose Estimation Module

4.1.1. Initializing the Sequential Monte Carlo parameter estimation - initSMC Function

In the function *initSMC* we initialize the variables of the Sequential Monte Carlo parameter estimation, *i.e.*, the initial distribution $p(\beta_0)$ [Eq. (10) in Vicente et al. (2016a)], and the initial artificial dynamic noise. The **Listings 1** contains the *initSMC* function where some of the variables (in red) are parametrized at initialization time (check sub-section 5.2.1 for more details on the initialization parameters). We use a random seed generated according with the current time and initialize each angular offset with a Gaussian distribution: N(initialMean; initialStdDev).

4.1.2 Read Image, Read Encoders, ProcessImages and SendData

The left and right images along with the head and arm encoders are read at the same time to ensure consistency between the several sensors.

The reading and processing procedure of the images are defined inside the function:

`handPoseEstimationModule::updateModule()`

that can be found on the file:

`src/handPoseEstimationModule.cpp.`

The function process Images (see **Listings 2**) applies a Canny edge detector and a distance transform to both images separately. Moreover, the left and the right processed images are merged, *i.e.*, concatenated horizontally, in order to be compared to the generated hypotheses inside the Robot's internal model.

The Hand pose estimation module sends: (1) the pre-processed images, (2) the head encoders and (3) the arm encoders (θ) along with the offsets (β) to the Robot's internal model[6]. This procedure is defined inside the function:

`handPoseEstimationModule::runSMCIteration()`

4.1.3. Update Likelihood

The Hand Pose Estimation module receives the likelihood vector from the Robot's internal model and updates the likelihood value for each particle on the for-loop at line:

`handPoseEstimationModule.cpp#L225`

[3]http://www.yarp.it/classyarp_1_1os_1_1RFModule.html
[4]http://www.yarp.it/classyarp_1_1os_1_1BufferedPort.html
[5]http://www.yarp.it/classyarp_1_1os_1_1RpcServer.html

[6]See Eq. 1 and handPoseEstimationModule.cpp#L214

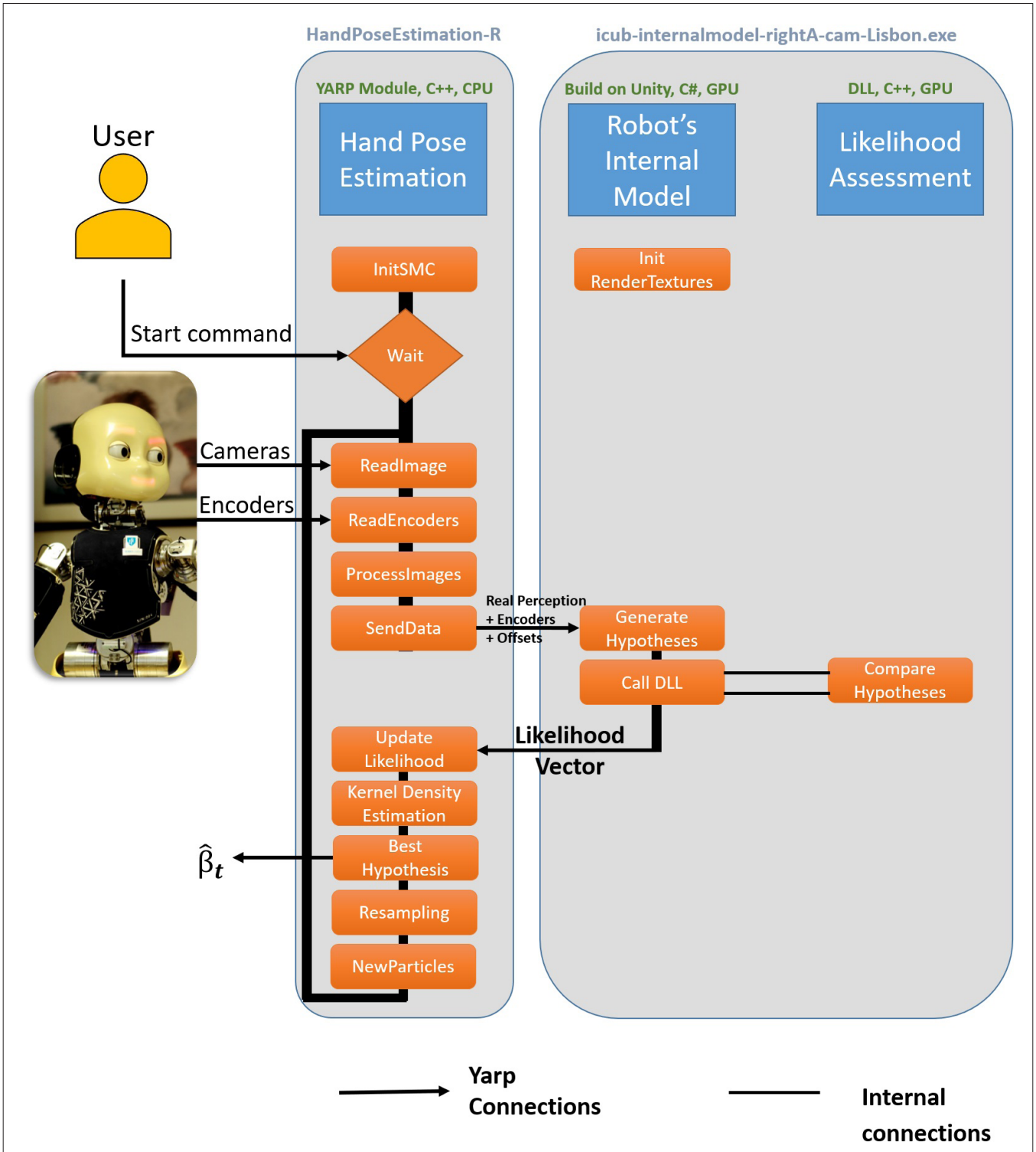


FIGURE 1 | Architecture of the software. The hand pose estimation component (*handPoseEstimation*) initiates the Sequential Monte Carlo parameter estimation method (*initSMC*) and waits for a start command from the user. The perception and proprioception (cameras and encoders) of the robot are received and the parameter estimation starts. The real image and the particles are sent (*sendData*) to the Robot's internal Model (*icub-internalmodel-rightA-cam-Lisbon.exe* or *icub-internalmodel-leftA-cam-Lisbon.exe*) in order to generate the hypotheses. The likelihood assessment of each hypothesis is calculated using a Dynamic Link Library (DLL) file inside the Robot's internal model. The likelihood of each particle is saved and a Kernel Density estimation is performed to calculate the best calibration parameters. The Resampling step is performed and a new set of particles are saved for the next iteration of the Sequential Monte Carlo parameters estimation.

Listing 1 | HandPoseEstimationModule::initSMC Function. Defined in handPoseEstimationModule.cpp

```
1. bool handPoseEstimationModule :: initSMC ( )
2.  {
4.     // Generate random particles
5.     srand((unsigned int)time(0)); // make sure random numbers are really
random.
6.     rngState = cvRNG(rand());
7.     // initialize Beta1
8.     cvRandArr(&rngState, particles 1, CV_RAND_NORMAL,
cvScalar(initialMean), cvScalar(initialStdDev));
9.     … … // similar for particles2 to particles6
10.     cvRandArr (&rngState, particles7, CV_RAND_NORMAL,
cvScalar(initialMean) , cvScalar(initialStdDev));
11.     // Artificial Noise Initialization
12.     artifNoiseStdDev = initialArtificialNoiseStdDev;
13.  }
```

4.1.4. Kernel Density Estimation

Although the state is represented at each time step as a distribution approximated by the weighted particles, our best guess for the angular offsets can be computed using a Kernel Density Estimation (KDE) to smooth the weight of the particles according to the information of neighbor particles, and choose the particle with the highest smoothed weight ($\omega'^{[i]}$) as our state estimate [Section 3.5 of Vicente et al. (2016a)].

The implementation of the KDE with a Gaussian kernel can be seen in **Listings 3**. The double for-loop implements the KDE accessing each particle (*iParticle*) and computing the influence of each neighbor (*mParticle*) according to the relative distance in the 7D-space between the two particles and the likelihood of the neighbor *[cvmGet (particles, 7,mParticle)]*. The parameters that can be fine-tuned are highlighted in red.

4.1.5. Best Hypothesis

The best hypothesis, computed using the KDE, is sent through a YARP buffered port from the module after *N* iterations. The port has the following name:

`/hpe/bestOffsets:o`

The parameter *N* (the number of elapsed iterations before sending the estimated angular offsets) can be changed by the user at initialization using the `minIteration` parameter (check Section 5.2.1 for more details) and the objective is to ensure the filter convergence before using the estimate (*e.g.*, to control the robot). This is an important parameter since in the initial stages the estimation can jump a lot from an iteration to the next one (before converging to a more stable solution).

Listing 2 | HandPoseEstimationModule::processImages. Defined in handPoseEstimationModule.cpp

```
1. Mat handPoseEstimationModule :: processImages (Mat inputImage)
2.  {
3.     Mat edges , dt Image;
4.     cvtColor(inputImage, edges, CV_RGB2GRAY);
5.     // Blur Image
6.     blur(edges, edges, Size (3, 3));
7.     Canny(edges, edges, 65, 3*65,3);
8.     threshold(edges, edges, 100,255,THRESH_BINARY_INV); // binary Image
9.     distanceTransform(edges, dt Image, CV_DIST_L2, CV_DIST_MASK_5);
10.     return dtImage;
11.  }
```

Listing 3 | Kernel Density Estimation with Multivariate Normal Distribution Kernel: modules/handPoseEstimation/src/handPoseEstimationModule.cpp

```
1. void handPoseEstimationModule :: kernelDensityEstimation ( )
2.{
3.   // Particle i
4.   double maxWeight = 0.0;
5.   for (int iParticle = 0; iParticle <n Particles; iParticle ++)
6.   {
7.     double sum1 = 0.0;
8.     // Particle m
9.      for (int mParticle = 0; mParticle <nParticles; mParticle++)
10.     {
11.        double sum2 = 0.0;
12.        if ( (float) cvmGet (particles, 7, mParticle) > 0 )
13.        {
14.           // Beta 0.. to..6
15.           for (int joint = 0; joint <7; joint ++)
16.           {
17.              // || pi–pj ||^2 / KDEStdDev ^2
18.              sum2 += pow( ((float) cvmGet (particles, joint, mParticle)–
(float)  cvmGet (particles, joint, iParticle )) , 2) / pow(KDEStdDev, 2);
//  Multivariate normal distribution
19.           }
20.           sum1 += s t d :: exp(–sum2/( 2) ) *cvmGet (particles, 7 ,  mParticle);
21.        }
22.     }
23.     sum1 = sum1 / ( nParticles*sqrt (pow(2*M_PI, 1) *pow(KDEStdDev,  7) ) );
24.     double weight = alphaKDE*sum1 + cvmGet (particles, 7 ,  iParticle);
25.     if (weight>maxWeight)
26.     {
27.        maxWeightIndex= iParticle; // save the best particle index
28.     }
29.  }
30.}
```

4.1.6. Update Artificial Noise, Resampling and New Particles

The artificial noise is updated according to the maximum likelihood criteria. See the pseudo-code on **Listings 4**, which corresponds to line 230 to 254 in the file:

`src/handPoseEstimationModule.cpp`

We update the artificial noise according to the maximum likelihood, *i.e.*, if the maximum likelihood is below a certain threshold (*minimumLikelihood*), we do not perform the resampling step and we increase the artificial noise. On the other hand, if the maximum likelihood is greater than the threshold we apply the resampling and decrease the artificial noise. The objective is to prevent the particles to become trapped in a "local maximum" since the current best solution is not worthy of resampling the particles. Indeed, this approach will force them to explore the state space.

The trade-off between exploration and exploitation is measured according to the maximum likelihood in each time step of the algorithm. The idea is to exploit the low number of particles in a clever way. Moreover, the upper and lower bound ensure, respectively, that: (1) the noise will not

Listing 4 | Pseudo Code updating artificial noise corresponding to part of the function runSMCIteration() within file: src/handPoseEstimationModule.cpp

```
1. IN handPoseEstimationModule :: runSMCIteration ( )
2.{
3.   …
4.   // Resampling or not Resampling. That's the Question
5.   if (maxLikelihood >minimumLikelihood) {
6.      systematic_resampling ( ); // Check Section Resampling and New Particles
7.      reduceArtificialNoise ( );
8.   }
9.   else { // do not apply resampling stage
10.      increaseArtificialNoise ( );
11.   }
12.   if (artifNoiseStdDev > upperBoundNoise) { // upperbound of artificial noise
13.      artifNoiseStdDev = upperBoundNoise;
14.   }
15.   if (artifNoiseStdDev < lowerBoundNoise) { // lowerbound of artificial noise
16.      artifNoiseStdDev = lowerBoundNoise;
17.   }
18.   addNoiseToEachSample ()
19.}
```

Listing 5 | Pseudo-Code Robot's internal model.

```
1. InitRenderTextures ( ) // Initialization of the strutures to receive
2.
3.for (each iteration) // for each iteration of the SMC
4.{
5.   waitForInput ( ); // wait for input vector with particles to be generated
6.
7.   for (each particle) {
8.      moveTheInternalModel ( ) // Change the robot's configuration
9.      RenderAllucinatedImages ( ); // render left and right image on a render
texture
10.      nextFrame ( );
11.   }
12.   // After 200 frames call DLL function
13.   ComputeLikelihood (AllucinatedImages (200), RealImage) // Call the DLL
function (CudaEdgeLikelihood) to compare the hypotheses with the real image.
14.}
```

increase asymptotically and the samples will be spread over the 7D state-space and (2) the particles will not end-up all at the same value, which can happen when the random noise is Zero.

On the resampling stage, we use the systematic resampling strategy [check Hol et al. (2006)], which ensures that a particle with a weight greater than $1/M$ is always resampled, where M is the number of particles.

4.2. Robot's Internal Model Generator

The **Listings 5** shows the general architecture of the Robot's Internal Model Generator using pseudo-code.

4.2.1. Initialization of the Render Textures

The render textures, which will be used to render the two camera images, are initialized for each particle for both left and right views of the scene.

4.2.2. Generate Hypotheses

The hypotheses are generated on a frame-based approach, *i.e.*, we generate one hypothesis for each frame of the "game". After we receive the vector with the 200 hypotheses to generate, we virtually move the robot to each of the configurations to be tested and record both images (left and right) in a renderTexture.

After the 200 generations, we call the likelihood assessment DLL function to perform the comparison between the real images and the generated hypotheses.

The available version of the Robot's internal model generator is an executable compiled and self-contained which works on Windows-based computers with the installed dependencies[7]. Moreover, this does not require neither the Unity® Editor to be installed in the computer nor the Unity Pro license.

More details on the creation of the Unity® iCub Simulator for this project can be found in Vicente et al. (2016b) Sec. 5.2 - "The Unity® iCub Simulator".

4.3. Likelihood Assessment Module

The likelihood assessment is based on the observation model defined in Vicente et al. (2016a) Section 3.4.2.

We exploit an edge-based extraction approach along with a distance transform algorithm computing the likelihood using the Chamfer matching distance [Borgefors and Bradski (1986)].

In our code, these quantities are computed in the GPU using the OpenCV and CUDA libraries, and the interoperability between these libraries and the OpenGL library. The solution adopted was to add the likelihood assessment as a `cpp` plugin called inside the internal model generator module. The likelihood.cpp file, particularly the function *CudaEdgeLikelihood*, is where the likelihood of each sample is computed. Part of the code of the likelihood function is shown and analysed in **Listings 6**. Up to the line 21 of the Listings 6, we exploit the interoperability between the libraries used (OpenGL, CUDA, OpenCV) and after line 21 we apply our likelihood metric using the functionality of the OpenCV library, where *GgpuMat* is the generated Image of the *ith* sample and GgpuMat_R is the real Distance Transform image. In line 35, the lambdaEdge is a parameter to tune the distance metric sensitivity, which is initialized at the value 25 in line 1 (corresponding to line 148 of the `C++` file)[8]. When the generated image does not have edges (*i.e.*, the hand is not visible by the cameras), we force the likelihood of this particle to be almost zero (line 37 and 39, respectively). The maximum likelihood (*i.e.*, the value 1.0) is achieved when each entry of

[7]The list of dependencies can be seen on Section 5.1.2

[8]Check Vicente et al. (2016a) Eq (21) for more details on the lambdaEdge parameter

Listing 6 | Likelihood Assessment: modules/likelihoodAssessment/src/likelihood.cpp

```
1. int lambdaEdge = 25;
2. // For each particle i – line 149 modules / likelihoodAssessment / src / likelihood.cpp
3. // Interopelability between the several libraries (OpenGL , CUDA, OpenCV)
4.   gltex =(GLuint) (size_t) (ID[i]); // ID is a vector with pointers to the render textures
5.   glBindTexture(GL_TEXTURE_2D, gltex);
6.   GLint width, height, internalFormat;
7.   glGetTexLevelParameteriv(GLTEXTURE_2D, 0, GL_TEXTURE_COMPONENTS, &internalFormat); // get internal format type of GL texture
8.   glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &width); // get width of GL texture
9.   glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &height); // get height of GL texture
10.
11.  checkCudaErrors( cudaGraphicsGLRegisterImage ( &cuda_tex_screen_resource , gltex , GL_TEXTURE_2D, cudaGraphicsMapFlagsReadOnly ) );
12.  // Copy color buffer
13.  checkCudaErrors( cudaGraphicsMapResources ( 1, &cuda_tex_screen_resource , 0 ) );
14.  checkCudaErrors( cudaGraphicsSubResourceGe tMappedArray ( &cuArr , cuda_tex_screen_resource, 0, 0 ) );
15.  BindToTexture( cuArr); // BindToTexture Functions defined in Cuda_Gl.cu
16.
17.  DeviceArrayCopyFromTexture( ( float3*) gpuMat.data, gpuMat.step, gpuMat.cols, gpuMat.rows );//DeviceArrayCopyFromTexture function defined on Cuda_Gl.cu
18.
19.  checkCudaErrors( cudaGraphicsUnmapResources ( 1, &cuda_tex_screen_resource, 0 ) );
20.  checkCudaErrors( cudaGraphicsUnregisterResource (cuda_tex_screen_resource) );
21.  cv::gpu::cvtColor(gpuMat, GgpuMat,CV_RGB2GRAY);
22.
23.  // Apply the likelihood Assessment
24.  // GgpuMat – generated Image
25.  // GgpuMat_R – Real Distance Transform image
26.  cv :: gpu :: multiply (GgpuMat, GgpuMat_R, GpuMatMul);
27.  cv :: Scalar sumS = cv :: gpu :: sum(GpuMatMul);
28.
29.  /*
30.  Check the article:
31.  Online Body Schema Adaptation Based on Internal Mental Simulation and Multisensory Feedback, Vicente et al.
32.  In particular, Equation (21)
34.  */
35.  sum = sumS [0]*lambdaEdge; // lambdaEdge is a tuning parameter for distance sensitivity
36.  nonZero = (float) cv::gpu::countNonZero (GgpuMat); // generated image
37.  if (nonZero ==0) {
38.     likelihood [i] = 0.000000001; // Almost Zero
39.  }
40.  else {
41.     result = sum/nonZero;
42.     likelihood[i] = (int) ((cv::exp(– result)) *1000);
43.  }
44.}
```

the result image is zero. This happen when every edge on the generated image match a zero distance on the distance transform image. The multiplication by 1,000 and the *int* cast in line 42 is used to send the likelihood as a int value (the inverse process is made in the internal model when it receives the likelihood vector) and it is one of the limitations of the current approach due to software limitations the authors could not send directly a double value between 0 and 1.

5. APPLICATION AND UTILITY

The Markerless kinematic calibration can run during normal operations of the iCub robot. It will update the joint offsets according to the new incoming observations. Moreover, one can also stop the calibration and use the estimated offsets so far, however, to achieve a better accuracy in different poses of the end-effector the method should be kept running in an online fashion to perform a better adaptation of the parameters.

The details of the dependencies, installation and how to run the modules can be found at ***Online Documentation page*** (check Section 2).

5.1. Installation and Dependencies

The dependencies of the proposed solution can be divided in two sets of libraries: (1) the libraries needed to run the handPoseEstimation module, and (2) the libraries needed to run the Robot's internal model and the likelihood Assessment.

5.1.1. Hand Pose Estimation Module

The handPoseEstimation depends on YARP library, which can be installed following the installation procedure of the official repository[9]. Moreover, it depends on the OpenCV library[10].

We tested this module with the last release of YARP (*i.e.*, June 15, 2017), version 2.3.70, with the OpenCV library V2.4.10 and V3.3 and the code works with both versions. The authors recommend the reader to follow the official installation guides for these libraries.

To install theses modules, one can just run *CMake* using the *CMakeLists.txt* on the folder:

```
/modules/handPoseEstimation/
```

5.1.2 Robot's Internal Model Generator and Likelihood Assessment

The Robot's internal model and the likelihood assessment depend on YARP library for communication and on the OpenCV library with CUDA enabled computation (*i.e.*, installing the CUDA toolkit) for image processing and GPGPU accelerated algorithms. A Windows machine should be used to install this module.

The tested version of the OpenCV library was V.2.4.10 with the CUDA toolkit 6.5. The C# bindings for the YARP middleware on a windows machine should be compiled. The details regarding the installations procedures can be found at the following URL:

[9]https://github.com/robotology/yarp

[10]It is not mandatory the CUDA-enabled capabilities

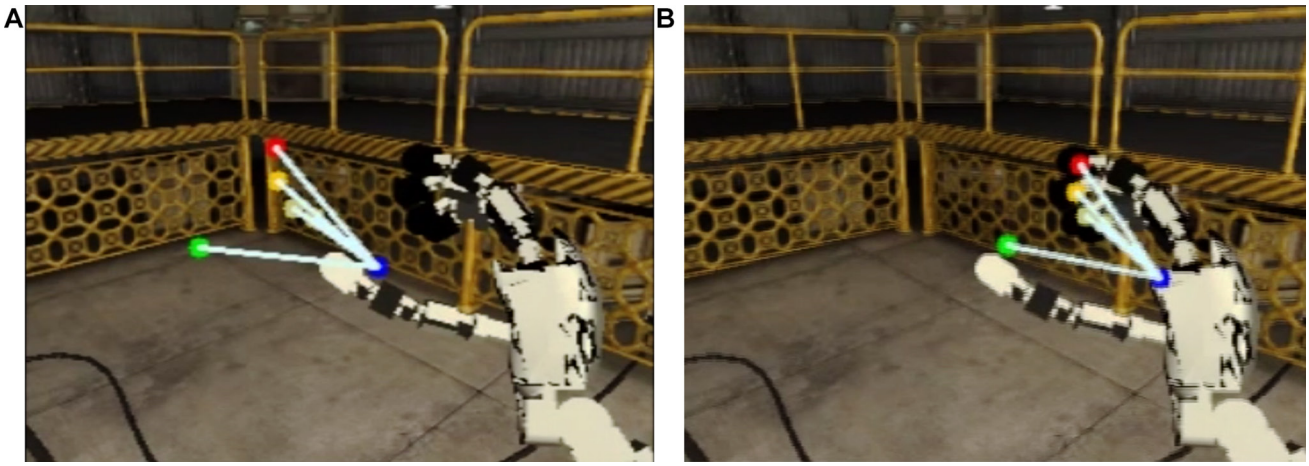


FIGURE 2 | Projection of the fingertips on the left camera on simulated robot experiments. The blue dot represents the end-effector projection (*i.e.*, base of the middle finger), the red represents the index fingertip, the green the thumb fingertip, the dark yellow the middle fingertip and the soft yellow the ring and little fingertips. On the left image **(A)** is the canonical projection (*i.e.*, with $\hat{\boldsymbol{\beta}} = 0$) and on the right image **(B)** the estimated offsets ($\hat{\boldsymbol{\beta}}$).

`http://www.yarp.it/yarp_swig.html#yarp_swig_windows`.

The C# bindings will allow the internal model generator to communicate with the other modules.

The C# bindings will generate a DLL file that, along with the DLL generated from the likelihood assessment module, should be copied to the `Plugins` folder of the internal model generator. In the official compiled version of the repository this folder has the following path: `internalmodel/icub-internalmodel-rightA-cam-Lisbon_Data/Plugins/`

The complete and step-by-step installation procedure can be seen in the ***Online Documentation page*** on the Installation section.

5.2. Running the Modules

The proposed method can run on a cluster of computers connected with the YARP middleware. The internal model generator should run on a computer with Windows Operating System and with CUDA capabilities. The step-by-step running procedure guide can be found on the ***Online Documentation page***. The rest of the section is organized with a high level perspective of running the algorithm. The YARP connections required between the several components can be connected through the XML file under the `app/scripts` folder.

5.2.1. Running the Hand Pose Estimation and its parameters

The Hand Pose Estimation can be initialized using the yarpmanager or in a terminal running the command:

```
handPoseEstimation [--<parameter_name>
<value > ...]
```

where, `<value>` is the value for one of the parameters (`<parameter_name>`) defined in the itemize list below:

- `name`: name of the module (default ="hpe")
- `arm`: arm which the module should connect to. (default = right')
- `initialMean`: mean for the initial distribution of the particles [in degrees]. (default = 0.0°)
- `initialStdDev`: StdDev of the initial distribution of the particles degrees
- `artificialNoiseStdDev`: initial Artificial Noise (StdDev) to spread the particles after each iteration (default = 3.0°)
- `lowerBound`: artificial noise lower bound (StdDev). Should be greater than Zero to prevent the particles to collapse in one single value (default = 0.04°)
- `upperBound`: artificial noise upper bound (StdDev). The artificial noise should have a upper bound to prevent the particles to diverge after each resampling stage (default = 3.5°)
- `minimumLikelihood`: minimumLikelihood [0,1] in order to resample the particles (default = 0.55)
- increaseMultiplier: increase the artificial noise of a certain value (currentValue*increaseMultiplier) if the maximum likelihood is lower than the minimumLikelihood (default = 1.15)
- `decreaseMultiplier`: decrease the artificial noise of a certain value (currentValue*decreaseMultiplier) if the maximum likelihood is greater than the minimumLikelihood (default = 0.85)
- `KDEStdDev`: StdDev of each kernel in the Kernel Density Estimation algorithm (default = 1.0°)
- `minIteration`: minimum number of iterations before sending the estimated offsets. The objective is to give time to the algorithm to converge, without this feature one can receive completely different offsets from iteration t to $t + 1$ during the filter convergence (default = 35)

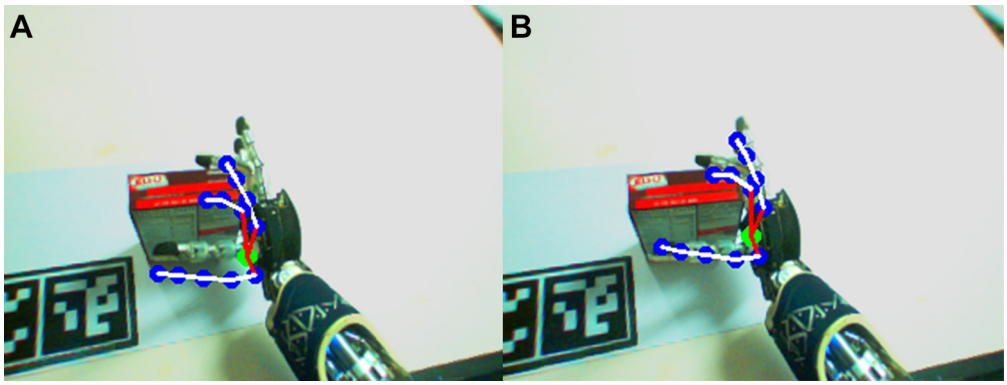


FIGURE 3 | Projection of the fingertips on left camera in real robot experiments. On the left image **(A)** the canonical projection (*i.e.*, with $\hat{\beta} = 0$) is shown, and on the right **(B)** the projection according with the corrected kinematic chain using the estimated offsets ($\hat{\beta}$).

5.2.2. Running the Robot's Internal Model

The internal model generator should run on a terminal using the following command:

```
icub-internalmodel-rightA-cam-Lisbon.exe
-force-opengl
```

The `-force-opengl` argument will force the robot's internal model to use the OpenGL library for rendering purposes, which is fundamental for the libraries interoperability.

5.2.3. User interface

The user can send commands to the Hand Pose estimation algorithm through the RPC port *hpe/rpc:i*. The RPC port acts like a service to the user where the algorithm can be started, stoped or paused/resumed. It is also possible to request the last joint offsets estimated by the algorithm. The thrift file (*modules/handPoseEstimation/handPoseEstimation.thrift*) contains the input and output of each RPC service function (*i.e.*, *start, stop, pause, resume, lastOffsets and quit*). More details about these commands can be seen in the use procedure on the documentation. Moreover, after connecting to the RPC port (*yarp rpc hpe/rpc:i*), the user can type `help` to get the available commands. The module also replies the input and output parameters of a given command if the user type *help FunctionName* (*e.g., help start*).

6. EXPERIMENTS AND EXAMPLES OF USE

The experiments performed with the proposed method on the iCub simulator, with ground truth data, have shown a good accuracy on the hand pose estimation, where artificial offsets were introduced in the seven joints of the arm. The results on the real robot have shown a significant reduction of the calibration error [Check Vicente et al. (2016a) Section 5 for more results in simulation (Section 5.1) and with the real iCub (Section 5.2)].

For the reader to be able to test the algorithm, the authors collected a simulated dataset (encoders of the head and arms, and the left and right images) which can be used to test the algorithm. The simulation results of the present article were obtained running the above-stated code with the default parameters on the collected dataset.

The dataset[11] was collected using a visual simulator based on the CAD model of the iCub humanoid robot adding artificial offsets in the arm joints. The artificial angular offsets β were the following:

$$\beta = \{-10.0, -10.0, 6.0, -7.0, -1.0, -20.0, 7.0\}°.$$

The robot performed a babbling movement which consists in a random walk in each joint. The minimum and maximum values of the uniform distribution used to generate the babbling movement starts at [−5, 5]°, and is reduced during the movement to [− 0.5, 0.5]°, respectively. Despite a great amount of errors in the robot's kinematic chain, the algorithm was able to converge to the solution in **Figure 2**. Moreover, the cluttered environment on the background did not influence the filter convergence. The reader can see the projection of the fingertips on the left camera image: (1) according to the canonical representation on **Figure 2A** (where it is assumed an error-free kinematic structure, *i.e.*, with $\hat{\beta} = 0$ and (2) the corrected kinematic structure using the algorithm implemented and documented in this code paper on **Figure 2B**.

The convergence of the algorithm along with a side-by-side comparison with the canonical solution can be seen in the following video: https://youtu.be/0tzLFqZLbxc

On the real robot, we already performed several experiments in previous works, with different initial and final poses using the 320 × 240 cameras. In **Figure 3** one can see one example of the hand estimation. While the image on the left (**Figure 3A**) shows the canonical estimation of the hand projected on the left camera image according to the non-calibrated kinematic chain, the image on the right (**Figure 3B**) shows the corrected kinematic chain which originates a better estimation of the hand pose. The rendering of the estimated hand pose was done taking into account the joint offsets on the kinematic chain before computing the hand pose in the image reference frame.

[11] https://github.com/vicentepedro/eyeHandCalibrationDataset-Sim

7. KNOWN ISSUES

There are some known issues or limitations in this algorithm and its software. The Windows dependency of the internal model generator module can be a problem for non-windows users. Moreover, the number of particles in the Sequential Monte Carlo is fixed (200 particles), which we found to be a good trade-off between accuracy and speed [check Vicente et al. (2016a) for more details on this matter].

The camera size is also fixed to the 320 × 240 resolution, which is sufficient to most of the experiments performed on the iCub. Indeed, to the authors' knowledge, this is the most popular resolution in the iCub community. The camera resolution can be modified by changing the input resolution on the hand pose estimation module and on the internal structures of the internal model and the likelihood assessment. However, this demands for a recompilation of the internal model generator which could not be done without a Pro license of Unity®.

The limitation on the integration of the likelihood assessment and the *int* cast discussed in Section 4.3 should be investigated since we are truncating the likelihood and in the end we have, at most, three significant figures of the likelihood value.

Hand occlusions can also be problematic at this stage of the work since we are not dealing explicitly with them. If the hand is occluded for a long period, the filter can start to diverge since it does not find a good match of the hand model in its perception.

8. CONCLUSION AND FUTURE WORK

In this paper, we have shown how to calibrate the eye-hand kinematic chain of a humanoid robot – the iCub robot. We have provided a tutorial on how to execute the module and how it works, its inputs and outputs. Our proposed work could be beneficial for research works with the iCub humanoid robot, from manipulation related fields to human-robot interaction, for instance. The results have shown a good accuracy in simulation and in a real-world environment. For future work, we are planning to extend the architecture. A useful feature is to be able to predict if the hand is present or not in the image or if it is occluded in order to perform a better match between the perception and the generated hypotheses. We will investigate the possibility of running the internal model simulator on different platforms (*i.e.*, Linux, macOS), which seems to be a new feature of the Unity game engine editor environment.

AUTHOR CONTRIBUTIONS

In this work, all the authors contributed to the conception of the markerless eye-hand kinematic calibration solution and to the analysis and interpretation of the data acquired.

FUNDING

This work was partially supported by Fundação para a Ciência e a Tecnologia [UID/EEA/50009/2013] and PhD grant [PD/BD/135115/2017] and by EPSRC UK (project NCNR, National Centre for Nuclear Robotics, EP/R02572X/1). We acknowledge the support of NVIDIA Corporation with the donation of the GPU used for this research.

REFERENCES

Borgefors, G., and Bradski, G. (1986). Distance transformations in digital images. *Computer Vision Graphics and Image Processing* 34 (3), 344–371. doi: 10.1016/S0734-189X(86)80047-0

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Fanello, S.R., Pattacini, U., Gori, I., Tikhanoff, V., Randazzo, M., and Roncone, A. (2014). "3D stereo estimation and fully automated learning of eye-hand coordination in humanoid robot" *IEEE-RAS International Conference on Humanoid Robots* 1028–1035.

Fantacci, C., Pattacini, U., Tikhanoff, V., and Natale, L. (2017). Visual end-effector tracking using a 3D model-aided particle filter for humanoid robot platforms. *arXiv preprint arXiv* 1703, 04771.

Garcia Cifuentes, C., Issac, J., Wuthrich, M., Schaal, S., and Bohg, J. (2017). Probabilistic articulated real-time tracking for robot manipulation. *IEEE Robot. Autom. Lett.* 2 (2), 577–584. doi: 10.1109/LRA.2016.2645124

Gratal, X., Romero, J., and Kragic, D. (2011). "Virtual Visual Servoing for Real-Time Robot Pose Estimation" *Proc. of the 18th IFAC World Congress* 9017–9022.

Hol, JD., Schon, TB., and Gustafsson, F. (2006). "On resampling algorithms for particle filters" *IEEE Nonlinear Statistical Signal Processing Workshop* 79–82.

Kantas, N., Doucet, A., Singh, S. S., and Maciejowski, J. M. (2009). "An overview of Sequential Monte Carlo methods for parameter estimation on general state space models," in *IFAC Symposium on System Identification (SYSID)*, Vol. 42, 774–785. doi: 10.3182/20090706-3-FR-2004.00129

Metta, G., Fitzpatrick, P., and Natale, L. (2006). YARP: yet another robot platform. *International Journal of Advanced Robotic Systems* 3 (1), 8. doi: 10.5772/5761

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23 (8-9), 1125–1134. doi: 10.1016/j.neunet.2010.08.010

Nickolls, J., Buck, I., Garland, M., and Skadron, K. (2008). Scalable parallel programming with CUDA. *Queue* 6 (2), 40–53. doi: 10.1145/1365490.1365500

Vicente, P., Jamone, L., and Bernardino, A. (2016a). Online body schema adaptation based on internal mental simulation and multisensory feedback. *Front. Robot. AI* 3:7. doi: 10.3389/frobt.2016.00007

Vicente, P., Jamone, L., and Bernardino, A. (2016b). Robotic hand pose estimation based on stereo vision and GPU-enabled internal graphical simulation. *J. Intell. Robot. Syst.* 83 (3-4), 339–358. doi: 10.1007/s10846-016-0376-6

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer, CF, and handling Editor declared their shared affiliation.

frontiers in Robotics and AI

CODE
published: 22 August 2018
doi: 10.3389/frobt.2018.00098

A Framework for Fast, Autonomous, and Reliable Tool Incorporation on iCub

*Tanis Mar, Vadim Tikhanoff and Lorenzo Natale**

iCub Facility, Istituto Italiano di Tecnologia, Genoa, Italy

One of the main advantages of building robots with size and motor capabilities close to those of humans, such as iCub, lies in the fact that they can potentially take advantage of a world populated with tools and devices designed by and for humans. However, in order to be able to do proper use of the tools around them, robots need to be able to *incorporate* these tools, that is, to build a representation of the tool's geometry, reach and pose with respect to the robot. The present paper tackles this argument by presenting a repository which implements a series of interconnected methods that enable autonomous, fast, and reliable tool incorporation on the iCub platform.

Keywords: tool use, code:cplusplus, tool incorporation, affordances, iCub, 3D reconstruction, humanoid, tool pose

OPEN ACCESS

Edited by:
Alexandre Bernardino, Instituto Superior Técnico, Universidade de Lisboa, Portugal

Reviewed by:
Vishwanathan Mohan, University of Essex, United Kingdom
Jose Antonio Gaspar, Universidade de Lisboa, Portugal
Pedro Vicente, Universidade de Lisboa, Portugal, in collaboration with reviewer JAG

***Correspondence:**
Lorenzo Natale
lorenzo.natale@iit.it

Specialty section:
This article was submitted to Humanoid Robotics, a section of the journal Frontiers in Robotics and AI

Received: *10 August 2017*
Accepted: *30 July 2018*
Published: *22 August 2018*

Citation:
Mar T, Tikhanoff V and Natale L (2018) A Framework for Fast, Autonomous, and Reliable Tool Incorporation on iCub. Front. Robot. AI 5:98. doi: 10.3389/frobt.2018.00098

1. OVERVIEW

A critical problem in most studies of tool use in developmental robotics is that actions are performed without considering the geometry or pose of tools that the robot uses. Instead, most experiments apply standard grasps and assume pre-defined kinematic end-effector extensions that do not take into account the particular pose of the tool in the robot's hand (Gonçalves et al., 2014; Dehban et al., 2017). In order to overcome this limitation, this paper presents a repository which implements a series of interconnected methods that enable autonomous, fast, and reliable estimation of a tool's geometry, reach and pose with respect to the iCub's hand, in order to attach it to the robot's kinematic chain, thereby enabling dexterous tool use. Indeed, this methods have been successfully applied in the study presented in Mar et al. (2017).

The repository can be found at:
https://github.com/robotology/tool-incorporation

We name this process *tool incorporation* because of its meaning referring to embodiment (literally, *in-corpore*), as it enables iCub to build a representation of the tool with respect to, and included in, its own body representation. The iCub is a full body humanoid robot with 53 Degrees of Freedom (DoF) (Metta et al., 2010), including head, arms, and torso. The iCub software is structured as modules that communicate with each other using YARP middleware, which enables multi-machine and multi-platform integration (Metta, 2006). Modules provide specific functionalities, and work together in form of applications to achieve desired behaviors on the iCub. Vision is provided by the cameras mounted in the robot's eyes, from which stereo matching can be applied to estimate depth (Fanello et al., 2014). Image processing is achieved with the help of OpenCV and PCL libraries, for 2D and 3D processing respectively (Rusu and Cousins, 2011; Itseez, 2015). All the methods described in this paper are implemented as functions in the `toolIncorporation` module.

The remainder of this paper is structured according to the main methods required to incorporate tools. Section 2 describes the methods for tool recognition, or visual appearance learning if the tool has not been seen before. Section 3 presents a method that enables iCub to reconstruct a 3D representation of the tool in its hand using its stereo-vision capabilities. Section 4 explains the meaning and estimation of the tool's intrinsic frame and of the tooltip. Finally, section 5 details a method for faster estimation of a tool's pose when its model is available.

2. TOOL RECOGNITION

The first step for tool incorporation is to recognize the tool in the robot's hand, so that its model can be loaded if the tool is known, or its visual appearance learned otherwise. To that end, the method applied in this work builds upon the techniques described in Pasquale et al. (2016). In that paper, a pre-trained CNN (AlexNet trained on imageNet, Krizhevsky et al., 2012) learned to associate a cropped image of an object presented by the experimenter with a provided label. In this work, we extended this approach in order to reduce the need of an external teacher, so that it is only required to hand over the tool to the robot and provide its label.

Once the iCub robot is grasping the tool in its hand, exploration is performed by moving it to different poses, so that it can be observed from different perspectives (implemented in function `exploreTool`). These poses are predefined to utilize the range of iCub's wrist joints to achieve distinct perspectives.

On each of the considered poses iCub focuses on the tool's effector, understood as the part of the tool that interacts with the environment. However, at this point the robot has no information about the tool's geometry or pose in order to estimate where the effector might be (these are discussed in section 4.3). Therefore, in order to locate the effector, iCub initially looks just slightly over its hand (10 cm along the X axis and −10 cm along the Y axis of the hand reference frame). Then, it locates the tooltip on the image by iteratively extracting the tool outline from the disparity map, and looking at the point in the blob further away from the hand reference frame. This process, which is implemented in function `lookAtTool`, is repeated until the position of the estimated tooltip is stable, or a given number of iterations has been surpassed.

Once iCub is correctly gazing at the tool effector, a series of images of the tool are obtained by cropping a region around the tool, which is determined by the bounding box of the closest blob obtained with `dispBlobber`[1], plus a margin of 10 pixels on each side. Finally the cropped images are fed to a CNN whose output feeds in turn a linear classifier which associates them to the user provided tool label. This process is performed by the `onTheFlyRecognition` application, which is called from by the `learn` function provided with the tool label.

This sequence –tool effector location and subsequent cropping of the tool region to feed the CNN– is repeated for all the exploration poses considered, which provides enough perspectives to recognize the tool in any future pose in which it might be grasped in the future.

After the visual appearance of the set of available tools has been learned, the process of classification is simple. After iCub is given any tool, it observes it in any of the exploratory poses and uses the same method to crop it from the rest of the image. The cropped image is in turn sent to the trained classifier (in this case, using the `recognize` function), which returns the estimated label of the tool. It should be noted that tools can be learned in either terms of instances or categories. In the first case, the user should provide a distinct label for each individual tool given to iCub, and an associated pointcloud model. In the second case, tools of the same category (e.g., rakes, sticks, shovels), should be given the generic label of that category, and a generic model of the tool category provided.

3. TOOL 3D RECONSTRUCTION

In cases where a 3D model of the tool is not available, the robot should be able to reconstruct it through exploration. In this section we describe an approach that allows iCub to achieve this, without the need of external intervention by the experimenter. Essentially, it consists of iterative segmentation, reconstruction, and merge of the tool's partial views from different perspectives.

Similar techniques have been presented in many different papers in the recent years (Ren et al., 2013; Zhang et al., 2015). However, most of these studies assume either a fixed camera and an object being moved externally (by the user or on a turning table), which could not be considered autonomous; or a fix scene and a moving camera/robot navigating around it, which is unfeasible on the current iCub setup. Therefore, in the present work we implemented a method by means of which iCub can reconstruct a tool's complete pointcloud representation by obtaining partial view reconstructions from different perspectives and incrementally merging them together.

The method applied to observe the tool effector is analogous to the one described in the previous section for learning the tool's visual appearance, and in fact, both processes can be run simultaneously (by calling the `exploreTool` function with the 2D and 3D flags active). For reconstruction, the steps performed at each exploration pose are the following:

- Segmentation:
  After the gaze is properly oriented toward the tool effector, as described in section 2, instead of just cropping the bounding box around the tool, the tool blob is segmented with the `dispBlobber` module, which returns the pixels in the image that correspond to the tool.
- Reconstruction:
  This list of pixels is sent to the `seg2cloud` module, which computes the 3D coordinates of each point in the robot reference frame and returns them as a pointcloud. This pointcloud is transformed from the robot frame to the hand's reference frame using the robot's kinematics, which greatly facilitates subsequent merging, as the hand

[1]https://github.com/robotology/segmentation/tree/master/dispBlobber

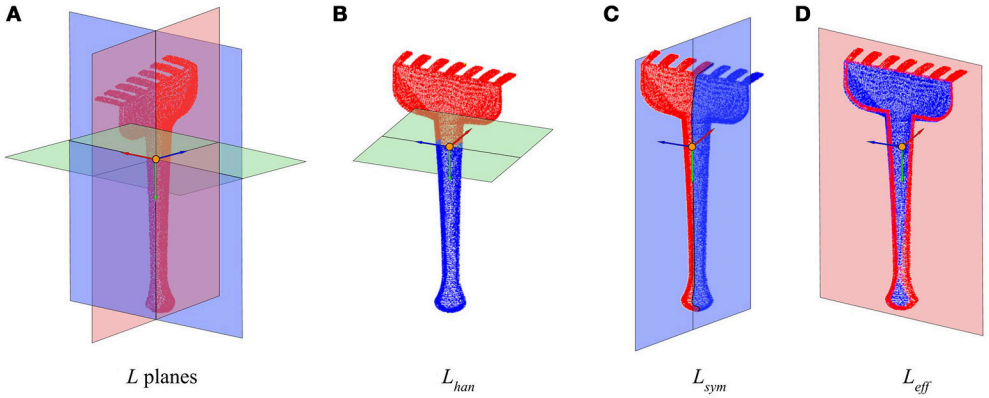


FIGURE 1 | Planes and axes that determine the tool's intrinsic reference frame. **(A)** Tool model divided by its three characteristic L planes. **(B)** Handle plane L_{han}. **(C)** Symmetry plane L_{sym}. **(D)** Effector plane L_{eff}. The reference frame in all figures shows $\mathbf{f}_{eff}$ in red, $\mathbf{f}_{han}$ in green, and $\mathbf{f}_{sym}$ in blue.

provides a coherent reference frame for all the partial reconstructions.

Moreover, we can safely assume that the tool is connected with the hand, and it does not extend beyond certain boundaries. Therefore, in order to remove any points on the reconstructed pointcloud that might belong to the background, the pointcloud is truncated in all three axes of the hand reference frame, removing all the points outside the $(0.0, 35)$cm range in the X axis, $(-30, 0.0)$cm range in the Y axis, and $(-15, 15)$cm range in the Z axis. Additionally, as in many cases part of the hand might also be present in the reconstructed pointcloud, it is removed by filtering out all the points in the which are inside a radius of 8 cm from the origin of the hand reference frame. Finally, the pointcloud is smoothed by applying a statistical filter for outlier removal. The described pointcloud reconstruction, transformation and filtering are performed by the `getPointCloud` function.

- Merging:

  Although all the partial reconstructed pointclouds are represented in a coherent reference frame, they are not perfectly aligned due to errors in depth estimation and robot kinematics. Therefore, a further refinement step is performed using the Iterative Closest Point algorithm (ICP) (Besl and McKay, 1992). We assume that the required refinement is small and thus discard as unsuccessful those cases in which the resulting roto-translation is larger than a given threshold. Finally, in order to merge overlapping surfaces and reduce noise, the resulting pointcloud is downsampled uniformly using a voxelized grid.

As a result of this process, a complete pointcloud representation of the explored tool is obtained, which also reflects the pose with which the tool is being grasped by iCub. We refer to this representation as an **oriented pointcloud model**, that is, the available pointcloud model of the tool being held by the robot, whose coordinates match the position of the actual tool with respect to the robot's hand reference frame.

4. TOOL REFERENCE FRAME AND TOOLTIP ESTIMATION

Although the pose of the oriented pointcloud model corresponds to that of the tool in the robot's hand, its orientation is not readily available for the robot, as it is only implicit in the pointcloud representation. In the present section we present a method to make this information explicit, based on the definition and estimation of a reference frame intrinsic to each tool, applicable to the vast majority of man-made tools that could be present in a robotic tool use scenario. This frame of reference, referred to as **tool intrinsic reference frame**, and denoted as **f**, identifies the effector and handle of the tool, provides its orientation with respect to the hand reference frame, and facilitates the computation of the tooltip's location.

4.1. Tool Reference Frame Definition

Given any radial tool[2], generally we can define three orthogonal characteristic tool planes as can be observed in **Figure 1**, denoted together as a tool's L planes:

- **Handle plane** (L_{han}): It is perpendicular to the handle axis, and divides the tool into the effector and the handle sides.
- **Symmetry plane** (L_{sym}): It is the plane with respect to which the tool has the maximum symmetry. It runs along the handle and divides the tool into two equal (or almost) longitudinal halves.
- **Effector plane** (L_{eff}): Orthogonal to the two previous planes, usually divides the "forward" and "back" sides of the tool, forward being the side where the effector is.

The planes' normal vectors can be chosen so that they define a right-hand reference frame, which we refer to as the **tool intrinsic reference frame**,(**f**). To this end, the origin and orientation of the

[2]We refer as radial tools to tools consisting of clearly distinct handle and effector, which are grasped from the handle with the thumb toward the effector of the tool (called radial grip).

corresponding axes is chosen so that they preserve the following characteristics:

- **Effector axis (*X*)** ($\mathbf{f}_{eff}$): It is positive in the direction of the effector, i.e., toward the "forward" side of the tool.
- **Handle axis (*Y*)** ($\mathbf{f}_{han}$): Is positive in the direction toward the handle, and negative in the direction toward the effector side of the tool.
- **Symmetry axis (*Z*)** ($\mathbf{f}_{sym}$): The symmetry basis vector is obtained as the outer product of the other two to ensure orthogonality, so it is positive on the "left" side of the tool, if the effector is looking "forward".

4.2. Tool Reference Frame Estimation

Based on the previous definitions, here we propose a method to automatically estimate the tool intrinsic reference frame $\mathbf{f}$ of a tool's pointcloud representation W, relying solely on the assumption that W represents an oriented pointcloud model, that is, it is expressed with respect to the hand reference frame of the robot. The proposed procedure consists on the following steps, which can be observed in function `findSyms`:

1. Find the pointcloud's main axes: The estimation of $\mathbf{f}$'s origin and direction can be achieved by computing the covariance matrix of the pointcloud W. The origin is determined at the center of mass o, and the 3 eigenvectors $\mathbf{v}$ with larger eigenvalues λ correspond to the pointcloud's main axes:

$$C = cov(W), \tag{1}$$
$$C\mathbf{v} = \lambda\mathbf{v}, \tag{2}$$
$$L[i] \perp \mathbf{v}[i], i \in \{0, 1, 2\}. \tag{3}$$

   Therefore, this set of orthogonal vectors $\mathbf{v}$ defines a set of orthogonal planes that approximate the tool planes L, but their correspondence with the specific planes defined above, as well as their orientation, need to be determined to fully characterize $\mathbf{f}$.
2. Identify the planes:
   a. Handle plane L_{han}: The handle is situated along the longest tool dimension. Thus, the eigenvector with largest eigenvalue indicates the direction of the handle axis, normal to the Handle plane. That is,

$$\mathbf{v}_{han} = \mathbf{v}[n], \quad \text{where } n = \underset{i\in\{0,1,2\}}{\arg\max}\,(\lambda[i]), \tag{4}$$
$$\text{accordingly, } L_{han} = L[n] \tag{5}$$

   b. Symmetry plane L_{sym}: The symmetry plane corresponds by definition to the plane with respect to which the tool has the maximum symmetry. Thus:

$$L_{sym} = L[m], \quad \text{where } m = \underset{j\in\{0,1,2\}\neq n}{\arg\max}\,(sym(L[j])). \tag{6}$$

   c. Effector plane L_{eff}: The effector plane is computed in relation to previous two planes, as the plane orthogonal to both the Handle and the Symmetry plane:

$$L_{eff} = L[k], \quad \text{where } k \in 0, 1, 2 \neq n, m \tag{7}$$
$$L_{eff} \perp L_{sym} \perp L_{han} \tag{8}$$

3. Find the axes orientations:
   a. Handle axis $\mathbf{f}_{han}$: Determines the side where the handle of the tool is (opposite of the effector). Following the assumption that W is represented with respect to the hand reference frame, it follows that the handle is on the side of L_{han} that contains the origin of the pointcloud reference frame (i.e., the hand). Thus, the orientation of $\mathbf{f}_{han}$ is set so that the positive values correspond to the side of L_{han} that contains the origin.
   b. Effector axis $\mathbf{f}_{eff}$: In order to determine the direction that corresponds with "forward" in a tool, we consider the saliency of the features on each side of the effector plane. Specifically, the "forward" side of the pointcloud W is defined as the side where the effector half of the tool (determined in the previous step) contains points further away from the tool's intrinsic reference frame origin o. Thus, the orientation of the effector axis $\mathbf{f}_{eff}$ (perpendicular to the effector plane) is set such that the positive values are located on the salient side of the effector plane.
   c. Symmetry axis $\mathbf{f}_{sym}$: The orientation of $\mathbf{f}_{sym}$ is chosen so that the set of axes defined by $\mathbf{v}$ corresponds to a right-handed coordinate system. Thus, it is computed as the cross product between the handle and effector axes basis vectors:

$$\mathbf{f}_{sym} = \mathbf{f}_{han} \times \mathbf{f}_{eff} \tag{9}$$

The tool intrinsic reference frame $\mathbf{f}$ is actually expressed on the same frame of reference that the pointcloud reconstruction from which it is estimated, that is, the hand reference frame. Thus, the equations of the frame's axes represent explicitly the orientation of the tool in any of its three axis.

One of the strengths of this approach to estimate the tool's frame of reference $\mathbf{f}$ is that it relies on very few and general assumptions to be met in order to work successfully, namely, that the tool's handle axis is longer that any other axis, and that the tool has a certain degree of symmetry along a plane that contains that axis. Moreover, the method is also very robust to noise in the 3D representation of the tool, since all the computations required throughout the process of determining $\mathbf{f}$ have a high tolerance to noise. Indeed, as most of the decisions are made in terms of comparison (symmetry between two sides of a plane, longest axis, furthest away point), if noise affects the whole pointcloud similarly, it would not modify their outcome.

4.3. Tooltip Estimation

As stated above, one of the main advantages of estimating the tool reference frame $\mathbf{f}$ is that it enables to precisely locate the tooltip, required to perform the extension of the robot's kinematic chain to the new end-effector provided by the tool. Thus, the tool tip is defined in terms of the concepts defined and estimated above:

Tooltip: Location on the tool represented by the point on the Symmetry plane of the tool, above the Handle plane (i.e., on

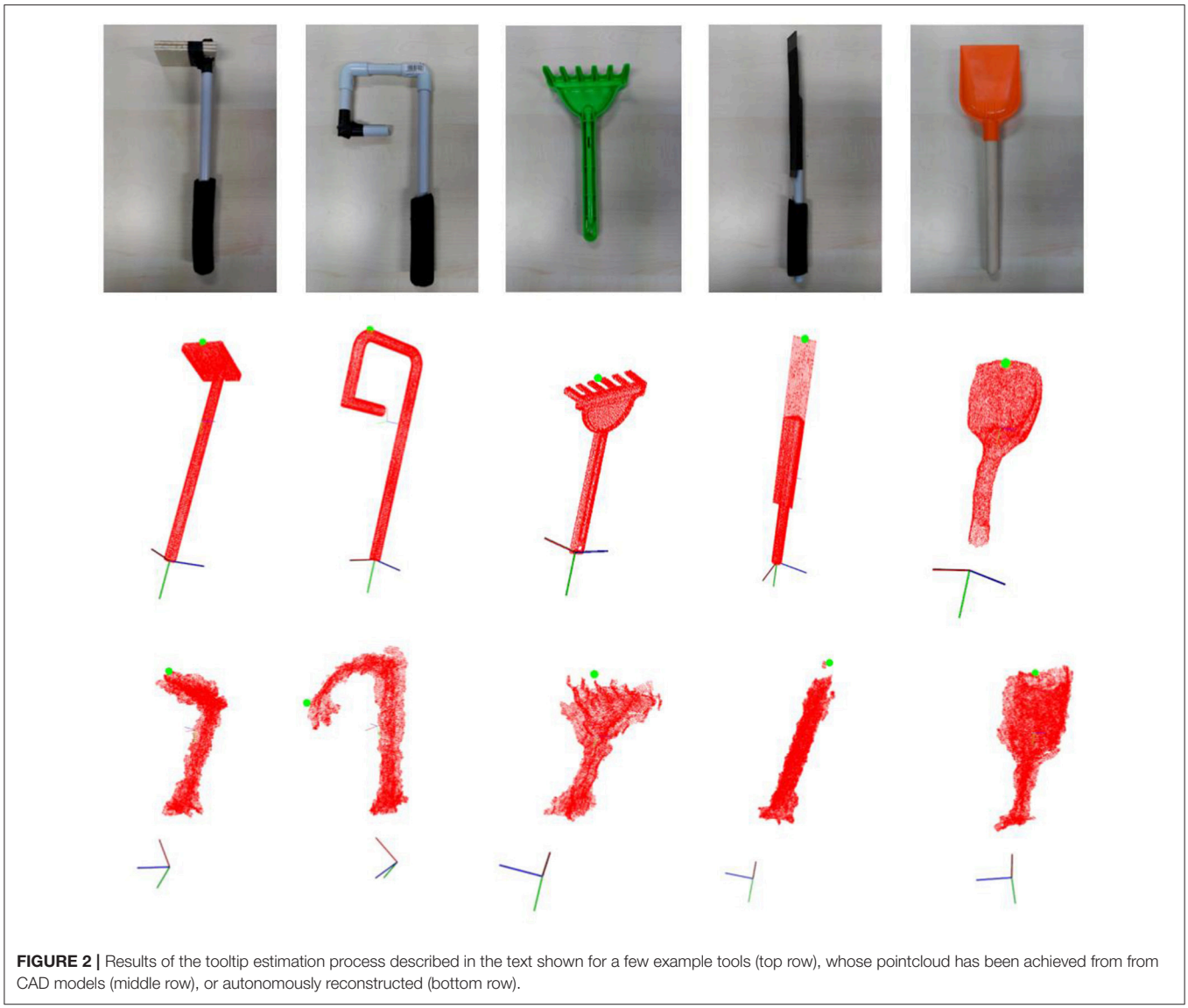


FIGURE 2 | Results of the tooltip estimation process described in the text shown for a few example tools (top row), whose pointcloud has been achieved from from CAD models (middle row), or autonomously reconstructed (bottom row).

the effector side), furthest away from the Effector plane, on the positive side of the effector axis.

The estimated tool reference frame **f** and tooltip for a small sample of tools can be observed in **Figure 2**, where it can be observed that the estimated tooltip coincides to what most people would consider to be the tooltip of those tools.

In our code, this definition is implemented by the function `findTooltipSym`, which computes the tooltip location based on the information from the tool planes provided by the previous steps.

5. TOOL POSE ESTIMATION

The methods described in sections 3 and 4 allow the robot to reconstruct a tool's geometry and estimate its pose even in the case of previously unseen tools. However, this is a time consuming approach that is not necessary if a 3D pointcloud model of the tool or tool category is already available, either from a CAD model or from a previous reconstruction. For these cases, in this section we introduce a fast and reliable method for pose estimation, based on the alignment of the available model with a single partial view reconstruction to the tool in the robot's hand, implemented in the function `findPoseAlign`.

Qualitatively, the *tool pose* represents the way in which the tool is being grasped with respect to the hand's reference frame. Numerically, we can express the tool pose in terms of the 4×4 roto-translation **Pose Matrix** P required to transform the hand reference frame $< H >$ frame to any reference frame intrinsic to the tool $< T >$, that is,

$$< T > = P < H > \tag{10}$$

The hand reference frame $< H >$ is defined by the robot kinematics. The tool reference frame $< T >$ applied can be arbitrarily chosen, as long as it is coherent among all the tools that can be considered, as the Pose is expressed in relative terms.

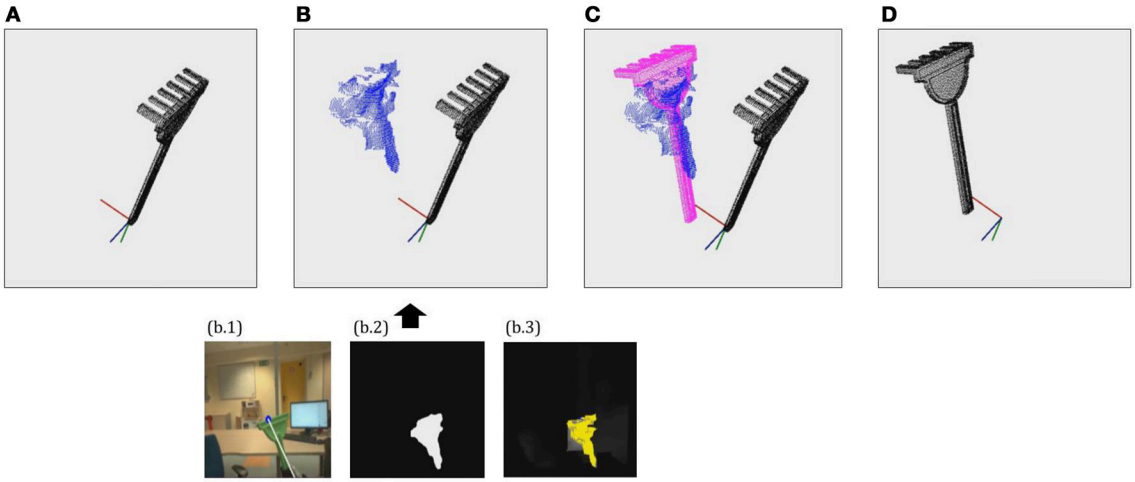


FIGURE 3 | Example of the tool pose estimation through alignment process. **(A)** Load 3D pointcloud model on canonical pose. **(B)** Extract partial reconstruction using `seg2cloud` model (segmentation + depth estimation). **(C)** Find Pose Matrix *P* by aligning 3D model to partial reconstruction. **(D)** Obtain oriented pointcloud model by applying *P* to the 3D model.

This means that *P* can also be understood as the required transformation to align a tool 3D model from its canonical pose to the pose in which is the tool is being held by the robot, given by the oriented pointcloud model. In this work, this transformation is estimated by aligning the available model of the tool with a partial reconstruction obtained through iCub's disparity.

To that end, iCub first applies the method described in section 2 to identify the tool instance or category and load the corresponding model. Then, it fixates the gaze on the tool's effector and extracts a partial pointcloud reconstruction, using the same methods applied on each of the exploration poses considered for tool reconstruction, as detailed in section 3. Then, the ICP algorithm is applied in order to align the pointcloud model loaded from memory to the partial reconstruction just obtained. Finally, the alignment matrix returned by the ICP is checked to assess whether it corresponds to a feasible grasp pose in terms of translation from the origin and rotation in Z and X axes. If the alignment estimated by ICP corresponds to a feasible grasp, then the returned alignment matrix is assigned to *P*, and applied to transform the canonical pointcloud model available in memory in order to obtain the oriented pointcloud model. This process can be observed in **Figure 3**.

Thereby, after the pose estimation process iCub has explicit information about the precise geometry and pose of the tool in its hand. Therefore, it can apply the method described in section 4.3 to determine the position of the tooltip with respect to the robot's hand reference frame, and hence extend the kinematics of the robot to incorporate the tip of the tool as the new end-effector for further action execution.

6. CONCLUSION

In the present paper we have introduced the concept of tool incorporation, that is, the process whereby the iCub robot is able to recognize a tool, estimate its geometry, pose and tooltip, and use this information to use the tool as its new end-effector. In particular, we have introduced a repository which implements a set of interconnected methods to perform such tasks in a fast and reliable way on iCub platform. By applying these methods, the robustness of the desired tool use behaviors as well as the ease of implementation can be substantially increased, by reducing the necessity of applying predefined parameters to represent the tools.

Despite its clear advantages, this approach does however suffer from a few limitations. On the one hand, it only works properly with radial tools where handle and effector are clearly distinct and are grasped radially (in the direction of the iCub's thumb). On the other, the 3D reconstruction quality, while generally enough to estimate the tool frame and the tooltip, does yield relatively noisy models. These issues clearly demand further work on tool incorporation mechanisms in order to facilitate robotic tool use.

AUTHOR CONTRIBUTIONS

TM is the main author of the code and paper. VT provided technical guidance and assistance, and reviewed both the code and the paper. LN provided high-level supervision, and reviewed the paper before submission.

REFERENCES

Besl, P., and McKay, N. (1992). A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.* 14, 239–256. doi: 10.1109/34.121791

Dehban, A., Jamone, L., and Kampff, A. R. (2017). "A deep probabilistic framework for heterogeneous self-supervised learning of affordances," in *Humanoids 2017* (Birmingham).

Fanello, S. R., Pattacini, U., Gori, I., and Tikhanoff, V. (2014). "3D Stereo estimation and fully automated learning of eye-hand

coordination in humanoid robots," in *Humanoids 2014* (Madrid), 1028–1035.

Gonçalves, A., Abrantes, J., Saponaro, G., Jamone, L., and Bernardino, A. (2014). "Learning intermediate object affordances: toward the development of a tool concept," in *IEEE International Conference on Development and Learning and on Epigenetic Robotics (ICDL-EpiRob 2014)* (Genoa), 1–8.

Itseez (2015). *Open Source Computer Vision Library*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). "ImageNet classification with deep convolutional Neural Networks," in *Advances in Neural Information Processing Systems* (Lake Tahoe, NV), 1–9.

Mar, T., Tikhanoff, V., and Natale, L. (2017). What can I do with this tool? Self-supervised learning of tool affordances from their 3D geometry. *IEEE Trans. Cogn. Dev. Sys.* 1. doi: 10.1109/TCDS.2017.2717041

Metta, G. (2006). *Software Implementation of the Phylogenetic Abilities Specifically for the iCub & Integration in the iCub Cognitive Architecture*. Technical Report 004370.

Metta, G., Natale, L., Nori, F., Sandini, G., Vernon, D., Fadiga, L., et al. (2010). The iCub humanoid robot: an open-systems platform for research in cognitive development. *Neural Netw.* 23, 1125–1134. doi: 10.1016/j.neunet.2010.08.010

Pasquale, G., Ciliberto, C., Rosasco, L., and Natale, L. (2016). "Object identification from few examples by improving the invariance of a deep convolutional neural network," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)* (Daejeon: IEEE). doi: 10.1109/IROS.2016.7759720

Ren, C. Y., Prisacariu, V., Murray, D., and Reid, I. (2013). "STAR3D: Simultaneous tracking and reconstruction of 3D objects using RGB-D data," in *Proceedings of the IEEE International Conference on Computer Vision* (Sydney, NSW), 1561–1568.

Rusu, R. B., and Cousins, S. (2011). "3D is here: Point Cloud Library (PCL)," in *Proceedings - IEEE International Conference on Robotics and Automation* (Shanghai).

Zhang, Y., Gibson, G. M., Hay, R., Bowman, R. W., Padgett, M. J., and Edgar, M. P. (2015). A fast 3D reconstruction system with a low-cost camera accessory. *Sci. Rep.* 5, 1–7. doi: 10.1038/srep10909

Conflict of Interest Statement: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

The reviewer JAG and handling Editor declared their shared affiliation.

Advantages of publishing in Frontiers

OPEN ACCESS
Articles are free to read for greatest visibility and readership

FAST PUBLICATION
Around 90 days from submission to decision

HIGH QUALITY PEER-REVIEW
Rigorous, collaborative, and constructive peer-review

TRANSPARENT PEER-REVIEW
Editors and reviewers acknowledged by name on published articles

Frontiers
Avenue du Tribunal-Fédéral 34
1005 Lausanne | Switzerland

Visit us: www.frontiersin.org
Contact us: info@frontiersin.org | +41 21 510 17 00

REPRODUCIBILITY OF RESEARCH
Support open data and methods to enhance research reproducibility

DIGITAL PUBLISHING
Articles designed for optimal readership across devices

FOLLOW US
@frontiersin

IMPACT METRICS
Advanced article metrics track visibility across digital media

EXTENSIVE PROMOTION
Marketing and promotion of impactful research

LOOP RESEARCH NETWORK
Our network increases your article's readership