

Spring 2014

Software Architecture and Development for Controlling a Hubo Humanoid Robot

Manas Ajit Paldhe

Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses



Part of the [Computer Sciences Commons](#), and the [Robotics Commons](#)

Recommended Citation

Paldhe, Manas Ajit, "Software Architecture and Development for Controlling a Hubo Humanoid Robot" (2014). *Open Access Theses*. 232.

https://docs.lib.purdue.edu/open_access_theses/232

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Manas Paldhe

Entitled

Software Architecture and Development for Controlling a Hubo Humanoid Robot

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

CHUN-SING GEORGE LEE

Chair

CHENG-KOK KOH

YUNG-HSIANG LU

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): CHUN-SING GEORGE LEE

Approved by: M. R. Melloch

Head of the Graduate Program

04-17-2014

Date

SOFTWARE ARCHITECTURE AND DEVELOPMENT FOR CONTROLLING A
HUBO HUMANOID ROBOT

A Thesis

Submitted to the Faculty

of

Purdue University

by

Manas Paldhe

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2014

Purdue University

West Lafayette, Indiana

This thesis is dedicated to my dear parents.

ACKNOWLEDGMENTS

Foremost, I would like to express my gratitude to my advisor Prof. C. S. George Lee, for his guidance and encouragement throughout my research work. I cannot imagine having a better advisor and mentor for my research. I also wish to thank Prof. Cheng-Kok Koh and Prof. Yung-Hsiang Lu for serving on my Advisory Committee.

I am thankful to my lab-mates at the Assistive Robotics Technology Laboratory (ARTLab): Roy Chan, Yan Gu and Andy Park, for stimulating discussions and their support. I also want to thank my friends Jingru Luo and Yajia Zhang at the Intelligent Motion Laboratory, Indiana University, Michael Grey and Pete Vieira at the Humanoid Robotics Lab, Georgia Institute of Technology, and Robert Ellenberg and Daniel Lofaro at the Drexel Autonomous Systems Laboratory, Drexel University for sharing their software and helping me with my problems throughout this research work.

I also want to thank my family and friends for their supports. I am indebted to them for constant motivation.

Last but not the least, I would like to thank the Defense Advanced Research Projects Agency (DARPA) and the National Science Foundation (NSF) for sponsoring this research. This work was supported in part by the DARPA award # N65236-12-1-1005 for the DARPA Robotics Challenge (DRC) and by the NSF under Grant CNS-0960061. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of DARPA and the NSF.

TABLE OF CONTENTS

	Page
LIST OF TABLES	viii
LIST OF FIGURES	ix
ABSTRACT	xii
1 INTRODUCTION	1
1.1 Motivation	6
1.2 Related work	8
1.3 Contributions	13
1.4 Impact	15
1.5 Organization of the thesis	16
2 OVERVIEW OF HUBO2+ HUMANOID ROBOT	18
2.1 Introduction	18
2.2 Mechanical structure and representation of Hubo2+ robot	19
2.3 Electronic and electro-mechanical modules of the Hubo2+ robot	24
2.4 Integration of all electronic and electro-mechanical modules into one system	35
2.5 Summary	37
3 ARCHITECTURE OF HUBO-ACH: A LOW LEVEL HUMANOID ROBOT CONTROLLER	38
3.1 Introduction	38
3.2 Purpose of hubo-ach	38
3.3 Architecture of ROS	39
3.3.1 Overview of ROS	39
3.3.2 Analysis of non-real-time behavior of ROS	41
3.3.3 Application of ROS	42

	Page
3.4 Architecture of hubo-ach	43
3.4.1 Overview of hubo-ach	43
3.4.2 Ach: Inter-process communication (IPC) library	47
3.4.3 Data structures	49
3.4.4 Abstracting communication	57
3.4.5 Organizational flow of hubo-ach	60
3.5 Hubo-console and Hubo-read: Wrappers around hubo-ach	65
3.6 Summary	67
4 ARCHITECTURE OF HUBO-MOTION-RT: A HIGH LEVEL HUMANOID ROBOT CONTROLLER	69
4.1 Introduction	69
4.2 Purpose of hubo-motion-rt	69
4.3 Architecture of hubo-motion-rt	71
4.3.1 Overview of hubo-motion-rt	71
4.3.2 Abstracting out ach communication for sensor feedback . . .	73
4.3.3 Joint velocity control and torque control	74
4.3.4 Gravity compensation	79
4.3.5 Balance controller	84
4.3.6 Integration of all controllers into one package	85
4.4 Writing custom software using hubo-motion-rt	88
4.5 Summary	90
5 APPLICATIONS AND USAGE OF HUBO-ACH AND HUBO-MOTION- RT	91
5.1 Introduction	91
5.2 Teleoperation of a robot	91
5.3 Hubo-read-trajectory	92
5.4 Hubo-neck	95
5.5 Hubo-init	96
5.6 Gravity-compensation-based trajectory following	98

	Page
5.7 ROS_hubo_ach	101
5.8 Summary	103
6 EXPERIMENTAL WORK ON DRC-HUBO TO TEST HUBO-ACH AND HUBO-MOTION-RT	104
6.1 Introduction	104
6.2 Analysing the use of force/torque sensors for detecting grasping . .	105
6.3 Ladder climbing	106
6.3.1 Motion planning for ladder climbing	108
6.3.2 Execution of the planned trajectory	114
6.3.3 Climbing using only position-control mode	115
6.3.4 Climbing using only PWM-control mode in the upper-body of the robot	116
6.3.5 Climbing partly using position-control and partly using PWM- control mode	116
6.4 DARPA Robotics Challenge- 2013 Trials	120
6.5 Summary	121
7 CONCLUSION	123
7.1 Introduction	123
7.2 Analysis of the performance of the software packages	123
7.3 Analysis of the experimental work	128
7.4 Future work	128
LIST OF REFERENCES	130
A D-H REPRESENTAION AND TRANSFORMATION MATRICES . . .	137
A.1 D-H Representation and forward kinematics	137
A.2 Inverse kinematics	141
B DYNAMIC COMPUTATIONS	144
B.1 Jacobian matrix	144
B.2 Newton-Euler computations	146
C HUBO-ACH MANUAL	148

	Page
C.1 Installation	148
C.2 Usage	148
C.3 Support	150
D HUBO-MOTION-RT MANUAL	151
D.1 Installation	151
D.2 Usage	152
D.3 Support	152
E HUBO-READ-TRAJECTORY MANUAL	154
E.1 Usage	154
E.2 Support	156
F HUBO-INIT MANUAL	157
F.1 Usage:	158
F.2 Support	158

LIST OF TABLES

Table	Page
1.1 Design philosophies of Hubo-ach, ROS, and MRDS.	12
2.1 Specification of Hubo2+ humanoid robot in ARTLab.	20
5.1 Comparison of various remote access software.	97
6.1 Comparison of various ladders.	115
6.2 Comparison of the three ladder climbing strategies.	118
A.1 D-H Parameter of Hubo2+ robot's right arm. These parameters are used to compute forward and inverse kinematic solutions.	138
A.2 D-H Parameter of Hubo2+ robot's right leg. These parameters are used to compute forward and inverse kinematic solutions.	139

LIST OF FIGURES

Figure	Page
1.1 Robots being used in a wide range of applications like manufacturing, surgeries, space exploration and assisting in rescue operations. [2–5] . . .	2
1.2 Depiction of how robots will affect daily life in the future. [6–9] . . .	3
1.3 Humanoid robots performing different tasks. [45–48]	5
2.1 A picture of the Hubo2+ humanoid robot showing the location of various electronic modules.	19
2.2 Engineering drawing of Hubo2+ humanoid robot showing some of the dimensions and size of different parts [31].	22
2.3 Denavit-Hartenberg representation of a Hubo2+ humanoid robot. . .	23
2.4 The two computers of Hubo2+ robot at ARTLab: (left) head-computer and (right) body-computer.	26
2.5 Depiction of how the position-control mode functions.	27
2.6 Depiction of how the PWM-control mode functions.	27
2.7 Motor controller boards used in Hubo2+ robot.	29
2.8 The power supply of Hubo2+ robot. AC input is provided to the robot.	30
2.9 Battery used in the Hubo2+ robot.	31
2.10 A picture of the IMU sensor mounted on the waist of the Hubo2+ robot.	32
2.11 Pictures of the force/torque sensors used on the wrist (left) and ankles (right) of Hubo2+ robot.	33
2.12 Communication in the Hubo2+ robot.	34
2.13 Peak system’s four channel CAN card used for CAN communication in the Hubo2+ robot at ARTLab.	34
3.1 Illustration of ROS publishers and subscribers.	40
3.2 Illustration of ROS service.	40
3.3 Illustration of how ROS can be used to run a robot.	41
3.4 Architecture of hubo-ach.	46

Figure	Page
3.5 Histograms of Ach and Pipe messaging latencies. Benchmarking performed on a Core 2 Duo running Ubuntu Linux 10.04 with PREEMPT kernel. The labels s/r indicate a test/run with sending processes and receiving processes [71].	48
3.6 Communication between processes using Ach.	48
3.7 hubo_joint_state: The data structure that stores the joint and encoder data.	51
3.8 hubo_ft: The data structure that stores force/torque sensor data. . . .	52
3.9 hubo_imu: The data structure that stores IMU sensor data.	52
3.10 hubo_joint_status : The data structure that stores a joint's working state.	53
3.11 hubo_state: The data structure used to store the current state of a humanoid robot.	55
3.12 hubo_ref: the data structure that stores the latest commands sent to a joint.	57
3.13 Joint motion in filter-mode (left) and direct reference mode (right). . .	57
3.14 Components of hubo_ach.	59
3.15 Organizational flow of hubo_ach.	61
3.16 Depiction of how hubo-read and hubo-console interact with hubo-ach .	67
4.1 Architecture of hubo-motion-rt.	72
4.2 Schematic of joint motion in position-control mode using hubo-motion-rt.	76
4.3 Error in the joints of left arm when the motion is executed using the PWM-control mode (top) and the position-control mode (bottom). . .	81
4.4 Error in the joints of right arm when the motion is executed using the PWM-control mode (top) and the position-control mode (bottom). . .	82
4.5 Block diagram of gravity-compensation controller.	83
4.6 Data-flow diagram of the balance controller implemented in balance-daemon.	86
4.7 Communication between various processes in hubo-motion-rt [57]. . . .	89
5.1 Schematic depicting data flow when using hubo-read-trajectory.	93
5.2 Schematic depicting data flow when using hubo-neck.	95
5.3 Screenshot of Hubo-init in use to home all the joints.	99
5.4 Screenshot of Hubo-init in use to monitor the robot joints.	99

Figure	Page
5.5 Screenshot of Hubo-init in use to monitor the sensor values.	100
5.6 Working of hubo-init.	100
5.7 Schematic depicting execution of gravity-compensation based trajectory following.	101
5.8 Schematic depicting how using hubo-ach, ROS can be used to control humanoid robot.	102
6.1 Force/torque sensor data when the rail is grasped (hand closed) and when it is not (hand open) during the climbing of the first step of the ladder.	107
6.2 (a) Parameter specification of a 3D ladder model. (b) Point-contacts and their normals (red arrow). A support polygon (green region) can be calculated based on the contacts to check the stability of the robot.	109
6.3 Three steps in motion planning for ladder climbing based on motion primitives. If one step fails, the planning process will trace back to the previous step. Prior information is being used to assist in each step.	112
6.4 Seed examples for motion planner- from left to right: placeHands, liftL-Foot, liftRFoot.	112
6.5 Snapshots of DRC-Hubo robot climbing stairs in the Armoury, Drexel University, Philadelphia [84].	117
6.6 Snapshots of DRC-Hubo robot climbing a ship-ladder [80,85,86].	118
6.7 Snapshots of DRC-Hubo robot climbing the DRC ladder [87].	119
6.8 Snapshots of DRC-Hubo robot climbing the industrial ladder during the DARPA Robotics Challenge- Trials 2013 [88].	122
7.1 Architecture of hubo-ach.	124
7.2 Architecture of hubo-motion-rt.	125

ABSTRACT

Paldhe, Manas M.S.E.C.E, Purdue University, May 2014. Software architecture and development for controlling a Hubo humanoid robot. Major Professor: C.S. George Lee.

Due to their human-like structure, humanoid robots are capable of doing some complex tasks. Since a humanoid robot has a large number of actuators and sensors, controlling it is a difficult task. For various tasks like balancing, driving a car, and interacting with humans, real-time response of the robot is essential. Efficiently controlling a humanoid robot requires a software that guarantees real-time interface and control mechanism so that real-time response of the robot is possible. Additionally, to reduce the development effort and time, the software should be open-source, multi-lingual and should have high-level constructs inbuilt in it.

Currently Robot Operating System (ROS) and Microsoft Robotics Developer Studio (MRDS) are most commonly used software packages for controlling robots. Since ROS uses Transmission Control Protocol (TCP) for inter-process communication, the latency in communication is high. Therefore, if ROS is used, the robot cannot respond in real-time. On the other hand, MRDS is not an open-source but a proprietary software package. Therefore it cannot be optimized for a particular robot. Thus, there is an urgent need to develop a real-time, open-source, modular, and thin software for controlling humanoid robots. This thesis describes the design and architecture of two software packages developed to fill this gap.

It is expected that in the near future a large number of humanoid robots will be used all around the world. The humanoid robots will be used to perform various tasks. The developed software packages have the potential to be the most commonly used software packages for controlling humanoid robots. These packages will assist

humans in controlling and monitoring humanoid robots to perform search-and-rescue operations, explore the universe, assist in household chores, etc.

1. INTRODUCTION

The technological advances during the Information-Technology Age, have fostered a growing interest in humanoid robotics. This has led to development of wide variety of humanoid robots like Honda's ASIMO, Rainbow's Hubo2+, Boston Dynamics's Atlas etc. These humanoid robots are capable of performing various locomotion and manipulation tasks like walking on uneven terrain, climbing up ladders, performing synchronized dancing, assisting elderly people in simple chores, playing sports etc. The development of humanoid robotics is fueled by the dream that humanoid robots will have a place as utility products assisting humans in daily life. Since the humanoid robots can traverse in environments where the wheel-based robots cannot reach, the utility of humanoid robots significantly increases. Assisting in search-and-rescue operations, performing tasks in hazardous places, and performing manufacturing and assembly tasks are some examples where humanoid robots could be used in the near future.

The concept of robots has been around since a long time. However, it was only in the 20th century, that, the robots first came into existence. By 1970's robots were being used to automate the manufacturing processes. From using robots to do repetitive and simple tasks back then, today, manufacturing companies use robots to perform tasks that require great precision, accuracy, and consistency [1]. As is depicted in Fig. 1.1 and Fig. 1.2, robots with wide variety of designs are used. Robotic arms, all-terrain vehicles, drones and robots that look like humans (called humanoid robots) are a few standard designs. These robots are used to manufacture and assemble cars, perform medical surgeries, explore the universe and even assist in search-and-rescue operations. Some roboticists believe that the development of humanoid robots is currently parallel to the development of personal computers in the late 1970s. Delivering packages, performing search-and-rescue operations in inhospitable areas, taking care

of elderly and disabled human beings are some ways robots will be used in the future (Fig. 1.2).



Fig. 1.1.: Robots being used in a wide range of applications like manufacturing, surgeries, space exploration and assisting in rescue operations. [2–5]

Research in robotics envisions robots incorporating artificial intelligence, emotional quotient, locomotion, and manipulation capabilities so that the robots can move, perform, ‘feel’ and ‘think’ like humans. A few years ago, this concept existed only in fiction. However, research in the field of humanoid robotics has brought this concept close to reality. A humanoid robot is a robot with its body shape built to resemble that of the human body. So apart from being used in the above applications, humanoid robots are also used to simulate human motions and thus contribute in the field of bio-mechanics. Researchers working with humanoid robots, in turn, borrow strategies and techniques that humans use to accomplish tasks, and apply them to the humanoid robots. Due to their similarity with humans, humanoid robots can do



Fig. 1.2.: Depiction of how robots will affect daily life in the future. [6–9]

most of the tasks that a human can do. Therefore, in dangerous and unsafe situations like search-and-rescue operations, firefighting, and military exercises, humanoid robots can be used, instead of endangering human lives.

A robot consists of many rigid links connected to each other. The connection between two links is called a joint. A joint provides a robot with one degree of freedom. A robot with n joints is said to have n degrees of freedom. Each joint is controlled using a electrical, hydraulic or pneumatic actuator. The motion of a joint can be angular rotation or linear translation. The more the degrees of freedom a robot has, more configurations can the robot be in. Thus, the complexity of a robot is generally proportional to the number of degrees of freedom. With advancement in

technology, the complexity of robots has significantly increased. Not only the number of degrees of freedom, but also, the type and number of sensors used in a robot, have considerably increased with time. Humanoid robots are no exception to this.

Initial humanoid robots were two-legged robots without an upper-body. Researchers were trying to make the robots walk like humans and by 1994, robots could walk over uneven surfaces [10, 11]. Biped robot Waseda-Hitachi Legs-11 (WHL-11), developed by cooperation between Waseda University and Hitachi in 1985, had 12 degrees of freedom. WHL-11 robot was state-of-the-art at that time. Today, human-sized humanoid robots like Hubo2+ have 40 degrees of freedom [12, 13]. Similarly, the number of sensors in the robot have increased significantly. For example, the number increased from one inertial measurement unit (IMU) in WHL-11 to four force/torque sensors and three IMU sensors in the Hubo2+ humanoid robot. Cameras and 3D scanners like Kinect, which were not even a part of robots back in 1980, are an important part of robots today. Robots use these 3D scanners to identify objects, navigate around obstacles [14, 15] and by understanding human emotions, interact with humans [16].

Today there are a various models of full-scale humanoid robots. Hubo2+ robot developed by Rainbow Company [12, 13, 17], Atlas and PETMAN robots developed by Boston Dynamics [18–21], ASIMO developed by Honda [22, 23], Valkyrine and Robonaut developed by NASA [24, 25], and NAO humanoid robot developed by Aldebaran Robotics [26, 27] are some examples. Researchers have demonstrated how these robots can be used to accomplish various tasks. Research is being carried out so that humanoid robots can be used to assist humans in disaster relief operations [28–30], play sports like soccer and baseball [31–35], and even assisting humans in daily tasks like cooking pancakes [36] and making coffee [35]. Figure 1.3 shows Topio robot playing table-tennis, Okonomiyaki robot flipping pancakes, NAO humanoid robot playing soccer, and Rollin Justin (RJ) humanoid robot catching a ball. According to Honda, the ASIMO robot is ready to work as a receptionist [23]. In addition to this, researchers are also working to develop algorithms so that humanoid robots can

perform cooperative tasks with humans or other humanoid robots [37–39]. Since robots are expected to work in close proximity with humans, it is necessary that they understand human emotions. Therefore researchers are also trying to make the robots more socialable [40–43]. Humanoid robots are not only expected to interact with humans and help them on the Earth, but also, assist them explore the universe, thereby reducing the risks of endangering the lives of astronauts. Therefore they are also being used in space operations. Robonaut developed by NASA has been tested at the International Space Station (ISS). [24, 25, 44]



Fig. 1.3.: Humanoid robots performing different tasks. [45–48]

Controlling a robot with such a large number of joints and sensors is a difficult task. Robust communication channels and protocols, which have minimum communication delay have to be developed and implemented. Manipulating a robot with so many degrees of freedom also requires designing and implementing complex planning

and control algorithms. Durán and Thill explained some of the challenges faced by researchers working in the field of humanoid robotics [49]. Usage of actuators that can mimic the functioning of human muscles and cartilage, performing whole-body motions like jumping, running, crawling, etc., and cognition by robots are some areas in which significant amount of research is required. Since these robots are expected to assist in disaster relief, play sports, cooperate with humans and perform other difficult tasks, it is necessary that the planning and control algorithms are robust, efficient, and should be executed in real-time.

1.1 Motivation

It is evident that humanoid robots will be assisting humans in the near future. These robots need a robust and reliable control software. Developing a different control software specific to a robot is a cumbersome task. Similarity in the structure of humanoid robots leads to similarity in the architecture of control software as well. In order to reduce effort required by researchers, it is necessary to ensure that there exists a generic software that can be used for controlling humanoid robots. However, since each robot is different, it should also be possible to optimize the software for a particular robot. Therefore, the source code of the software should be available to public (open-source software). Last but not the least, using the software, the robot should be able to perform various tasks which may require real-time response. Thus an ideal software for controlling humanoid robot should be real-time, generic, and open source.

A wide range of software packages exist for controlling robots. Robot Operating System (ROS) developed by Willow Garage in 2007 and Microsoft Robotics Developer Studio (MRDS) developed by Microsoft in 2006 are two such most used software packages. Numerous robot manufacturers provide ROS packages for their robots. PR2, a personal robot developed by Willow Garage, NAO humanoid robot, and UBR-1 developed by Unbounded Robotics are some robots for which ROS packages are

already provided by their manufacturers. MRDS was used by Princeton University for the Defense Advanced Research Projects Agency (DARPA) Urban Grand Challenge in 2007. Lego Mindstorms, iRobot, and KUKA robots are all supported by MRDS. MRDS was also used by the popular social networking website MySpace for non-robotic back-end application.

Humanoid robots are expected to perform several tasks, which may require real-time response. Driving a car, balancing itself, playing sports, interacting with humans and performing critical manipulation tasks like surgeries, etc. are some examples where real-time response is critical. For a humanoid robot to successfully complete these tasks with reliability, the control software has to be real-time with low latency. Therefore real-time operation is the most important feature required in any humanoid-robot control software. Due to the difference in hardware of robots, control or planning parameters optimized for one robot, say Hubo2+ robot (which has electric actuators) may not be ideal for other humanoid robots like Atlas (which has hydraulic actuators). Therefore, a generic control software should also be flexible so that parameters and algorithms for control can be appropriately chosen and modified. Besides, understanding the implementation of the features and algorithms also helps the researchers to correctly interpret the results. Thus, it is very useful if the software package used is open-source.

ROS, though a very powerful tool for robot control, is not a real-time software package. Since a robot's real-time response cannot be compromised, ROS is not suitable for humanoid-robot control. On the other hand, MRDS is not open-source and so the researchers cannot optimize the internal framework for the robot.

Neither of the two widely used robot controlling software packages are suitable for controlling humanoid robots. So, there is an urgent need to design and develop a software package that is optimized for controlling humanoid robots to perform various tasks.

1.2 Related work

Open Robot Control Software (OROCOS) was among the first open-source robot control software packages. But, after the release of other robot control software packages like Robot Operating System (ROS) and Microsoft Robotics Developer Studio (MRDS), use of OROCOS has significantly reduced. Currently, the two widely used robot control software packages are ROS and MRDS. Although powerful, yet, as discussed earlier, ROS and MRDS are unsuitable for controlling humanoid robots. However, study of their architecture and design principles, helps us develop new control software for humanoid robots. This section discusses the general idea behind developing these robotics software packages.

ROS is a software framework for robot software development, providing operating-system-like functionality and can be used on a single computer or a computer cluster. ROS was originally developed in 2007 by Willow Garage. More than twenty institutions are collaborating in a federal development model to further enhance the features of ROS. It was built with the following core philosophies [50]:

- **Peer-to-peer communication:**

Complex robots with multiple links may have multiple on-board computers connected via a local area network (LAN). Running a central server would result in bottlenecking one particular link. Thus, using peer-to-peer communication would avoid the issue of bottlenecking.

- **Multi-lingual:**

A commonly used robotic platform should be able to cater to the preferences of all the developers. The programs for a particular application may be more easily developed in one language than other. Thus a widely-used robot control software should be multi-lingual. Thus, currently ROS supports four languages with very different paradigms: C++, Python, Octave, and LISP.

- **Tools-based design:**

ROS is designed such that a number of tools are built and run as various ROS

components. These tools perform various tasks like visualizing peer-to-peer communication topology, measuring bandwidth usage, auto-generating documentation, etc. Some services like global clock and logging module are still part of the core of the ROS. Though this architecture reduces efficiency, the stability and reliability of software are significantly higher.

- **Thin development model:**

Robotic software projects often have a lot of overlapping algorithms and drivers. Identical algorithms for inverse kinematics have been implemented in many robotic software packages. Makareno, et al. described the advantages and the need for the software packages to be thin [51]. A thin software program exemplifies code reusability: not only does it re-utilize existing code from various packages, but parts of thin software can themselves be re-utilized. ROS encourages development of software as packages that are stand-alone libraries. ROS itself re-uses codes from numerous open-source projects, thus making it a good example of a thin software package.

- **Free and open-source:**

Being open-source platform increases the chance of detecting bugs and thus fixing them across the whole project. Without being open-source, the project cannot support other projects following the thin philosophy. Therefore, ROS is made open-source and free.

The philosophy of having peer-to-peer communication, multi-lingual support, being thin, free and open source are all applicable to humanoid robotics. However, the tools-based design is a drawback that significantly hurts the real-time performance of a humanoid robot. For real-time applications, the controller must servo the joints motors as fast as possible. Reduction in efficiency directly affects the frequency of the controller and thus the real-time response of the robot. Though ROS cannot be directly used for humanoid robot control, some of the design philosophies should be followed to develop new software for humanoid-robot control.

MRDS on the other hand has a different set of advantages. The most important advantage of MRDS over ROS is that MRDS is a real-time software package. Apart from being real-time, MRDS also provides a visual programming tool, an inbuilt 3D simulation package (which utilizes hardware accelerator) and packages for easy interface with the robot's sensors and actuators. The core features that MRDS is based on are [52]:

- **Concurrency and coordination:**

Robots by nature perform a lot of computations in parallel. Path planning, sensing, motion control, etc, are all done in parallel in a complex robot. Thus, it is necessary for robot control software to support parallel operations. Using Coordination and Concurrency Runtime (CCR) library, MRDS abstracts the memory locking and communication amongst various operating processes.

- **Distributed messaging:**

MRDS utilizes Decentralized Software Service Protocol (DSSP) to pass messages between different services with minimal overheads. This results in much quicker communication than traditional messages.

- **Simulation:**

A powerful 3D simulator is included as a part of MRDS so that the developer can check the controller or motion planner first using computer simulation before running it on the actual robot. Various objects can be easily created to modify the environment. Events like collisions and concepts like gravity are implemented in simulation using Ageia physics engine. A physics engine is necessary to model dynamic interactions and compute expected trajectory of the objects under consideration.

- **Programmer interaction:**

MRDS includes visual programming language for easy development. Hobbyists with no programming experience, using the visual programming language, can

develop and design control algorithms for a robot and use the simulator to evaluate its performance.

Since MRDS is not open-source, the implementations of the software cannot be directly picked up. However concurrency and coordination are necessary features and have to be incorporated in the new software packages. Inbuilt simulation is not a necessary feature as open source software packages for simulation like OpenRAVE [53] and Klamp [54] can be easily integrated with any robot control software. Similarly, visual programming language is not required, as at least in initial stages only experts in the field of robotics are the intended users, and they are familiar with programming.

OROCOS [55, 56] is among the first few generic real-time robot control software packages. It was developed along the following ideas:

- **Open source:**

OROCOS was built with the idea of being open source. By being open-source bugs can be detected and features can be easily added to a software. This also helps researchers understand the new algorithms being implemented and use them to improve the performance.

- **Modular and flexible:**

OROCOS is built to be modular so that researchers can easily develop new packages. Other researchers can then easily integrate the existing packages thus reducing effort required to run a robot. Flexibility helps researchers modify a certain section of the software to optimize its working for a particular robot.

- **Multi-lingual:**

OROCOS, like ROS, is multi-lingual. So, researchers can write a controller in language they are most comfortable in leading to considerable reduction in the development time.

The modularity is obtained by having multi-process architecture. However, ORO-COS does not provide multiple processes to publish to shared memory. Publishing

to shared memory is essential as different processes may want to send motion commands to a single process that controls the motors and actuators. A good example is the motion of upper-body of a humanoid robot. During normal operation, the process that manipulates those joints should send commands to the main control process. However in critical situations like the robot going off balance, it is essential that the process that ensures the balance of the robot controls the upper-body joints. Therefore, OROCOS is not suitable for controlling humanoid robots.

For different reasons, neither of the existing software packages suit the requirement for controlling humanoid robots. ROS is not real-time, MRDS is not open source, and OROCOS does not support multiple publishers. However, each of them has its own advantages. So a novel software package, which has the advantages of all of them was developed. The principles on which the new software packages are built are:

- Modularity and real-time coordination
- Open-source software
- Thin development
- Multi-lingual

Table 1.1: Design philosophies of Hubo-ach, ROS, and MRDS.

Software package	ROS	MRDS	Hubo-ach
Real-time	No	Yes	Yes
Open-source	Yes	No	Yes
Multi-lingual	Yes	No	Yes
Communication	Peer-to-peer	Distributed messaging	Peer-to-peer

1.3 Contributions

In Section 1.2, three widely used robot control software packages were discussed. The design philosophies behind their implementation were discussed. That laid the foundation to discuss the advantages, disadvantages, and shortcomings of the three software packages. Since it was clear that the existing packages are not suitable for controlling humanoid robots, the decision to develop a software package was taken. Based on the analysis, the requirements for the new software package were stated. Modularity and real-time coordination, multi-lingual, being open-source, and supporting thin development are necessary characteristics that the software should support.

The only shortcoming of using ROS for controlling humanoid robots is that it is not a real-time software. This section discusses the reason for the shortcoming. ROS has a tools-based design; which means that instead of having one main control loop, ROS has a number of tools and components. Each component is responsible for a particular objective. This approach is followed to make the controller more modular and less reliant on dependencies. When developing a controller for a robot, same principle is followed. For instance, one process will be responsible for executing computer vision algorithms, another will be responsible for executing planning algorithms and the third implements a controller that takes in the desired joint angles and sensor feedback to generate smooth motion of the robot. In order to implement such a multi-process tool based design architecture, communication between various processes needs to be robust. To ensure robust communication between multiple processes that may execute on one or multiple computers, ROS uses Transmission Control Protocol (TCP). Although TCP is optimized for accurate delivery, yet, the delivery time limit is not bounded. This drawback of TCP adversely affects the performance of ROS. In ROS, as the inter-process communication (IPC) is based on TCP, therefore, when one process is sending out data to other process, there is no bound on the maximum time it takes to synchronize the data. Also due to the TCP protocol, even

if newer data is available and is ready to be sent, older data will be delivered first. For control algorithms, the latest sensor data is much more important than older sensor data. Thus, TCP is not the best choice for inter-process communication in a robot.

To overcome this drawback, a new robot control software that incorporates all the advantages of ROS has been developed [31, 57]. But, unlike ROS, the software has low-latency communication between processes. To ensure low latency *non-blocking* memory share algorithm has to be implemented. A naive way to implement memory sharing is by using locks. A process when accessing shared memory ‘locks’ it. Thus other processes cannot write to it and corrupt the data. The process releases the lock when it has completed the memory operations. This algorithm however can lead to a situation wherein a process is unable to access the memory. Another possible problem that can arise is called deadlock. Deadlock is a situation in which two competing processes wait for the other to finish. For example, process A updates row 1 and then row 2 and process B updates row 2 and then row 1. Process A cannot start updating row 2 until process B is finished. Similarly process B cannot finish updating row 1 until process A is finished. Thus, there is a deadlock and neither of the two processes will ever finish. A non-blocking algorithm ensures that processes accessing a shared memory block do not have to wait indefinitely. By using non-blocking algorithms, data sharing across multiple processes can be done in a limited amount of time.

Using a non-blocking memory sharing algorithm provides an upper bound on the amount of time for inter-process communication. The software is also designed to share the latest data with other processes. Thus, using these two major changes in the software architecture, it is ensured that the designed software is real-time and suitable for humanoid-robot control. The time required to share complete information of the robot (sensor readings, desired joint position, errors and warnings) across various processes is measured to be 0.011 ms with standard deviation of 0.0033 ms as opposed to 1.002 ms with standard deviation of 0.287 ms when using ROS.

This thesis explains the design of two software packages developed with the above architecture. Hubo-ach is a low-level humanoid robot control software. Hubo-motion-

rt is a high-level humanoid robot controller with more abstract features. The reasons why various design decisions were taken are also explained. Based on the design, the implementation of the software packages and the usage for Hubo2+ humanoid robot is discussed. This thesis also explains how these packages can be used for controlling other humanoid robots.

Experiments to make a humanoid robot climb a ladder were performed using the developed software packages. This thesis also shows that a full-scale humanoid robot has the ability to climb industrial ladders in uncontrolled outdoor environment. The robot can counter the effects of sunlight and fluctuating wind speeds to successfully climb industrial ladders. Though the ability of humanoid robots to climb ladders has been demonstrated earlier, those experiments were performed in computer simulation or controlled indoor environments [58–60].

This thesis provides a design and implementation of robot control software that is real-time, multi-lingual, open source, and supports thin development. This software package is a significant improvement over the existing robot control software packages and is the most suitable software for controlling humanoid robots. Do note that these software packages were developed in collaboration with Humanoid Robotics Laboratory, Georgia Institute of Technology, and Drexel Autonomous Systems Laboratory, Drexel University. We have contributed in part to the developed software packages.

1.4 Impact

The number of humanoid robots being used all over the world is increasing at a fast rate. Competitions like DARPA Robotics Challenge (DRC) and RoboCup are promoting research and development of humanoid robots. This is increasing the participation of engineers and scientists to push forward the capabilities of humanoid robots. With a large number and variety of humanoid robots, using a common robot control software can significantly avoid duplication of effort. New algorithms and

controllers can be tested across all the robots. By being open-source, the algorithms and controller can be made more robust and efficient.

This thesis explains the design and development of two generic software packages for controlling humanoid robots. Just like ROS is most widely used software for controlling robots, these software packages have the potential of being used by a large number of researchers working on humanoid robots. Since the software packages are based on similar philosophies and are real-time software packages, they significantly improve upon ROS. Being open-source and supporting thin development philosophy, these software packages also reduce the development time and reduce duplication of effort. These packages can also use ROS and thus all its features. Thus, they have the potential to be the most commonly used software packages for controlling humanoid robots and to help reduce the effort of a lot of researchers who are striving to push forward the frontiers of the field.

Today, mobile phones have become a part of our daily lives. Android operating system, by being the most common operating system have greatly impacted our lives. Similarly, in the future, it is expected that humanoid robots will become a part of our daily lives. The robots will assist us in complex task like space exploration, disaster response, and will also prove to be able human companions like being an assistant, a sports companion etc. Like Android affects us today, in the future the developed software packages have the ability to indirectly simplify and improve the standard of human lives by being the standard and most used software for controlling humanoid robots.

1.5 Organization of the thesis

This thesis is organized into seven chapters. Chapter 2 provides an overview of the Hubo2+ humanoid robot. It discusses the hardware architecture of the robot. The Hubo2+ humanoid robot is used as an example to explain the software architecture. Therefore understanding the hardware of the robot is essential. Chapter 3 discusses

the software hubo-ach in detail. The purpose of hubo-ach, various structures used to store data, the inter-process communication software ‘ach’ and the internal structure of hubo-ach are discussed. Chapter 4 discusses the high-level controller ‘hubo-motion-rt’. Its purpose, structures used, and how it builds on hubo-ach are discussed. Chapter 5 discusses the application of hubo-ach and hubo-motion-rt for the development of custom-software packages. Software packages like “hubo-read-trajectory” and “hubo-init”, which are developed using hubo-ach and hubo-motion-rt are discussed. How these software packages can be used for controlling other humanoid robots is also discussed. Chapter 6 discusses some experimental work carried out with the developed software on two versions of Hubo humanoid robots: Hubo2+ and DRC-Hubo robot. Chapter 7 is the conclusion of the thesis. The chapter analyzes the developed robot control packages. It also discusses future work that needs to be carried out.

2. OVERVIEW OF HUBO2+ HUMANOID ROBOT

2.1 Introduction

Chapter 1 provided with the motivation and the need to develop a new software for controlling humanoid robots. Existing software packages like ROS and MRDS were discussed. Having analyzed these software packages, a novel software for controlling humanoid robots was designed. This software is based on some of the philosophies of existing software packages like ROS, but is more suitable for controlling humanoid robots. Though the developed software is generic and can be used for controlling any humanoid robot, throughout this thesis, a Hubo2+ humanoid robot is utilized as an example.

Before developing the software or deciding which software should be used for controlling a robot, it is necessary to understand the hardware specifications of the robot. Without properly understanding the hardware architecture, a developer cannot correctly determine what software to use. Since Hubo2+ humanoid robot is used as an example robot when discussing the software architecture and design, in this chapter, the hardware of Hubo2+ humanoid robot is discussed.

This chapter first discusses the kinematic structure of the robot. The complete physical structure of the robot is provided in Section 2.2. Section 2.3 describes various hardware modules used in the robot, the reasoning behind using those particular modules and the modes and features that those modules support. In Section 2.4 the design principles and decisions to integrate all the modules into one system are explained. It also explains how software running on the on-board computers communicates with various hardware modules. Section 2.5 summarizes the chapter and delineates how this chapter helps in understanding the rest of the thesis.

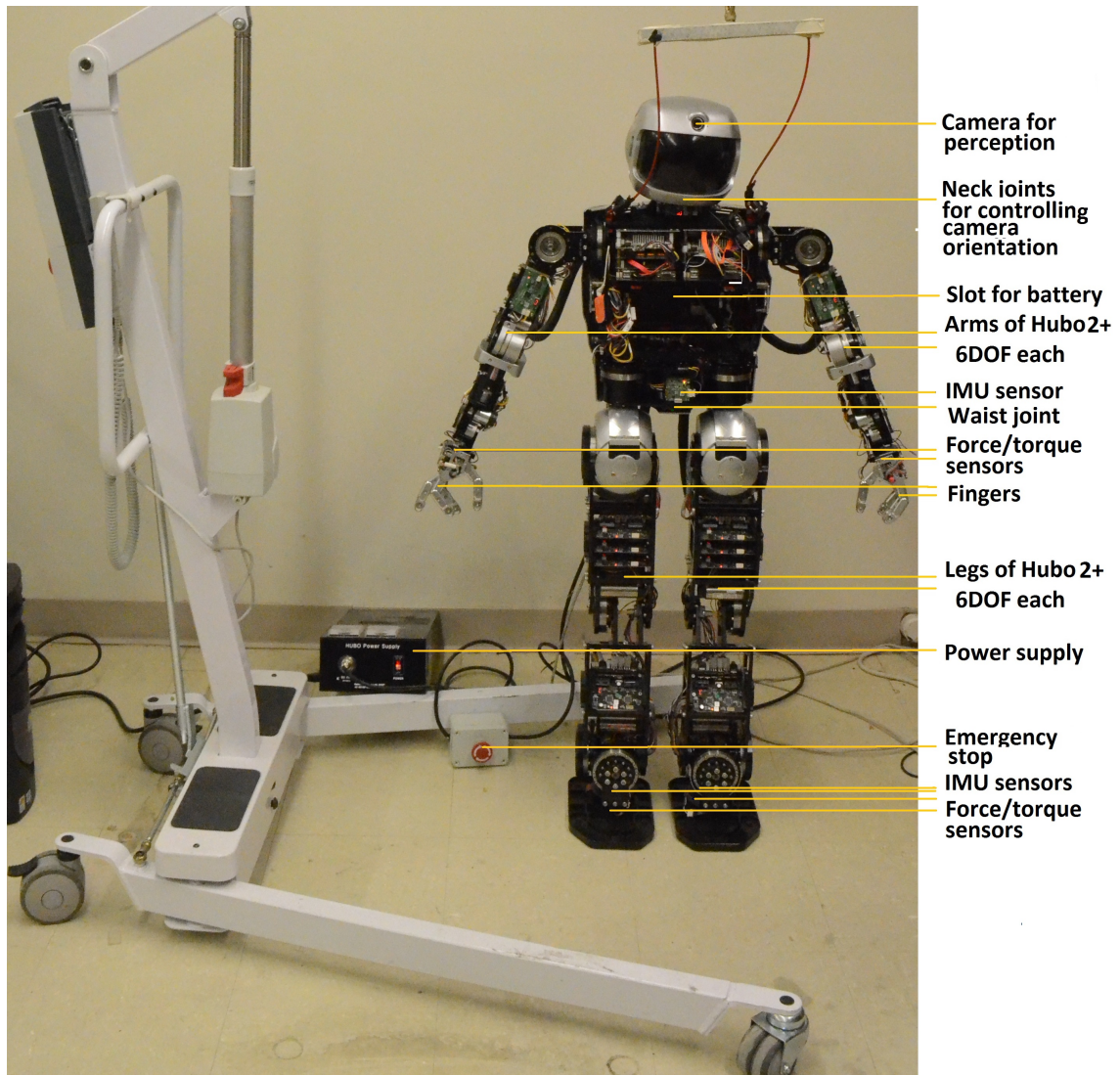


Fig. 2.1.: A picture of the Hubo2+ humanoid robot showing the location of various electronic modules.

2.2 Mechanical structure and representation of Hubo2+ robot

The Hubo2+ robot was designed by a team led by Prof. Jun-Ho Oh at the Hubo Lab in Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea. The robot is an upgrade over the KHR series of robots. Park, Kim et al. have provided a complete description of the mechanical design of KHR-3 humanoid robot

(commonly known as Hubo) [12,13]. In spite of being human-sized, Hubo2+ robot is very light and weighs just 43 kg. Table 2.1 provides a description of various modules of the robot. The engineering design of the robot is shown in Fig. 2.2.

Hubo2+ robot is a full-scale humanoid robot. It has a total of 38 degrees of freedom (DOF) (6 DOF in each leg, 6 DOF in each arm, 10 DOF in fingers for grasping, 1 DOF for waist rotation and 3 DOF for neck joints for controlling camera orientation), 4 three-axis force/torque (f/t) sensors and 3 inertial measurement units (IMUs). The locations of all these sensors are shown in Fig. 2.1. A force/torque sensor measures the normal force applied onto it and the moments about its x and y axis. An IMU unit measures the acceleration along and angular velocity about each axis. When the robot is stationary, the IMU sensors can be used to compute their inclination with ground [61]. Apart from the sensors, the robot is equipped with 2 on-board computers for perception. One computer called the body-computer, communicates with the motors and sensors through Controller Area Network (CAN) and the other computer, called the head-computer, is used for vision processing, and connects to an RGB camera. The electronic modules, their usage and features are discussed in detail in Section 2.3.

Table 2.1: Specification of Hubo2+ humanoid robot in ARTLab.

Parameter	Value
Height	1.30m
Weight	43kg
Number of non-finger joints	28
Number of finger joints	10
Number of IMU sensors	3
Number of Force-Torque sensors	4
Perception sensors	RGB camera
Input power	DC: 57V, 27A

In general, the kinematic structure of a robot is modeled using Denavit-Hartenberg (D-H) representation. Jacques Denavit and Richard Hartenberg proposed this convention system in 1955 so that the robot coordinate frames could be standardized. Fu, Gonzalez, and Lee have discussed how a robot can be represented using the D-H representation [62]. Though, the D-H parameters were originally proposed for single limb open-chain robots, the representation can be used for humanoid robots as well. The D-H representation of Hubo2+ robot is shown in Fig. 2.3.

The study of robot motion can be sub-divided into two categories. The first category is called kinematics. It is the analytical study of geometry of motion of a robot, without considering forces and moments that cause the motion. This category is further sub-divided into two sections called forward kinematics and inverse kinematics. Forward kinematics is used to find the position and orientation (pose) of the robot end-effectors, given all the joint angles. Inverse kinematics is employed to find what joint angles are required so that the robot end-effector is in desired pose. The second category, called dynamics deals with the forces and moments required to move a robot. This category is also divided into two sections. Forward dynamics is used when the desired force and torque at each joint is known. Using them the motion of the robot is estimated. The second category called inverse dynamics is used to compute torque and forces required to move a robot along a particular trajectory (that is, desired joint angles, velocities and accelerations are known).

For standardization, and so that other users can easily verify the representation, even a humanoid robot should be represented using the D-H representation. In Fig. 2.2, D-H representation of a Hubo2+ humanoid robot is depicted. A detailed explanation of the representation is provided in Appendix A. In case of a robotic arm, the base coordinate system of the robot is fixed with respect to ground. However, the Hubo2+ robot can walk around. So a base coordinate that is fixed with respect to the ground cannot be chosen. Instead a base coordinate that is equally close to every end-effector should be selected. Thus, the base coordinate system fixed to the torso

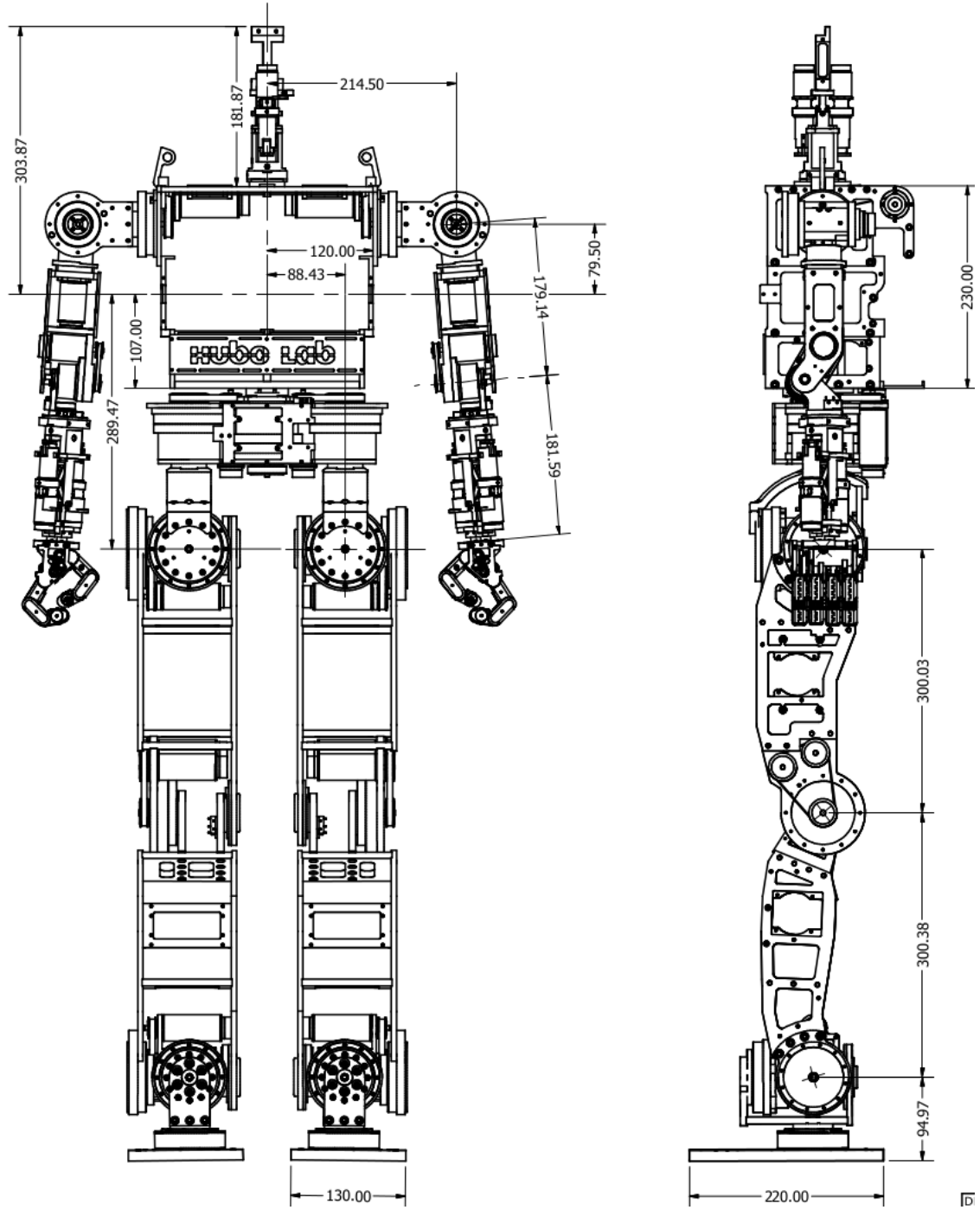


Fig. 2.2.: Engineering drawing of Hubo2+ humanoid robot showing some of the dimensions and size of different parts [31].

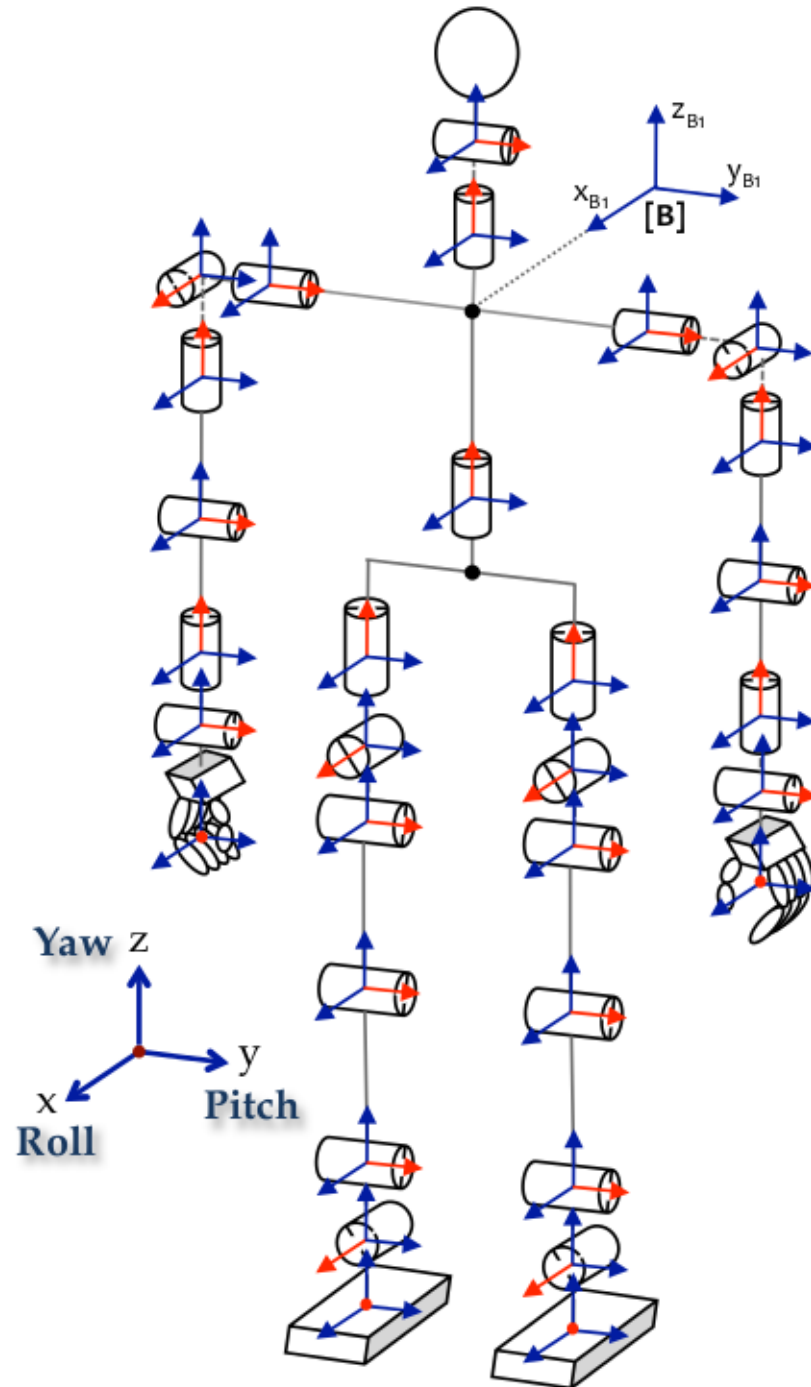


Fig. 2.3.: Denavit-Hartenberg representation of a Hubo2+ humanoid robot.

of the robot is chosen, and the forward and inverse kinematics calculations are done with respect to that coordinate frame.

Once the D-H representation of a robot is formulated, forward kinematics (obtaining the position and orientation of the end effector from current joint angles) can be easily done. Analytical closed-form inverse kinematic (IK) solution does not exist for all the robots. However, if three consecutive joint axes of a 6 DOF robotic arm intersect at a point or if they are parallel, then analytical closed-form IK solution can be obtained [63]. In the case of Hubo2+ robot, the arms have 6 DOF and the shoulder roll, shoulder pitch, and shoulder yaw joints of the arm intersect at a point. Similarly, the legs have 6 DOF, and the hip roll, hip pitch, and hip yaw joints intersect at a point. Thus analytical closed-form IK solution of each of the four limbs of the robot can be easily obtained. These solutions are included in Appendix A. Park, Ali, and Lee have described how these solutions are obtained [64, 65]. After formulating the D-H representation and obtaining the transformation matrices, the dynamic computations of the Hubo2+ robot can be done using Newton-Euler and Euler-Lagrange methods. To compute the torques required at the joints to move the robot recursive Newton-Euler computation method is employed. To compute the torques required at the joints so that desired force can be applied at the end-effector, Jacobian-matrix-based computations are used. Fu, Gonzalez, and Lee have discussed both methods in detail [62]. As in case of D-H representation, though these methods were initially developed for single limb open-chain robotic arms, they can be easily extended for humanoid robots. Appendix B discusses the formulation of the two methods for Hubo2+ robot.

2.3 Electronic and electro-mechanical modules of the Hubo2+ robot

In order to move a Hubo2+ robot, a number of different electronic and electro-mechanical modules are required. Motors or actuators are necessary for motion, controller boards are necessary for controlling the motors and encoders are required

for controller feedback. Apart from them, to make the robot completely autonomous, a variety of sensors like IMU for inclination calculations, force/torque sensors to detect external forces and contacts, proximity sensors to predict grasping, and cameras, 3D laser scanners, or kinect sensor for computer vision are required. The larger the number and variety of sensors a robot has, the more accurately it can model the environment around it. This section discusses various electronic and electro-mechanical modules present in Hubo2+ robot. The need for these modules and some of their salient features are discussed below.

- Computers for processing:

An on-board computer is essential for computer vision, trajectory planning and robot control. The computations for perception are independent of those for trajectory planning and control, but they are all equally computationally intensive. Therefore, instead of having one powerful computer, Hubo2+ robot has two smaller computers with equal or more computing power. One computer is dedicated to perception computations (and is called the head-computer) while the other is dedicated to trajectory planning and control (and is called the body-computer). In Hubo2+ robot, both computers have exactly the same specification. However, for faster I/O operations, the hard disk in the body computer of Hubo2+ robot at ARTLab has been replaced with a solid-state disk drive (Fig. 2.4).

- Motors and motor controller boards:

In Hubo2+ robot, all the joints are rotary. These joints are all controlled by Dynamixel's brushless DC (BLDC) motors. For robust control, these motors are attached to high accuracy (error < 0.001 radian) encoders. Apart from the ability to reach a particular position, for maneuvering objects, the motors should also be able to apply a set amount of force or torque. These Dynamixel motors support both modes of operation. To control the position of the motors or the torque that they apply, the motors are attached to joint motor controller boards

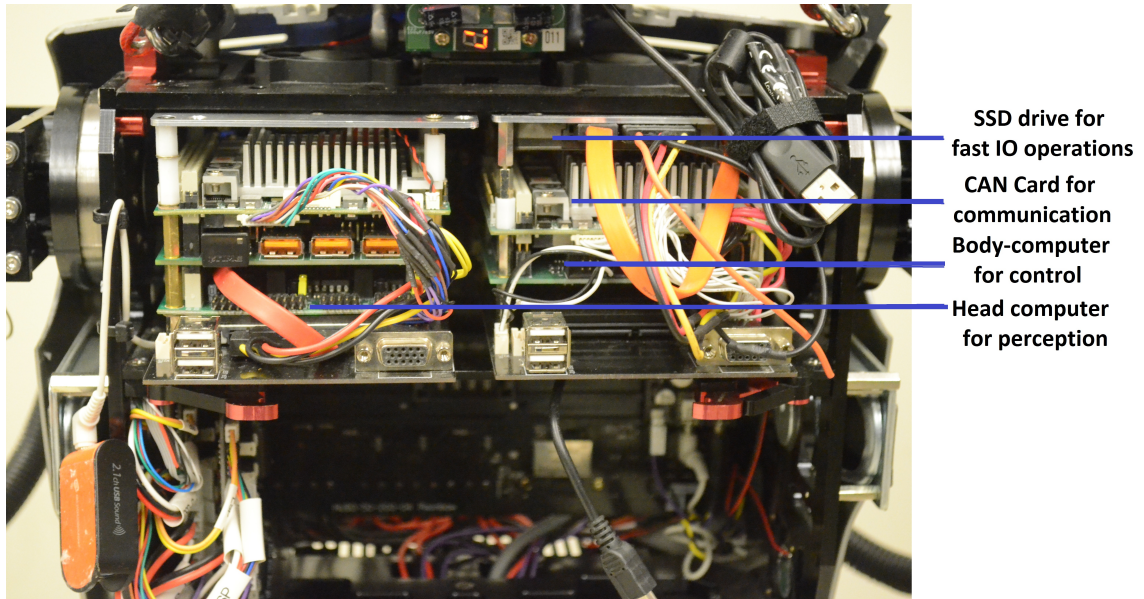


Fig. 2.4.: The two computers of Hubo2+ robot at ARTLab: (left) head-computer and (right) body-computer.

(JMC boards). These boards receive commands from the body-computer and execute them. When requested for the data, they also send the encoder feedback to the computer. Working of both these control mode is explained below:

- Position-control mode:

In this control mode, the JMC board is commanded to move the motor to a particular angle. The JMC board through an internal high gain proportional-integral-derivative controller (PID controller) moves the motor to desired position and maintains it there until the next command is obtained (Fig. 2.5). High PID gains are chosen as the hardware was designed with the philosophy that if the hardware does precisely what it is commanded to, then control feedback is not required in software [66].

- Compliance mode/PWM-control mode:

In this control mode, the JMC board is not sent a desired position, but is sent a desired pulse-width-modulation (PWM) value. PWM is a mod-

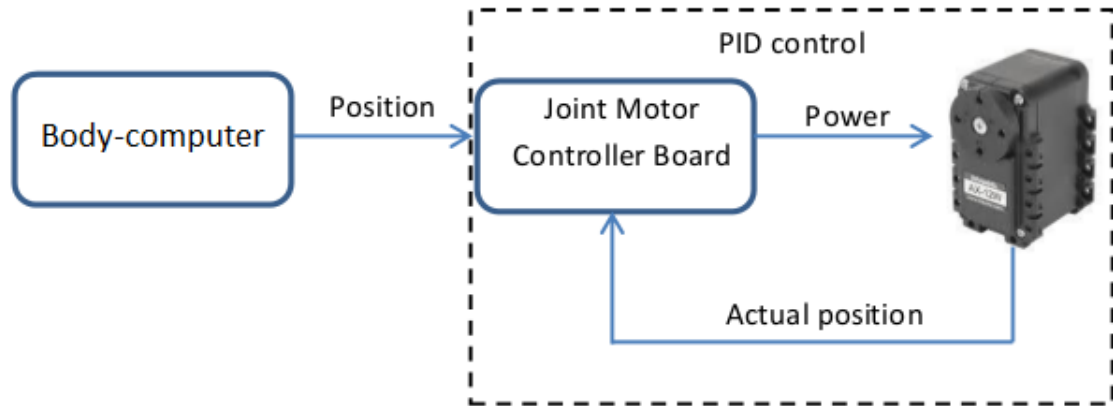


Fig. 2.5.: Depiction of how the position-control mode functions.

ulation technique used to encode information for transmission, mainly for controlling the power to electrical devices. The average voltage value over the PWM cycle is controlled by turning the signal line on and off. Longer the on-time is, longer is the power sent to the device. The ratio of on-time to the PWM cycle time is called a duty cycle.

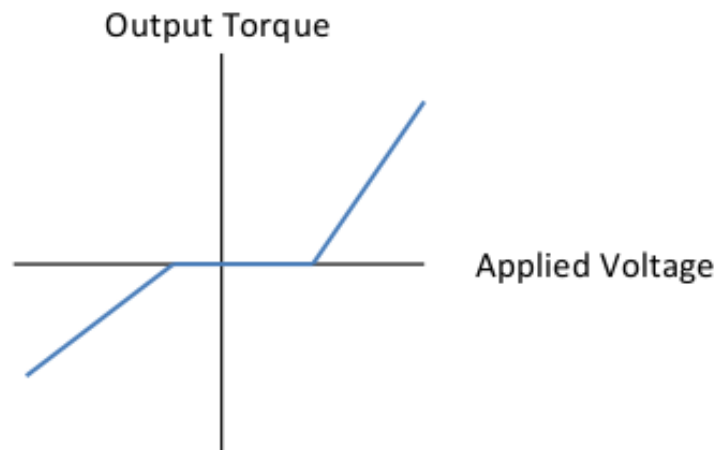


Fig. 2.6.: Depiction of how the PWM-control mode functions.

In the compliance mode of motor control, the on-board computers send a desired duty-cycle value to the JMC board. The JMC board applies pulses

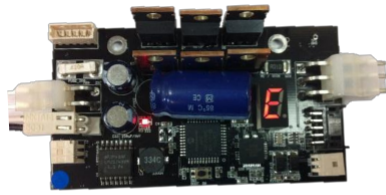
of that duty-cycle to the motor. This changes the effective voltage applied across the motor and thus the output torque. Thus, using this compliance mode a user can virtually control the applied torque (Fig. 2.6).

To control the torque, a look-up table that maps the applied duty-cycle value to the generated torque is created. This table is called Torque-PWM table. When a particular amount of torque is required, the Torque-PWM look-up table is scanned and the duty-cycle value that produces the required torque is found. This value is sent to the JMC board and thus the required torque is applied. Hubo2+ robot does not support PWM-control mode, thus only the normal position-control mode for robot motion can be utilized. In DRC-Hubo humanoid robot (upgraded version of Hubo2+ robot), either of the two modes for motor control can be utilized. Watanabe and Yuta have described how this mode can be implemented for a brushless DC motor [67].

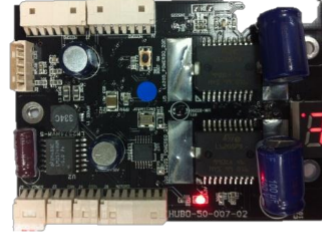
The torso of Hubo2+ robot has only one joint (waist joint) and has a single channel motor controller. The neck has three joints and so the motor controller for the neck is designed to have three channels. Arms and legs have 6 joints each. A six-channel motor controller board can be designed, but it will be huge and bulky. Thus three motor controller boards (2 channel each) are used. Since wrists and fingers have lower strength, a controller board with lesser output current is sufficient. Thus a more compact motor controller board is used for the wrist joints. Figure 2.7 shows different JMC boards used in Hubo2+ robot.

- Power Supply:

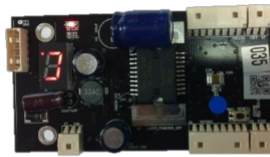
All the electronics and electro-mechanical systems need power for operation. Robots in general need to be mobile and thus, most of them run on a battery. Because of being heavy and having a large number of electronic modules, humanoid robots are power hungry. Although a battery can be used to power a humanoid robot, the on-time when using a battery is not high (less than an 1



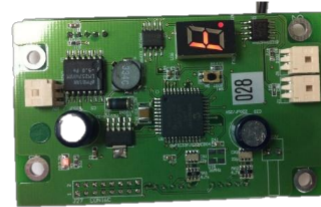
a) 1Ch BLDC motor controller for torso.



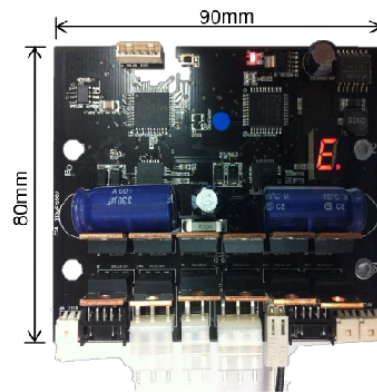
b) 3Ch DC motor controller for neck.



c) 2Ch DC motor controller for wrist.



d) 2Ch BLDC motor controller for elbow and shoulder yaw joints.



e) 2Ch BLDC motor controller for leg joints.

Fig. 2.7.: Motor controller boards used in Hubo2+ robot.

hour). Thus humanoid robots run on direct power supply as well as battery power.

Hubo2+ robot can run using either direct power supply or the battery. Hubo2+ robot comes with a power supply system that converts AC input (100-240V, 50-60Hz) into DC power supply (57V, 27A). The output is plugged into Hubo2+ robot via an emergency stop switch. If Hubo2+ robot endangers humans or itself, the emergency switch should be pressed (Fig. 2.8).

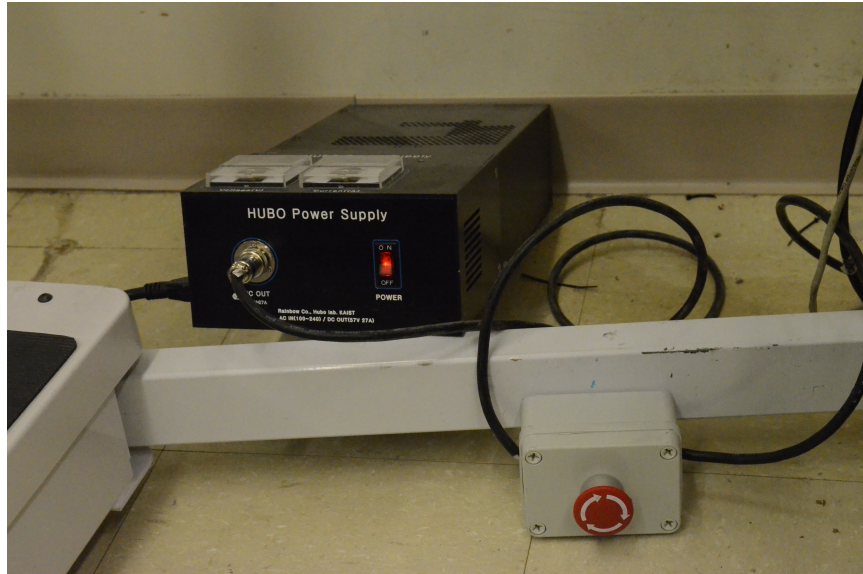


Fig. 2.8.: The power supply of Hubo2+ robot. AC input is provided to the robot.

For remote operations, Hubo2+ robot can be operated using a battery. It is a Lithium-Polymer battery. It supplies 48V and carries 7.8Ah of energy. The battery can provide a peak current of 60A and weighs 3.3 kg. Since there is no emergency stop switch, the robot should be operated with extreme caution when using a battery. Generally Hubo2+ robot can execute motions for approximately 50 minutes when running on the battery. Since the battery needs relatively large space, so it is placed at the torso of the robot (Fig. 2.9).

- Inertial Measurement Unit (IMU) sensors:

Humanoid robots can be considered as mechanical clones of humans. Humans using touch, and by sensing force, know how the ground is aligned and if they are falling. For the robot to know that it needs to have inertial measurement units. The IMU sensors measure the acceleration of the robot along various axes and the rotational velocity about them. The IMU sensors can be used to compute their inclination with respect to ground. Want et al. have described how the inclination is computed using adaptive kalman filter [61]. Thus, the

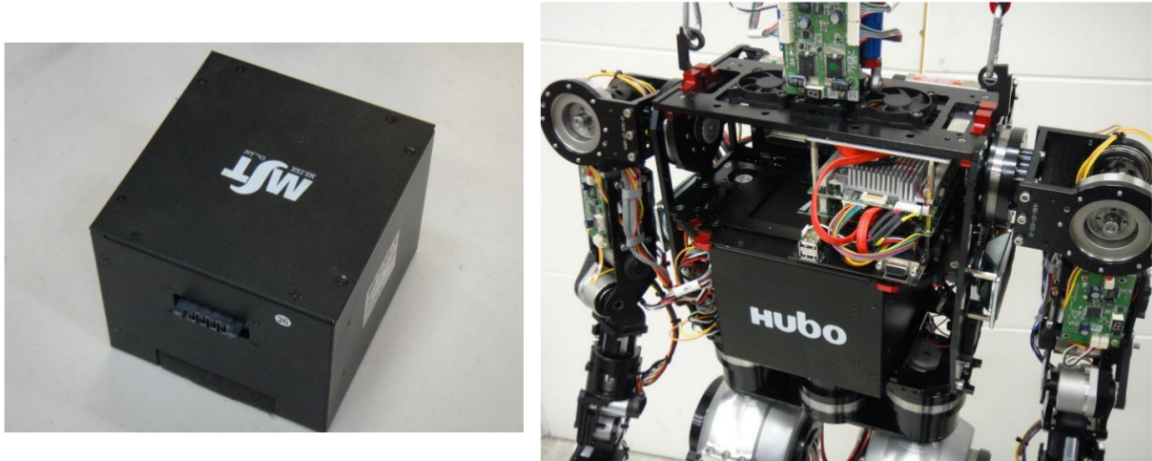


Fig. 2.9.: Battery used in the Hubo2+ robot.

IMU sensors mounted on feet inform about the support plane's elevation and the IMU sensor at the center of mass (waist) is required to keep the robot balanced. A photograph of the IMU sensor mounted on the waist of Hubo2+ robot is shown in Fig. 2.10. The data from the IMU sensors can also be used to compute the location of the robot with respect to its initial location. Golding and Lesh have described how the location of a human is estimated using the IMU sensors [68]. Same principles can be applied to a humanoid robot as well.

- Force/Torque sensors:

An important aspect of robotics is to maneuver objects. Precise manipulation often requires applying appropriate amount of forces. Pires et al have described the need of force/torque sensors in robots and how control for precise manipulation can be done using them [69]. Thus to know if the robot is applying required amount of force or not, force/torque sensors are required in the hands. The robot also needs the effective force and torque at the center of mass in order to balance itself. To know the effective force and torque at the center of mass, the forces and torques at the end effectors are required. Thus, for balancing, it is important to have force/torque sensors at the end-effectors. Hubo2+ robot

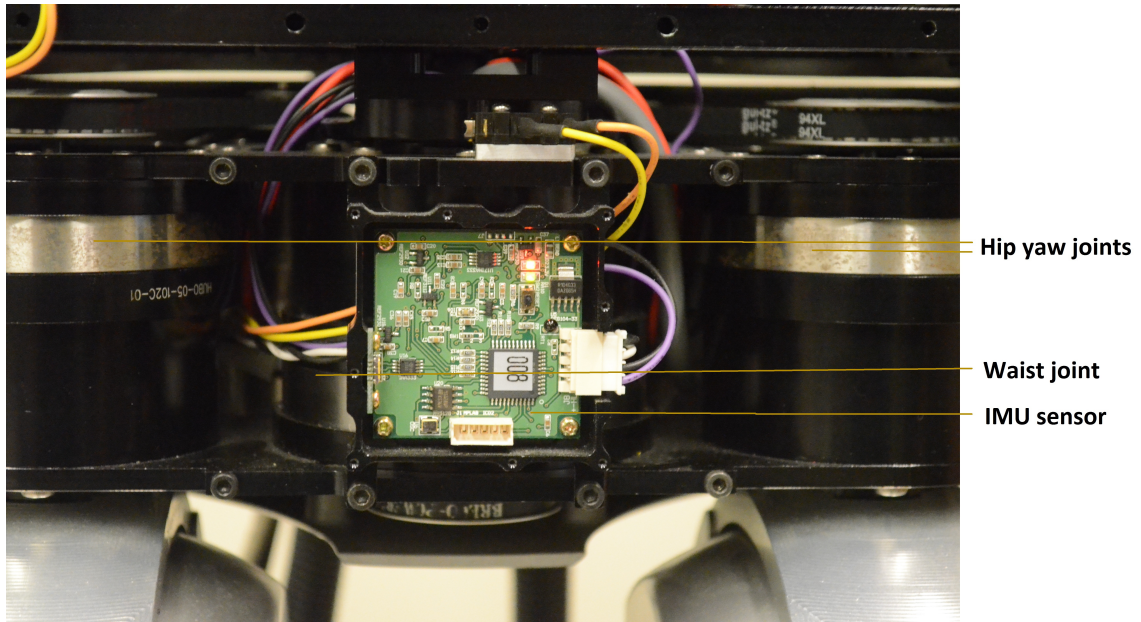


Fig. 2.10.: A picture of the IMU sensor mounted on the waist of the Hubo2+ robot.

has four three-axis force/torque sensors, two at the wrists and two at the feet. The force/torque sensors inform about the normal force F_z and the moments M_x and M_y . These sensors measure the forces applied onto the end-effectors and not the reaction forces. Photograph of the force/torque sensors are shown in Fig. 2.11.

- Infrared distance sensors:

When manipulating objects, camera view may get blocked by parts of the robot or the object itself. Thus, computer vision alone is not often sufficient to ensure that the robot is in a correct configuration to grasp the desired object. To know that, sensors are required at the end effectors. Knowing whether the robot has grasped or not can be inferred using force/torque sensors. But, when the end-effector is in the vicinity of the object, and has not grasped it yet, force/torque sensors cannot be used to predict if grasp will be successful or not. Thus, to be sure that grasping will be successful, it is necessary to know the distance of object from the end effector. Therefore four infrared-distance sensors are

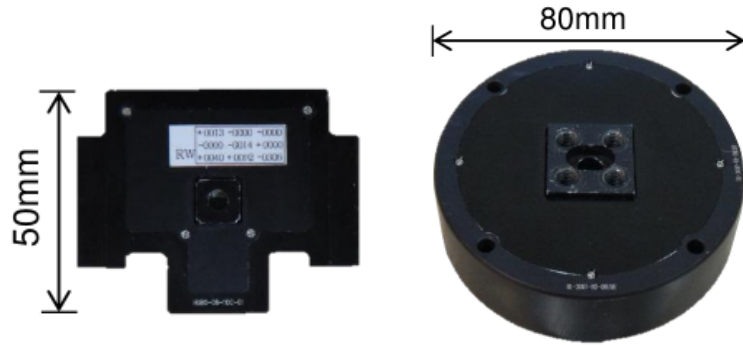


Fig. 2.11.: Pictures of the force/torque sensors used on the wrist (left) and ankles (right) of Hubo2+ robot.

installed on each palm of the DRC-Hubo robot. The sensors are not available in Hubo2+ robot.

- CAN card for communication:

Hubo2+ robot has 38 motors, 4 force/torque sensors, and 3 IMU sensors. There are a total of twenty-three JMC and sensor boards. Ensuring quick and reliable communication between the on-board computers and all of them is a very difficult task. This problem is faced in many industrial products like cars, aircrafts etc. and is solved using Controller Area Network (CAN) protocol. The body-computer as well as the JMC and sensor boards are connected to a CAN card (Fig. 2.12). Thus, the body-computer can interact with all of them and command them as required. A photograph of the CAN card used in Hubo2+ robot at ARTLab is shown in Fig. 2.13.

CAN is a multi-master protocol wherein there are many transmitters. Devices after scanning the message headers decide if they are interested to listen to the message or not. Farsi et al. have described the advantages of using CAN and provided some examples of how it is used for quick and reliable communication with minimal wiring [70].

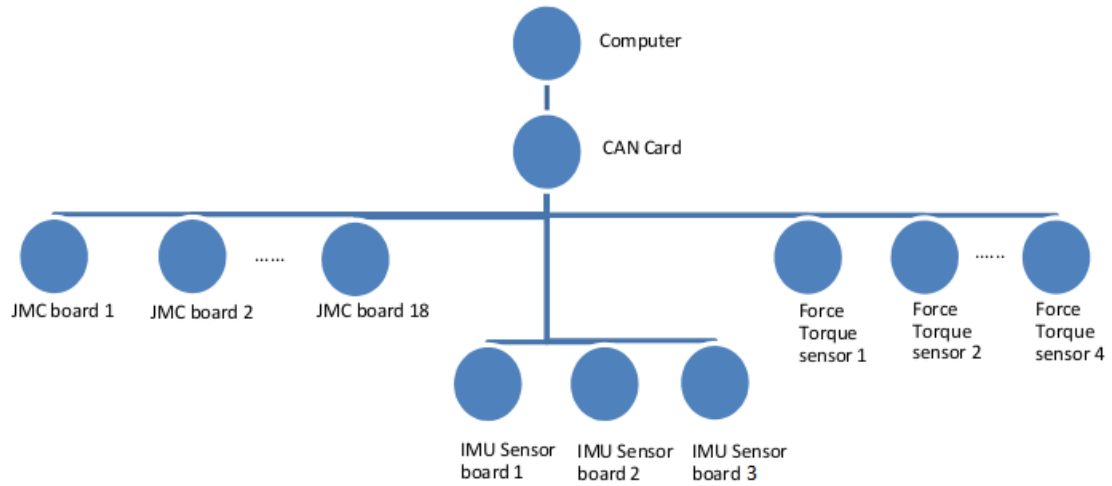


Fig. 2.12.: Communication in the Hubo2+ robot.

The main features of CAN communication that help in solving the problem caused by having a large number of electronic modules are:

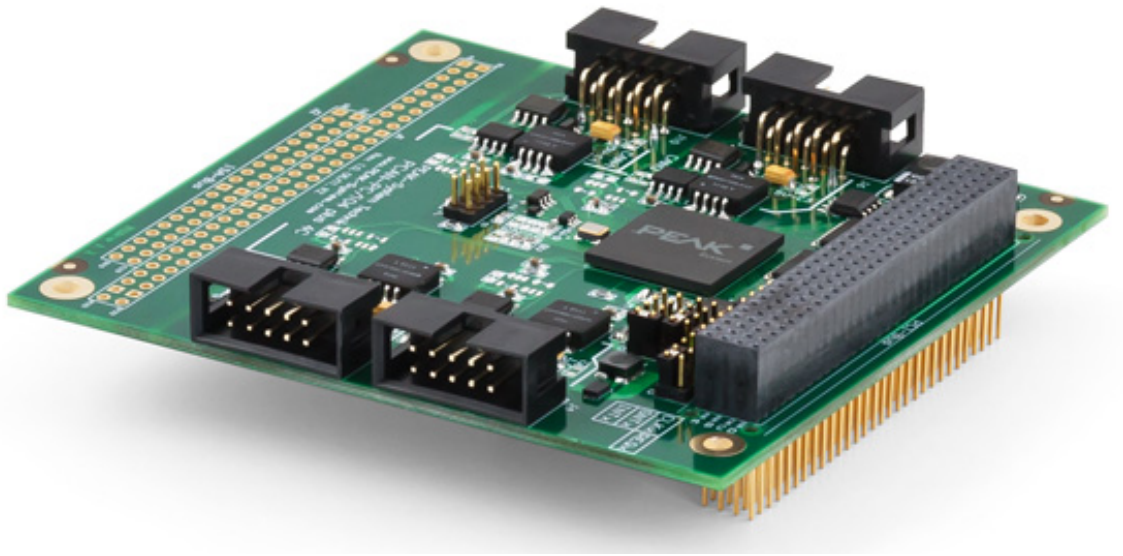


Fig. 2.13.: Peak system's four channel CAN card used for CAN communication in the Hubo2+ robot at ARTLab.

- Multi-master communication:

In CAN communication, there can be multiple transmitters who can send out messages. This essentially means that whenever an event occurs, message is sent out. The intended receiver obtains the message while others ignore it. With reference to the Hubo2+ robot, all the JMC boards, sensor boards and the on-board computer send out data. The data from the sensor boards and JMC boards are read by the on-board computer. The computer sends out messages to move the motors, initializes the sensors or requests the sensors to transmit the sensor readings. The intended boards read the messages and perform required operations.

- Priority-based bus arbitration:

When the bus is idle, multiple devices may start sending messages. To avoid this, only the device with highest priority (generally the lowest ID) starts transmitting.

By using CAN communication, the data-synchronization between controller boards and the on-board computer becomes faster and robust.

2.4 Integration of all electronic and electro-mechanical modules into one system

Previous section provided details of all the electronic and electro-mechanical modules. This section discusses how all these electronic and electro-mechanical modules are integrated into one functional unit for Hubo2+ robot. The communication between the body-computer and the large number of controller boards is done using the CAN card. This section discusses how the communication is implemented.

Hubo2+ robot has a large number of joint motors and sensors. Directly controlling all of them is inefficient. Thus, the solution is to use a two level-controller. All sensors and actuators are connected to a controller board. These boards provide the ground level of control. As explained in Section 2.3, using a high gain PID controller the JMC

boards ensure that the motors follow the command last obtained by the JMC boards. When requested, the JMC boards also send the encoder and joint status data (e.g. is the limit switch pressed? Is the joint in some error state? Is the motor driver working normally?) back to the body-computer. Similarly, the sensors upon obtaining data request, broadcast the sensor readings and calibrate the sensors whenever commanded to. At a high-level the body-computer on the Hubo2+ robot communicates with the joint motor controller (JMC) boards or the sensor boards through the CAN card. The software running on the computers uses the sensor information and generates a trajectory plan. At appropriate time, it sends commands to the JMC boards, and the JMC boards in turn control the motors.

In Hubo2+ robot, all the electronic modules are connected to the CAN bus. The body-computer and the controller boards can all write to the CAN bus. Thus the devices receive a large number of messages. Parsing and decoding all of them is neither necessary nor useful. Instead, the intended receiver is included in the message header. All the other devices connected to the CAN bus, read and parse the header. If the message is relevant to the device, the device then reads the complete message, otherwise it ignores the message. In the case of Hubo2+ robot, all the communication is directed towards or from the body-computer. Thus, the body-computer is either a transmitter or a receiver.

A variety of boards are connected to the CAN bus. Motor controller boards obtain position or PWM commands from the computer, and when requested, send the encoder and joint status information to the body-computer. The sensors obtain initialization commands from the body-computer. The IMU sensors send back acceleration information along the axes and rotational velocity about the axes. Similarly, the force/torque sensors send back the normal force acting on the robot and the moments about other two axes. Thus, the data communicated between the controller boards and the body-computer varies significantly. To simplify the task of communication, all the messages are encoded in a pre-defined format. The receiving device is thus easily able to decode the messages and perform required operations.

A software running of the body-computer has to encode the messages to move a joint in the pre-defined format and send it to the CAN bus. Whenever the software needs feedback from the encoders and sensors, it sends out request messages and waits for the encoder or sensor data. Based on the feedback, the new joint positions are calculated, encoded and sent to the joints. Such a feedback loop within the software is sufficient for controlling the Hubo2+ robot.

2.5 Summary

In this chapter Hubo2+ humanoid robot was introduced and discussed. The kinematic structure of Hubo2+ robot is modeled using the Denavit-Hartenberg (D-H) representation. Based on the D-H representation, the forward kinematics, inverse kinematics and dynamic computations can be performed. Various electronic modules used in the robot are explained. The need of those modules and their functionalities was explained. Finally the chapter also discussed how all these modules are connected into a functional Hubo2+ robot. The communication protocol was also explained. Description of how a software can communicate with the sensors and motors and make the robot perform different tasks will assist in understanding the architecture and design of hubo-ach and hubo-motion-rt. Chapters 3, 4, and 5 discuss the software architecture and implementation, for which an understanding of the robot hardware is crucial.

3. ARCHITECTURE OF HUBO-ACH: A LOW LEVEL HUMANOID ROBOT CONTROLLER

3.1 Introduction

The previous chapter discussed the hardware architecture of the Hubo2+ robot. The design of Hubo2+ robot's hardware was described. The need for various sensors and efficient communication protocol was discussed. Overview of how a software for controlling the Hubo2+ robot should be designed was also provided. This chapter focuses on software architecture of the robot. The design of lower-level control software: hubo-ach is discussed in detail. Various components of hubo-ach and the design decisions taken for its development are also discussed. Though Hubo2+ robot is used as an example, hubo-ach can be used for controlling other humanoid robots as well. This chapter concludes by providing a skeleton of how controllers can be developed using hubo-ach.

3.2 Purpose of hubo-ach

Chapter 1 established the need for a new software for controlling humanoid robots. The philosophies based on which the software is designed were pointed out. The main purpose of designing and developing hubo-ach is to provide a generic real-time, modular, multi-lingual, and open-source software that can be used for controlling humanoid robots, especially Hubo2+ robot. A secondary advantage is that the software would abstract out the drivers and application programming interfaces (APIs) to communicate with the actuators and sensors. Since hubo-ach is a generic software, if one desires to use it for a different robot, the drivers need to be integrated as a part of hubo-ach. However, once they are available, other developers can just re-use them. In

essence hubo-ach is required so that the requirement of a real-time, modular software package that can be used for controlling humanoid robot can be fulfilled.

3.3 Architecture of ROS

Chapter 1 provided the motivation to develop a new software for controlling humanoid robots. Features and drawbacks of using ROS- the most commonly used robot control software - were discussed. This section discusses the architecture of ROS in detail. Section 3.3.1 discusses the architecture of the ROS. Section 3.3.2 discusses why ROS is not suitable for real-time control. Section 3.3.3 discusses various applications of ROS.

3.3.1 Overview of ROS

In order to be modular and to ensure that one package does not affect functioning of another, ROS follows a multi-process architecture. A process that performs computations is called ‘node’. It is a software module performing a specific action. Different nodes communicate with each other by passing ‘messages.’ A message is data structure that is composed of primitive data types like integers, strings etc. and other messages. To establish communication, a node publishes a message to a given ‘topic.’ Topic is like a buffer where the communicated messages are stored. Multiple publishing nodes can write to a topic and multiple subscriber nodes can subscribe to a topic. The communication done using messages is broadcast type communication. The topic is implemented with a first-in-first-out(FIFO) protocol. Figure 3.1 depicts this process.

Often, processes need a question-answer type communication, which means a process *asks* other process a question and gets a response. This is similar to a web service, which is designed to support machine-to-machine interactions over the world wide web. ‘ROS service’ fulfills this requirement. Apart from the nature of communication, another major difference between ROS services and ROS messages is that

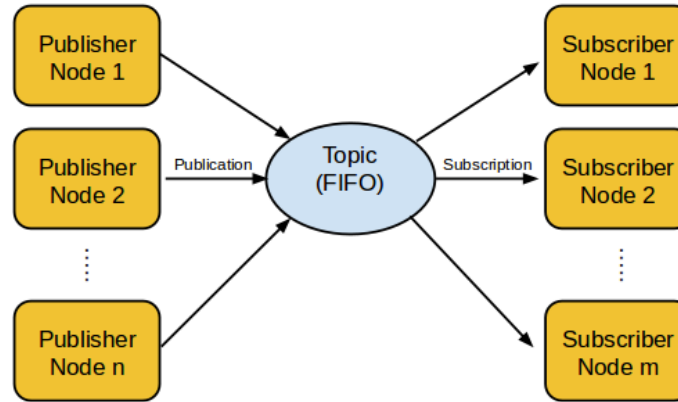


Fig. 3.1.: Illustration of ROS publishers and subscribers.

only a single node can advertise a service. If this constraint is not created, then the node using the service will get multiple responses and thus, it is difficult to know which is a correct response. Figure 3.2 depicts how a service type communication is done.

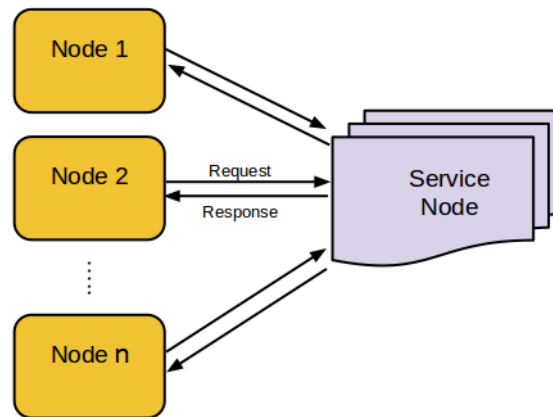


Fig. 3.2.: Illustration of ROS service.

A lot of robotic software packages have been designed using ROS. Industry support for ROS is available for many robots like NAO humanoid robot, UBR-1 robot, and PR2 robot. ROS user-community is expanding and therefore support for a wide

range of robots is available. ROS can be used for controlling any custom-made robot as well. Figure 3.3 depicts how a robot can be controlled using ROS. A general framework usually has multiple nodes, which communicate with each other using messages and services. Due to the multi-process architecture, a package can be easily updated without affecting other packages. Such an architecture also means, if one nodes crashes due to a bug in its implementation, other nodes are not affected.

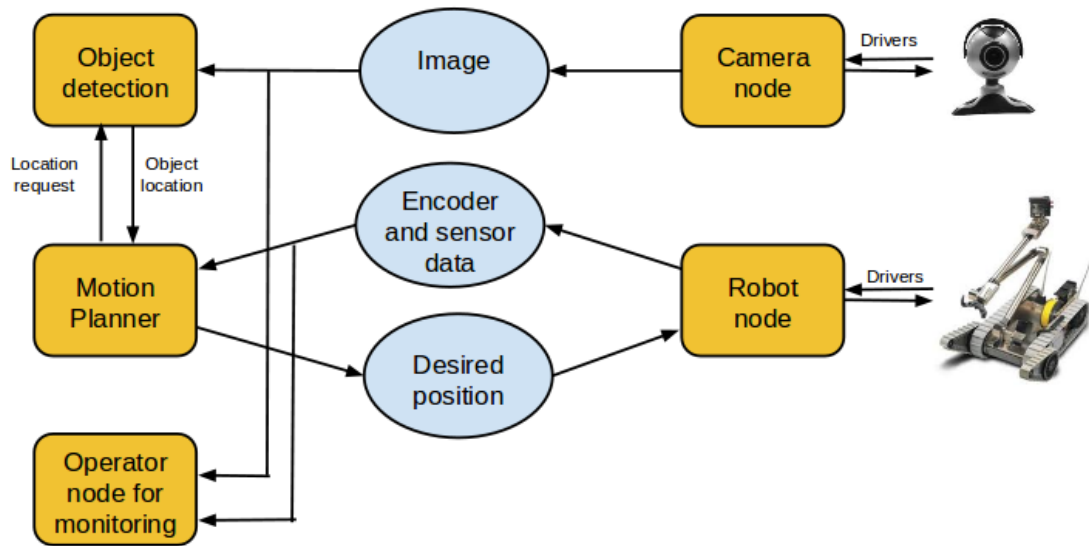


Fig. 3.3.: Illustration of how ROS can be used to run a robot.

3.3.2 Analysis of non-real-time behavior of ROS

ROS is designed with distributed computing in mind and so can be used across multiple machines. Nodes are unaware of where they are executing. In order to ensure that the communication between two processes running on the same or different machines is reliable, Transmission Control Protocol (TCP) is used. This is the same protocol used for internet communication. TCP is optimized for accurate delivery and ensuring that every message is delivered. However, it is not optimized for timed

delivery and so using TCP can cause long delays. Therefore for real-time applications like voice over Internet Protocol (IP), TCP is not used.

To test the delay in inter-process communication, a small experiment was performed. Five publisher nodes to publish data to a same topic at a fixed frequency were created. A listener node to listen to the data was created. The latency between publishing and receiving was noted down. At 200 Hz frequency, the average latency was 1.002 ms, the median was 1.004 ms and the standard deviation was 0.287 ms. The experiment was repeated for 1 kHz frequency. The average, median and standard deviation were observed to be 3.185 ms, 2.004 ms and 3.084 ms, respectively. These experiments were performed on a Intel i7 processor with 16GB of RAM and Ubuntu 12.04 running on it. With more number of nodes, the latency increases.

In the current state-of-the-art robots like Hubo2+ robot, the on-board computers communicate with controller boards at a frequency of 200 Hz (i.e. 5 ms) or more. With improvements in technology, this number is expected to increase. From the above experiments it can be concluded that even if five ROS messages publish and receive data, two-way communication requires approximately 2.6 ms ($= 2 * (\text{mean} + \text{standard-deviation})$) of delay, if publishing rate is 200 Hz, and approximately 12.4 ms, if the publishing rate is 1 kHz. The on-board computers on the Hubo2+ robot are not as powerful as the computer used in the experiment. Therefore, the latency is expected to go higher. Since, for real-time control, only a few milli-seconds are available, ROS cannot be used for controlling Hubo2+ humanoid robot for real-time applications.

3.3.3 Application of ROS

In spite of the non-real time behavior of ROS, it is widely used for controlling robots. This seems very intriguing. It should be noted, that a lot of robotic applications, do not require real-time response. Using autonomous robots for assembly tasks, executing motion planning algorithms, navigation in unknown environments,

or executing motions which are statically stable are some common applications where real-time response of a robot is not a necessity. ROS is multi-lingual, open-source, and supports thin development philosophy. It can be used on a cluster of computers. The ROS nodes are agnostic of where they are being executed. The inter-process communication is robust. Since the ROS-users-community is expanding, so a lot of support is also available. A large number of robot manufacturers provide packages to control their robots using ROS. Due to these advantages, it is very easy to develop software packages using ROS. Hence, applications where real-time response is not a necessity, ROS can be used.

3.4 Architecture of hubo-ach

ROS is not suitable for controlling humanoid robots for real-time applications. So a novel software package has been developed which has an architecture similar to that of ROS. It uses multi-process architecture and exploits all the advantages like modularity and segregation of different software packages. However, instead of using TCP for inter-process communication, an open source library called ‘Ach’ is used. Using a mutual exclusion (mutex) and condition variable, Ach synchronizes access to shared memory (channels). Thus, the channel-reader-processes can either periodically poll the shared memory for new data or choose to wait until a writer process has posted a new message. Using a read/write lock instead allows only polling. Additionally, synchronization using a mutex prevents starvation (a situation in which a process cannot access shared memory for a long time) which is necessary for maintaining real-time performance [71].

3.4.1 Overview of hubo-ach

Hubo-ach is developed to be real-time and modular software for humanoid robot control, which also abstracts out communication with controller boards. A humanoid robot has to execute multiple algorithms in parallel. Executing all of them in a single

process sequentially, violates the principle of modularity. In general, modularity of control software is achieved by running multiple processes in parallel (multi-process architecture). Therefore hubo-ach is also developed as a multi-process software.

The main process of hubo-ach is called “hubo-daemon”. It is responsible for the communication with joint motor controller (JMC) boards and sensor boards. This process via an inter-process communication (IPC) library shares the most recently received data with other processes. It also collects the most recent commands to the JMC and sensor boards, encodes them and sends them through CAN to the controller boards. Figure 3.4 depicts the architecture of hubo-ach.

Ach library is used for inter-process communication. For inter-process communication, ach channels are created. The data is shared across multiple processes through these ach channels. Detailed description of the Ach library is provided in Section 3.4.2.

To share the data effectively, hubo-ach creates and initializes many ach channels. Some of them are:

- hubo-ref: It is an ach channel that shares the commands to be sent to the motors. Other processes, which implement controllers for different joints, write to this ach channel. Hubo-ach reads from this channel, encodes the data and sends it to the JMC boards.
- hubo-ref-neck: It is an ach channel that shares the commands to be sent to the neck motors. The reason to have a different channel for neck joints is to give the user ability to independently control the neck. The neck configuration, due to its low mass and less range of motion, has very small effect on the position of the center of mass of the robot. Therefore, creating a different ach channel for neck gives the user the flexibility to move the neck joints without disturbing the other joints and thus monitor the environment without affecting the stability of the robot. If a different channel for neck motor control is not created, then

the computer vision algorithms and the trajectory planning and motion control algorithms will have to run within a single process, which is not desirable.

- `hubo-board-cmd`: It is an ach channel that shares the latest commands to initialize the motors and sensors. In case a joint goes into error states, it needs to be re-initialized and “homed”. Thus, the controller would need to write to the `hubo-board-cmd` so that the joint is correctly reset.
- `hubo-state`: It is an ach channel that shares the latest sensor and encoder feedback. Other processes can read from this ach channel and plan the trajectory. This data is also required for motion controllers. The status of various joints and the error and warning flags (Is the limit switch pressed or not? What mode the joint is operating in? Is the motor jammed?) are also a part of the data shared through this ach channel. Therefore, a robot operator or control software should monitor this data to ensure that the robot is safe and behaving as expected.

Hubo-daemon communicates with the JMC and sensor boards. This process, reads the `hubo-ref`, `hubo-ref-neck`, and `hubo-board-cmd` ach channels that store the latest commands to be sent to the JMC and sensor boards, encodes the data in them and sends it to the controller boards via CAN. Similarly, after acquiring latest data from sensors and encoders, `hubo-daemon` shares that with other processes through `hubo-state` ach channel.

Other processes can read the latest sensor and encoder data from `hubo-state`, perform the necessary computations, and write desired joint positions or other commands to the JMC boards in the `hubo-ref`, `hubo-ref-neck`, or `hubo-board-cmd` ach channels. Thus, using multi-process architecture, the communication with the controller boards is abstracted out. Multi-process architecture also leads to modularity as the communication with the controller boards, control algorithms, planning algorithms, and computer vision algorithms all run as different parallel processes. Since

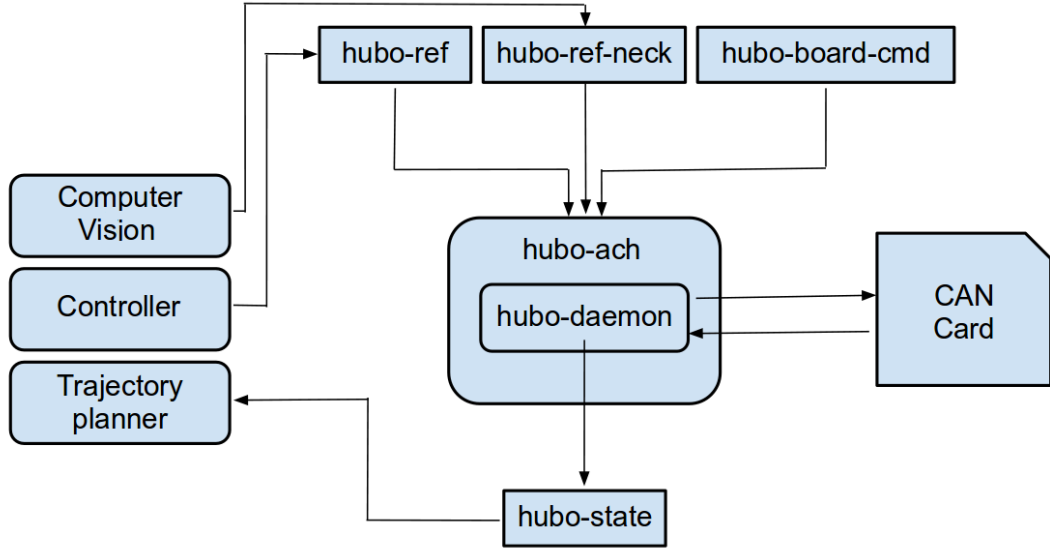


Fig. 3.4.: Architecture of hubo-ach.

these processes do not affect each other, they can be easily modified or upgraded without affecting other controllers or the hubo-daemon.

To obtain real-time functioning, the only requirement is that the inter-process communication has to be significantly faster than the frequency of hubo-daemon. Hubo-daemon runs at 200 Hz by default, as, that is the maximum communication frequency supported by the hardware of the Hubo2+ robot. Most other robots also have similar timings for communication with the controller boards. Currently, for any humanoid robot, the maximum frequency for communication with the controller boards is not more than 1 kHz. Using Ach, results in worst case latency of $20 \mu\text{s}$ per receiver at 1 kHz rate. Thus, Ach meets the latency requirements and can be used for inter-process communication.

With the described architecture, all the requirements for a humanoid robot control software package are satisfied. The package is real-time, modular, and abstracts out communication with the controller boards. Sections 3.4.3, 3.4.4, and 3.4.5 discuss the implementation of this architecture in detail.

3.4.2 Ach: Inter-process communication (IPC) library

In order to develop a real-time and open-source software, all the libraries used within this software must also follow similar, if not exact design philosophies. The ‘Ach’ library developed at Georgia Institute of Technology is efficient and open-source [71]. Due to the low latency in data sharing this library is specially suited for coordinating drivers, controllers and algorithms in robotic systems [71]. As described in Section 3.3.2, ROS is not suitable for real-time inter-process communication. As NAOqi and OROCOS do not provide the ability for multiple processes to publish and subscribe to shared memory, therefore they are also not suitable for real-time control [55, 72].

The latency in communication using ROS is in the order of a few milli-seconds. Thus, it is not suitable for real-time control in robots. Ach has latency delays of the order of a few microseconds. Even if a large number of processes share memory and the frequency of communication is increased significantly, the latency in communication is lesser than $100 \mu s$. The histogram of latency delays observed when Ach is used for inter-process communication is shown in Fig. 3.5. Ach also provides a ROS-messaging-like interface. Processes can read and write from a shared memory (channel). However, there is no interface like a ROS service. Hence, Ach has a better protocol for inter-process communication for robotic systems. Figure 3.6 depicts the use of Ach library for inter-process communication.

The basic usage of this IPC library is to share data across numerous processes. If any two or more processes P_1, P_2, \dots, P_n need to communicate, then this library can be used. To begin data sharing the processes need to share the following:

- The ach channel name through which the communication would be performed;
- A data structure that is specific to the processes.

Once this is shared and verified, any one of the n processes then initializes the channel and all of the processes can read data from the channel and write data to it. The channel stores the latest data written to it and data sharing across various

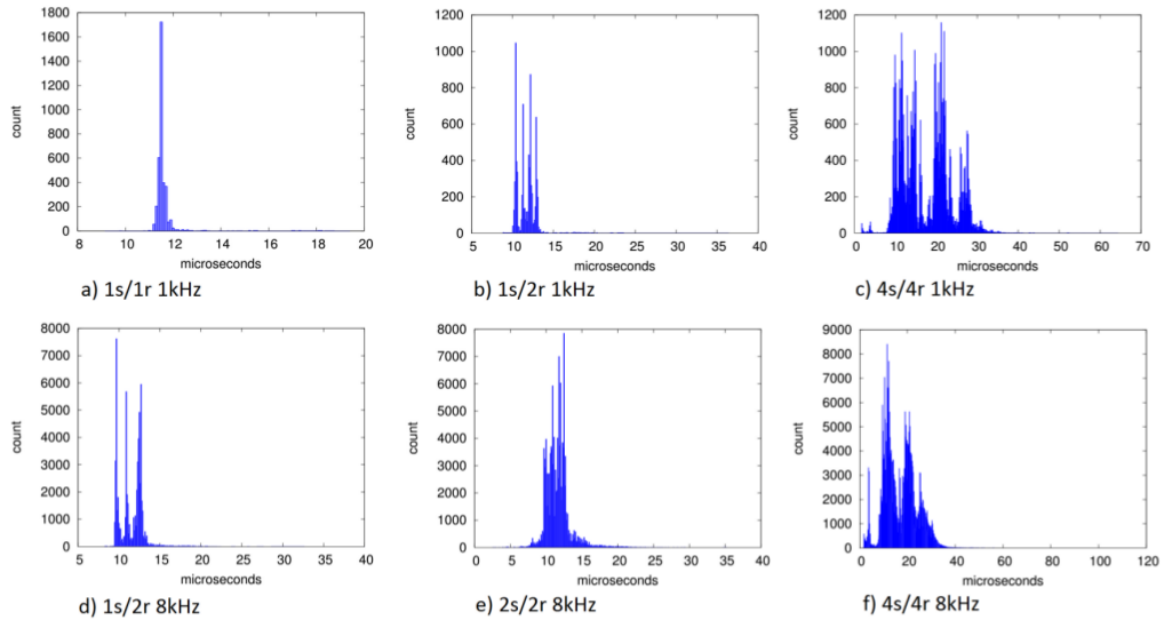


Fig. 3.5.: Histograms of Ach and Pipe messaging latencies. Benchmarking performed on a Core 2 Duo running Ubuntu Linux 10.04 with PREEMPT kernel. The labels s/r indicate a test/run with sending processes and receiving processes [71].

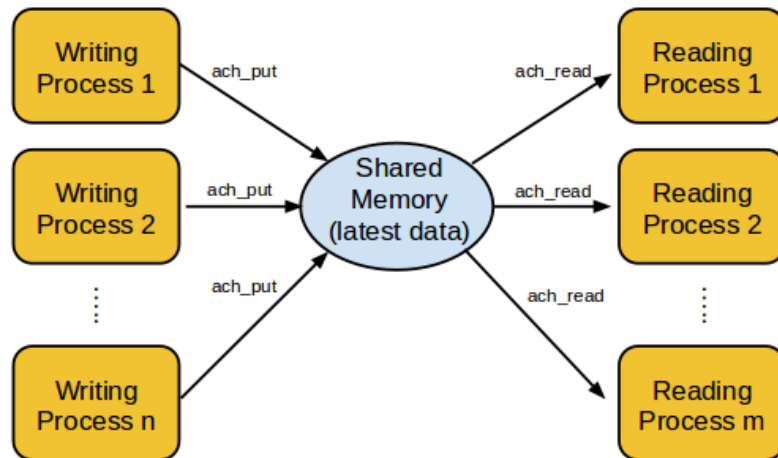


Fig. 3.6.: Communication between processes using Ach.

processes is done with ease. Since this library is optimized for real-time systems, it best suits our needs for real-time control of Hubo2+ robot.

When using Ach library, the latency in inter-process communication is of the order of a few micro-seconds. Using ROS, leads to inter-process communication latency of the order of a few milli-seconds. Clearly, for real-time applications, Ach should be used for inter-process communication. However, when real-time performance is not a necessity, ROS should be used as TCP leads to zero error in communication. Also, the fast-growing user community of ROS and availability of wide range of packages compatible with ROS significantly reduces the development time.

3.4.3 Data structures

A secondary motivation of hubo-ach is that the user does not have to worry about the communication with the joint motor controller and sensor boards. In case of Hubo2+ robot, the communication is done through the CAN card. Hubo-ach is a multi-process software package. It is expected that different processes will communicate with hubo-ach for controlling the robot. The multiple processes all required the latest sensor data. These processes would also send commands to the JMC boards through hubo-ach. One possible architecture is to send the latest commands immediately. That is, whenever hubo-ach receives new commands to move the joint motors from other processes, it sends the commands to the JMC boards. Similarly when hubo-ach receives request for the sensor data, hubo-ach can first obtain the latest sensor data from sensor boards and then send it to other processes. Though, this method will ensure that commands are immediately sent and latest sensor data is received, it poses the threat that the communication bus can get saturated to its full bandwidth. Also, if the data is requested more frequently than the refresh rate of the controller boards, the received data will be repeated. Therefore to avoid the communication bandwidth saturation and the duplication of received data, hubo-ach sends commands to the motor controller boards and requests and obtains the sensor

data at a rate of 200Hz. Currently 200Hz is the maximum frequency that Hubo2+ robot's hardware can support. If the software is used on a different robot, this rate should be altered to the maximum possible update rate.

To avoid this bandwidth saturation, the latest command that needs to be sent to the controller boards should be stored in a temporary buffer. During a cycle of hubo-ach, this data is read, encoded and sent to the controller boards. Similarly, the latest sensor and encoder data should be available for other controller programs. Thus a temporary buffer is also used to store the decoded data received from the sensors. The sensors and the encoders, together with the joint status, are complete information about the state of the robot. Therefore the data structure used to store this data is called `hubo_state`. This data structure is shared through hubo-state channel with other processes. The data received from encoders and sensors are very different. Thus, to elegantly organize all of those, sub-data structures to store the information should be created. Arrays of these data structures together form the `hubo_state` data structure.

The position commands sent to the motor controller boards are the reference values for the joints. Thus, the data structure used to store the latest position commands is called `hubo_ref`. Hubo-daemon obtains this data from other processes through `hubo-ref` ach channel. Finally, the buffer used to store commands to initialize and “home” the motors or sensors or choosing the mode of operation is called `hubo_board_cmd` as it sends the most basic commands for operating the boards.

- `hubo_joint_state`: The data structure used to store all the data about a joint is called `hubo_joint_state`. To know the behavior of a joint, one needs to know what is the desired position of the joint, that is, the last commanded position of the joint. This value is called ‘ref.’ As mentioned in Chapter 2, the joint can be in the position-control mode, or in the compliance mode. The variable ‘comply’ stores this information. The encoder value and the velocity of the joint are a must-know and are stored in the variables ‘pos’ and ‘vel’ respectively. Users may also need to know if the joint is active or not and ‘active’ variable stores

that information. Finally, if the joint is not behaving as expected, for correction, users would need to know if the joint had been calibrated or not and the variable ‘zeroed’ stores that information (Fig. 3.7).

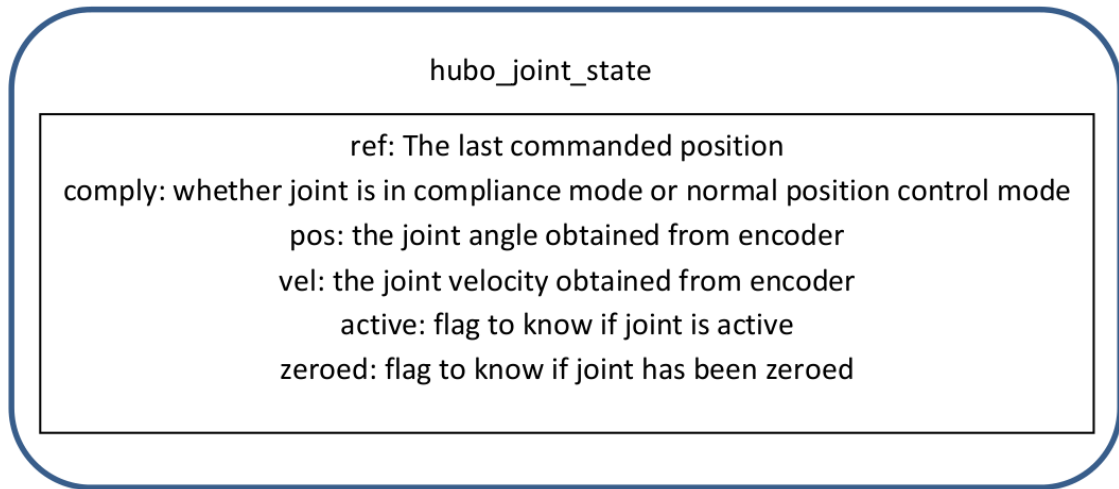


Fig. 3.7.: hubo_joint_state: The data structure that stores the joint and encoder data.

- hubo_ft: Similarly, the data structure that is used to store all the data about a force/torque sensor is called hubo_ft. The force/torque sensor informs about the moments produced by external forces along x and y axis and the normal force acting along z axis of the sensor. Thus the data structure stores three values namely: m_x and m_y which are the moments produced by external forces about the x and y respectively and f_z is the force along z axis of the force/torque sensor. In case, the software is used for a different robot that has 6-axis force/torque sensors, or the force/torque sensors on the Hubo2+ robot are upgraded, this data structure should be modified to include the new data. That is, the data structure should store three more values namely m_z , f_x , and f_y to store the moment about z axis and forces along x and y axes respectively (Fig. 3.8).

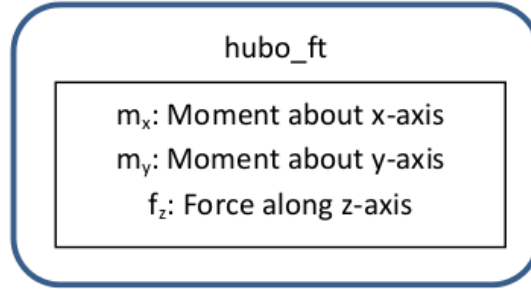


Fig. 3.8.: hubo_ft: The data structure that stores force/torque sensor data.

- hubo_imu: The data structure that stores the data from IMU sensors is called hubo_imu. The data from an IMU sensor is the acceleration along and angular velocity about its three axes. Thus the data structure has six variables namely: a_x , a_y , and a_z which are the accelerations along x , y , and z axis respectively and ω_x , ω_y , and ω_z which are the angular velocities about x , y , and z axis respectively. In case a new IMU sensors is used, which using magnetic compass also informs about geographical directions, the hubo_imu data structure should be updated to include that data. Thus, in that situation, the data structure should have three more components namely mc_x , mc_y , and mc_z to store the magnetic fields along the x - y - z axisi (Fig. 3.9).

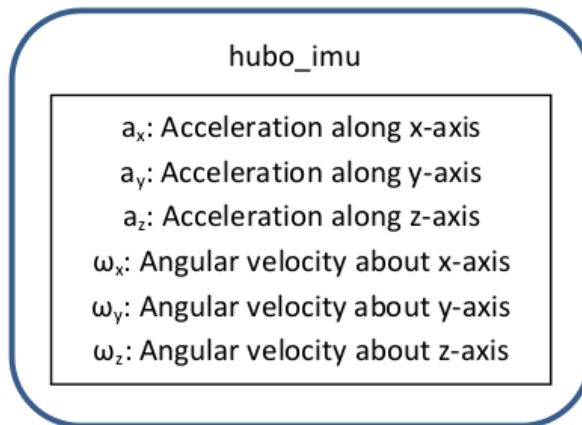


Fig. 3.9.: hubo_imu: The data structure that stores IMU sensor data.

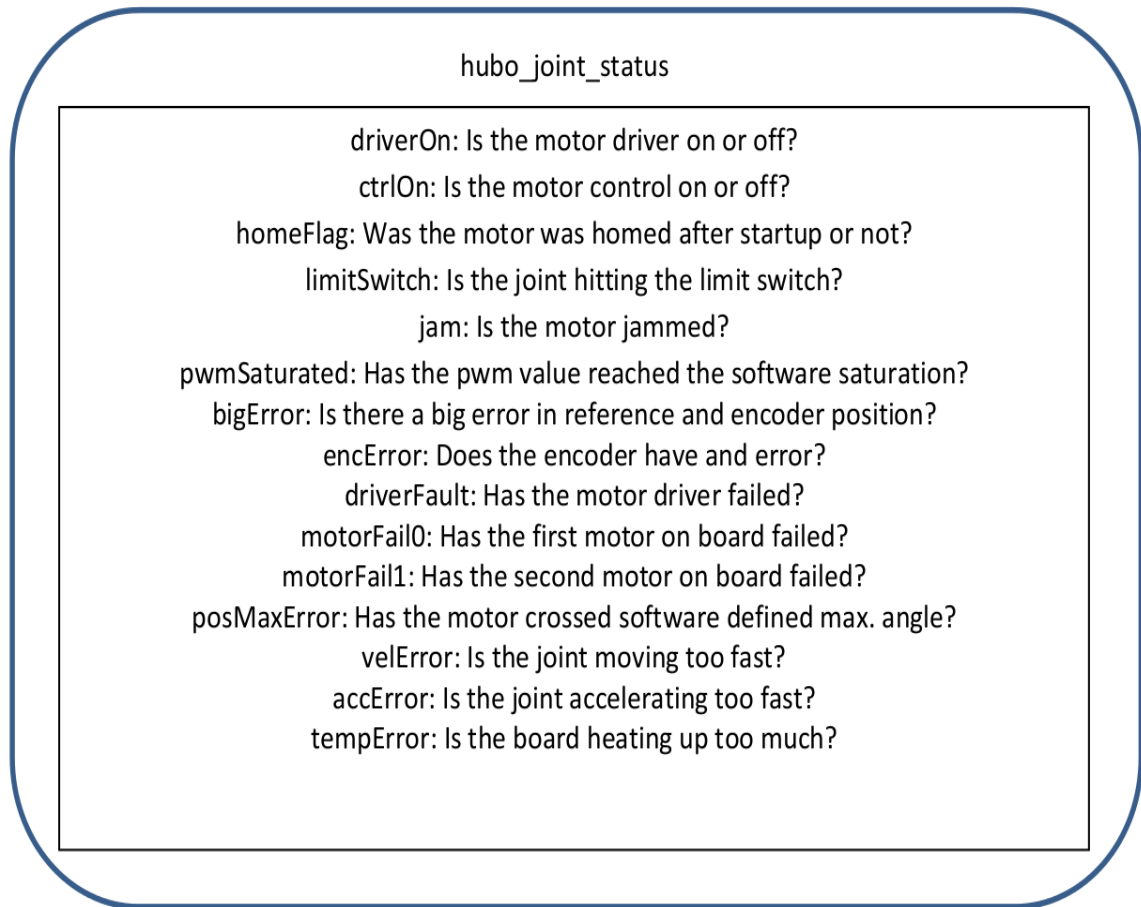


Fig. 3.10.: hubo_joint_status : The data structure that stores a joint's working state.

- hubo_joint_status: In some cases, specially when turning on the motors and when the joints are close to their rotation or torque limits, the motor controller may go into some error or warning states. Thus, a data structure that stores those errors and warning flags for each and every joint is required (Fig. 3.10). This data structure is called hubo_joint_status. For a given joint motor, it is essential to know if the motor driver is on or off, the motor-controller is on or off, etc. Also the control mode of the motor (rigid mode or compliance mode) must be known. 'homeFlag' is required to know if the motor was "homed" after start-up or not. When a motor is "homed" these flags are updated to their

default (working) state. During the motion, the motors can go into various errors or warning states. The possible states that it can go into are:

- The motor has hit limit switch.
- The motor has jammed.
- The motor has reached its PWM saturation limit.
- There is a huge difference in the reference and current motor position.
- The encoder has an error.
- The motor driver has failed.
- Any of the motors controlled by the board have failed.
- The joint has reached the minimum or maximum angle possible.
- The joint is moving or accelerating too fast.

Since this is the information about the working status of a joint, it is called `hubo_joint_status`. The robot operator should constantly monitor this status to check for any errors. In case of an error, the operator should take appropriate decision to rectify the error. Other humanoid robots will have some different errors and warning states. Variables to store such information should be included in this data structure.

- `hubo_state`: To store the complete information in the `hubo_state` data structure, an array of `hubo_joint_state` data structure of size equal to the number of joints in the robot, an array of `hubo_imu` data structures of size equal to the number of IMU sensors, an array of `hubo_ft` data structures of size equal to the number of force/torque sensors in the robot, and an array of `hubo_joint_status` of size equal to the number of joints in the robot are required. This data structure now stores the complete information of the robot (Fig. 3.11).

The `hubo_state` data structure is designed to store all the data that can be received from Hubo2+ robot. Though other humanoid robots would have different

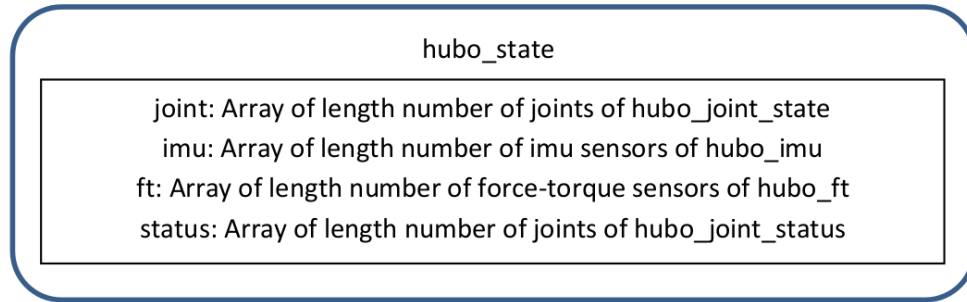


Fig. 3.11.: `hubo_state`: The data structure used to store the current state of a humanoid robot.

set of sensors, most of them do have IMU sensors and force/torque sensors. In case there are other type of sensors like proximity sensors or magnetic compass, data structures to store the sensor information should be created and they should be included as a part of the `hubo_state` data structure. Thus, for any humanoid robot `hubo_state` data structure stores all the robot information that can be received from it.

- `hubo_ref`: `Hubo_ref` data structure stores the latest commands to be sent to the JMC boards. For every joint, the command that is sent should have the information about what is the desired position of the joint, if `hubo-ach` should internally filter the motion so that the joint motion is smooth, and if the joint is in normal position control mode or in compliance mode. Thus the `hubo_ref` data structure consists of three arrays: one storing the desired position, the second for storing if internal filter should be used or not, and the third to store if the compliance mode should be turned on or not.

The `ref` array contains the latest position data that is to be sent to the motors (Fig. 3.12). Often the planner trajectory may not be smooth and sudden jumps in the motion are harmful to the robot. Hence, `hubo-ach` provides an internal filter to smoothen the motion. This is termed as the ‘mode’ of the

joint. It can be either 0, which corresponds to the filter mode, or 1, which corresponds to the direct reference mode. In the filter mode, the motor is not commanded to directly go to the desired location, but it is sent various intermediate points so that the motion is smooth. In the direct reference mode, the motor is commanded to directly go to the desired joint position. This mode should be used only when the trajectory is fine and has no big jumps for any joint. The motion of a joint in filter-mode and direct reference mode is shown in Fig. 3.13. Finally, the ‘comply’ array is to decide whether the motor should be in the compliance mode or normal position-control mode (discussed in Chapter 2).

Since `hubo_ref` stores the commands for actuator motion, the data structure can be used for all the humanoid robots. Depending upon the hardware of the humanoid robot, there may be more or fewer features that are supported. For example, instead of compliance, the hardware may support torque control. In that case, the ‘comply’ array should be replaced by ‘torque-control’ array. The filtering feature is a part of the software and not the hardware. So this feature is available for all the robots.

- `hubo_board_cmd`: Earlier the need to store the basic initialization and homing commands in a temporary buffer called `hubo_board_cmd` data structure was discussed. For every joint of a humanoid robot, this data structure stores the information about whether the board or the controller needs to be initialized, or the motor needs to be “homed”. The data structure is also used if the control mode of the joint needs to be switched. The data structure is divided into five parts. The ‘type’ variable is the command type, which informs what is required to be done. The ‘joint’ variable is the target joint. This may be unused if the command is generic to all the boards (for instance, “initializeAll” command to initialize all the sensors). ‘Param’ stores the parameters of the command, ‘iValues’ stores the integer values (e.g., changing the joint limits or control gains)

and ‘dValues’ stores the double-precision point values that may be required for the command. This data structure is generic to humanoid robots as all the controller boards in a robot need initialization and calibration. If some extra features are available, users can easily edit and extend this data structure to add them.

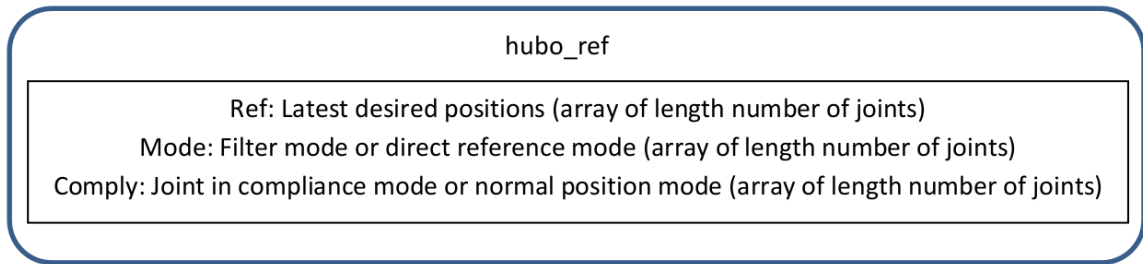


Fig. 3.12.: hubo_ref: the data structure that stores the latest commands sent to a joint.



Fig. 3.13.: Joint motion in filter-mode (left) and direct reference mode (right).

3.4.4 Abstracting communication

The data structures defined until now stored data to simplify the user control. These data structures are quite generic and can be easily modified and updated to

include a wide range of humanoid robots. However, being a low-level humanoid robot controller, hubo-ach needs to communicate through CAN card as well. Since different robots have different communication protocols and message encoding, the following data structures are specific to the Hubo2+ robot. However, by understanding the use and implementation of these data structures, users can write similar data structures for their specific humanoid robot.

- `hubo_joint_params`: Hubo-ach, in the case of Hubo2+ robot, has to interact with the CAN channel to encode the commands and decode the responses from each joint and sensor. The encoding and decoding information for all the joints is different but has same the characteristics. So two data structures, one for the joint parameters and the other for sensor parameters are created. Without these parameters, message decoding will be difficult and may yield incorrect information. For example, the angular displacement of a joint can be obtained from the number of counts of the encoder. The conversion from the number of counts of the encoder to the displacement in radians is obtained from the following formula:

$$radianAngle = 2.0 * \pi * encVal * \frac{(drive/driven)}{(harmonic * enc)} \quad (3.1)$$

where *radianAngle* is the current position of the joint in radians, *encVal* is the raw encoder data of the joint, *drive*, *driven*, *harmonic*, and *enc* are joint parameters that correspond, respectively, to the size of drive and driven wheel, the gear ratio of the harmonic drive, and the size of the encoder and π is the standard mathematical constant, approximately equal to 3.14159. These are all obtained from the datasheet provided with the robot and stored in the joint parameters data structure. Apart from this some additional information like the joint number, the CAN channel that the board is connected to, and the number of motors on the board are also stored in the data structure. These values may be changed if the hardware of the robot is changed. To simplify this change, these values are stored in a table called ‘joint.table.’ The user can

easily change the table. On start-up, hubo-ach reads up the values from the table. Thus, after hardware changes, instead of scanning through the code and changing the values, the user just changes the table. All this data is stored in a data structure called `hubo_joint_params`.

- `hubo_sensor_param`: Similar to `hubo_joint_param`, `hubo_sensor_param` stores the sensor number, the CAN channel it is connected to, the board number, a flag to mark if the sensor is active, and the name of the sensor. As in the case of joint parameters, instead of hard-coding the sensor parameters, they are stored in a table and upon start-up loaded into the corresponding data structure. The table to store the sensor parameters is called ‘`sensor.table`’.

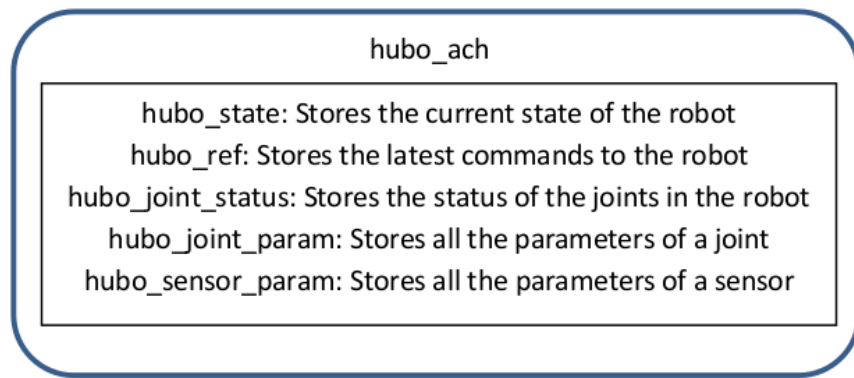


Fig. 3.14.: Components of `hubo_ach`.

Upon startup, `hubo-ach` first creates and initializes various ach channels that are used. Next it loads and parses joint and sensor tables to load the parameters. It updates `hubo_ref` and `hubo_state` ach channels and initializes all the motor controller boards. Then the main control loop of `hubo-ach` called “`huboLoop`” starts executing. Within the loop, `hubo-ach` first updates all the data structures to their latest values. Next, depending upon the data in `hubo_board_cmd` and `hubo_ref`, the software encodes the messages to JMC and sensor boards and sends them through CAN card. In this stage, requests for sensor and encoder data are also sent to the boards. Finally, it

receives the sensor, encoder and joint status data from the boards. It goes back into the loop and updates the data structures.

To use hubo-ach for controlling a different humanoid robot, only the message encoding-decoding and communication modules need to be written. The remaining framework does not need any significant changes. As for the users, for controlling the robot, they just need to read from the hubo-state ach channel to receive feedback and write to hubo-ref and hubo-board-cmd ach channels to move the robot. Thus, using this design, communication between the computer and the controller boards is abstracted out.

3.4.5 Organizational flow of hubo-ach

The data structures discussed in Section 3.4.3 and Section 3.4.4 were designed with the motivation to provide users the latest data sent to the humanoid robot as well as the feedback received from it. Since the data-sharing is done using Ach library, so any process or software that the user runs should be able to access all that data. Abstracting out communication and sharing the data throughout the computer is the most significant characteristic of hubo-ach. Abstraction leads to quicker controller development cycles and data sharing leads to modularity in development. Instead of having one single code implementing all the features, software can be developed so that different processes are responsible for achieving different objectives. This also allows the users to write his/her own control programs independently without affecting the hubo-ach software.

Hubo-ach, itself has to ensure smooth data sharing and communication. On start-up, it has to go through a few steps before beginning the control loop. First, as it has to share data across the processes, it creates and initializes various ach channels including those mentioned in Section 3.4.1. After initializing the ach channels, hubo-ach reads the table ‘joint.table’ and loads the joint parameters for each joint into its respective data structure. It then similarly loads the sensor parameters from the

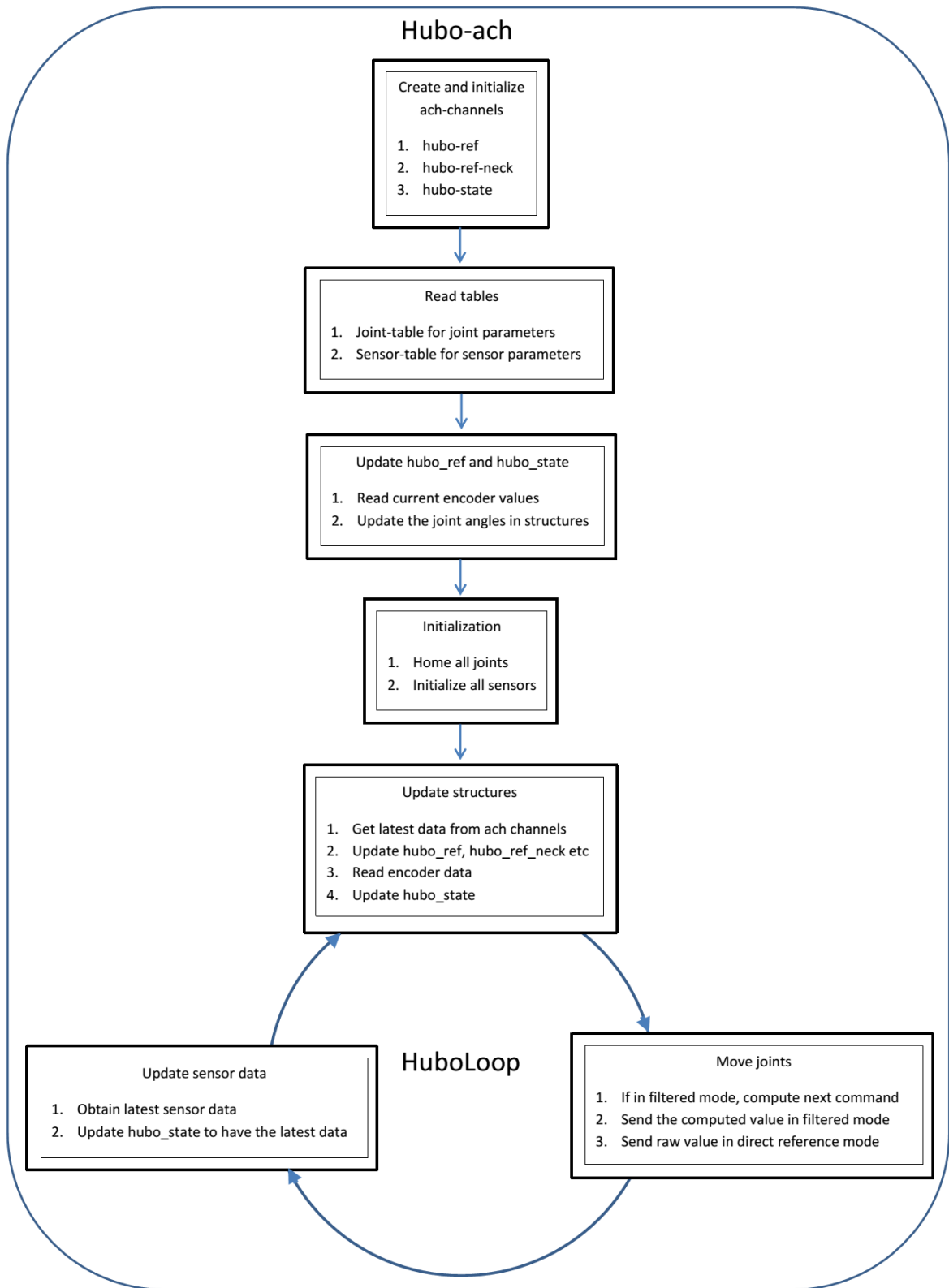


Fig. 3.15.: Organizational flow of `hubo_ach`.

‘sensor.table’ into the respective sensor data structures. From the joint numbers and sensor numbers, it computes the board headers for the CAN messages. The loading of joint and sensor parameters and calculating CAN message headers has finished the first step of starting hubo-ach.

Next step in the process is to update the `hubo_ref` and `hubo_state` data structures to have the current encoder values. This is easily done by obtaining the CAN data, decoding it and then getting the encoder values from each joint. The updated `hubo_ref` and `hubo_state` channels are put into the `ach` channel for third-party software to know the change. This step is a part of the 200 Hz frequency `huboLoop` as well.

Next, the control loop of hubo-ach termed as “huboLoop” starts execution. Within the loop, the latest data written to the `ach` channels, namely `hubo_ref`, `hubo_ref_neck` and `hubo_gains` are received. Next the encoder data from all the joints is received and the `hubo_state` data structure is updated. Finally it is time to send a command to the motors to move. As discussed in Section 3.4.3, the user is provided with two methods of joint control: filtered mode and direct-reference mode. When using the direct-reference mode, the `hubo_ref` data is directly sent to the motors. In the filtered mode, the data sent to motor is computed as:

$$computedReference = \frac{(encoder_value * (parameter - 1) + inputReference)}{parameter}$$

where *inputReference* is the desired joint position, *encoder_value* is the current joint position, *parameter* defines how densely the motion is interpolated, and *computedReference* is the position command that will be sent to the JMC boards.

The filter parameter is chosen as 40. This filtering means that the joint does not ever reach the actual value in `hubo_ref`. It reaches close to that value. So the filtered mode should be used only in a case when a slight error in motion is acceptable. Also, value 40 is chosen as it is optimized for Hubo2+ robot. For other humanoid robots, this value has to be tweaked. Theoretically, a better way to implement the filter would be to use linear or spline interpolation between the current encoder and the reference value. However, since the input trajectory is not expected to have very big

jumps (the difference in encoder value and reference value is expected to be less than 0.3 radians), a linear interpolation suffices.

Apart from the filter or no-filter mode, hubo-ach also supports the PWM-control mode. The main purpose of having the PWM-control mode is to provide pseudo-compliance control. Suppose the robot is trying to grasp an immovable object like the staircase's handrail. In position-control mode (also called as rigid mode), the joint angles are almost as desired. Due to errors in perception, it is possible that the computed joint angles are causing the robot to push into the hard and immovable rails. Not only does this over-torque some joints, but also this poses the risk of physically damaging the robot. To avoid this, the PWM-control mode is provided. Thus, if the applied PWM is limited, so the applied torque by the joints which in turn limits the force applied on the rails. Thus robot safety is not compromised. Though as discussed in Chapter 2, in the PWM-control mode, the JMC boards are sent the pulse-width data, for user-convenience the user is just required to put in the desired joint angle. Hubo-ach reads up the Proportional-and-Derivative (PD) controller gains from the table 'joint.table' and internally computes the PWM width (Eqs. (3.2)-(3.4)) and sends it to the JMC boards.

$$kP_error = Kp * (commanded_angle - current_pos) \quad (3.2)$$

$$kD_error = Kd * angular_velocity \quad (3.3)$$

$$duty_cycle = -(kP_err - kD_err + pwmCommand), \quad (3.4)$$

where Kp and kD are the proportional and derivative gains of the PD controller, *commanded_angle* is the desired position of the joint, *current_pos* is the current position of the joint, *angular_velocity* is the current angular velocity of the joint, and *pwmCommand* is an offset that the user can choose to give. As demonstrated later, this *pwmCommand* provides the basis for the gravity compensation controller. By default it is set to zero. If the gains are set high, then the joint follows the motion more accurately, but provides lesser compliance. If the gains are set low, then the

joint provides more compliance, but does not follow the motion accurately. Thus the gains must be chosen appropriately.

Before starting a controller and controlling the motors, it is necessary to calibrate them. “Homing” is the term given to initialize the motors and calibrate the encoders. “Initializing” is the term given to calibrate the IMU and force/torque sensors. Before any motors can be moved, homing them is necessary. For safety of the robot, the motors do not respond at all until “homed”. After commanded to “home”, the motor firmware takes control. The controller board moves the motors until the limit switch is pressed. Based on when the limit switches pressed, the encoders are calibrated. Similarly before using the sensors, they need to be initialized. Initialization basically changes the offset to make the current reading zero. Thus sensor initialization should be done only when the robot is stationary and hanging in air. Hubo2+ robot should be stationary to ensure that the angular velocity of the IMU sensors is zero. It should be hanging in air to ensure that there is no force acting on the force/torque sensors. Once homing and initialization are done, a custom-controller can easily move the robot, and the received feedback will also be correct.

After sending the commands to the motor, the board commands (to initialize, “home”, or change mode, etc.) are sent to the motors. Then, in consecutive steps encoder data, force/torque sensor data, IMU sensor data and power consumption data of the robot are all received. The `hubo_state` data structure is then updated to have these values. Figure 3.15 is an illustration of the complete process.

Thus, in essence, the main control loop sends the `hubo_ref` data (with or without interpolation) to the motors, sends commands to the boards to initialize, “home” or changes modes, and updates the `hubo_state` to have the latest encoder data, sensor data, and status of the joints.

3.5 Hubo-console and Hubo-read: Wrappers around hubo-ach

Section 3.4 discussed the design and implementation of hubo-ach. In this section, two wrapper programs that are implemented around hubo-ach to provide simple user-interface is discussed. The power of modularity of hubo-ach is demonstrated through these software wrappers.

Hubo-read:

Monitoring the state of a humanoid robot is essential to ensure the safety of the robot. It is also useful in debugging and analyzing the performance of a controller. Hubo-read is designed so that the user can see the state of the robot and log it for future reference. To monitor the state of the robot, hubo-read needs to just obtain the `hubo_state` data structure. Using Ach library, this can be easily done as described in Section 3.4.2. Hubo-read just needs to access the `hubo-state` ach channel and print out the data onto the terminal console. Algorithm 1 describes the working of hubo-read.

Algorithm 1 hubo-read

Access the `hubo-state` ach channel created by hubo-ach

while running **do**

 Read `hubo_state` data structure from `hubo-state` ach channel

 The `hubo_state` information is printed onto the terminal console

 Wait for time corresponding to the frequency

end while

Since hubo-read just reads and prints the `hubo_state` data structure, the wrapper can be used for any package and is not specific to Hubo2+ robot.

Hubo-console:

Hubo-read was developed to monitor the state of the robot. Hubo-console on the other hand is used to send commands to move the motors and initialize the sensors. To send such commands, hubo-console needs to access the `hubo-ref` and `hubo-board-cmd` ach channels. Hubo-ach reads data from these ach channels, encode them and

send them to the controller boards via CAN card. Algorithm 2 describes the working of hubo-console. Figure 3.16 shows the integration of hubo-read and hubo-console with hubo-ach.

Algorithm 2 hubo-console

Access hubo-ref and hubo-board-cmd ach channels created by hubo-ach

while running **do**

 Wait for the console input from the user

 Parse the console input

 According to the command update hubo_ref and hubo_board_cmd data structures

 Share the two data structures with hubo-ach via hubo-ref and hubo-board-cmd ach channels respectively

end while

Similar to hubo-read, hubo-console has no commands specific to Hubo2+ robot, and hence can be used for any humanoid robot.

Building custom controllers

In order to use a custom controller with hubo-ach, the controller should access the hubo-ref, hubo-board-cmd, and hubo-state ach channels. Then the controller can write trajectory data to hubo_ref data structure and send it through the hubo-ref channel to the robot. The controller can obtain feedback by reading the hubo_state data structure from the hubo-state ach channel and tweak the planned trajectory as required. Thus, using hubo-ach a user can easily and quickly develop a controller. If required, the controller can also reset the motors and change the control mode by writing to the hubo-board-cmd ach channel. Algorithm 3 describes how a custom controller should be developed.:

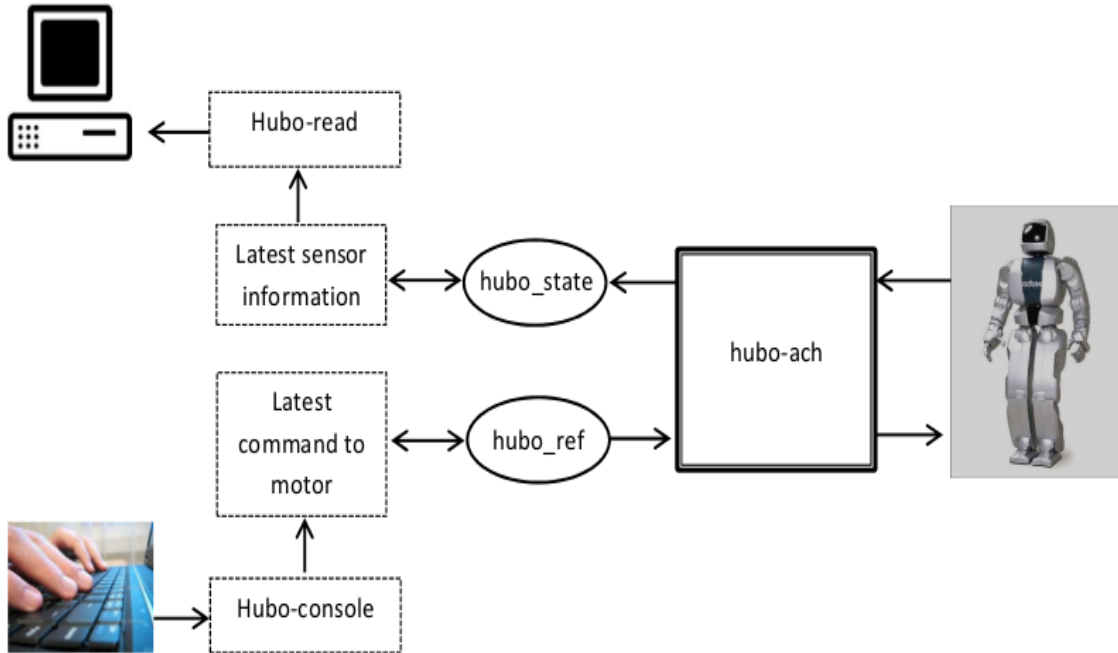


Fig. 3.16.: Depiction of how hubo-read and hubo-console interact with hubo-ach

Algorithm 3 generic controller

Access hubo-ref hubo-board-cmd and hubo-state channels

Plan trajectory

while trajectory is not over **do**

 Update hubo_ref to current trajectory set-point

 Send the new hubo_ref to robot via hubo-ref channel

 Read the latest hubo_state from hubo-state ach channel

 If required, reset the motors by writing to the hubo-board-cmd ach channel

 Generate the next trajectory set-point

end while

3.6 Summary

This chapter discussed the need of hubo-ach in Section 3.2. Section 3.3 discussed the architecture of ROS and analyzed why ROS is not optimal for real-time control.

A concise introduction to the Ach library used in Hubo-ach was provided in Section 3.4.2 to better understand its design. The architecture, design and various data structures used to develop the software were discussed in Sections 3.4.3, 3.4.4, and 3.4.5. Finally, two wrappers packages hubo-read and hubo-console for user interaction were discussed in Section 3.5. This laid the foundation to discuss how a user can design and implement custom controllers for a robot. In the next chapter, a higher-level humanoid robot controller called hubo-motion-rt is discussed. Some features and controllers developed as a part of that software and how it depends on hubo-ach is also be analyzed. Chapter 5 discusses some software packages based on hubo-ach and hubo-motion-rt. Their significance and relevance to humanoid robotics is also be explained.

4. ARCHITECTURE OF HUBO-MOTION-RT: A HIGH LEVEL HUMANOID ROBOT CONTROLLER

4.1 Introduction

Chapter 3 has discussed the architecture of the low-level humanoid robot controller `hubo-ach`. This chapter focusses on `hubo-motion-rt`, an abstract humanoid robot controller that is built using `hubo-ach`. Section 4.2 discuss the purpose of developing `hubo-motion-rt` and Section 4.3.1 describes its architecture. Sections 4.3.2, 4.3.3, 4.3.4, and 4.3.5 discuss various features and controllers of `hubo-motion-rt`. How all the controllers are integrated together into one software package is studied in Section 4.3.6. The methodology for developing controllers using `hubo-motion-rt` is explained in Section 4.4. Section 4.5 is summarizes the chapter.

4.2 Purpose of `hubo-motion-rt`

Humanoid robots are expected to perform very complex tasks. Locomotion tasks like walking on rough terrain, climbing a ladder, avoiding a fall in windy conditions, etc. require a robust balance controller. Similarly manipulation tasks like throwing a ball, painting a car, playing sports like table tennis (also known as ping pong) require precise control of the velocity of various joints. Torque control in the joints is required for compliant manipulation control, surgical applications etc. Last but not the least, just using PWM mode to control the joint position leads to high error in joint angles due to the gravitational force. To counter the gravitational forces extra torque needs to be applied. Computation and application of this extra torque to counter gravitaional forces is called gravity-compensation. Balance controller, joint velocity controller, joint torque controller, and gravity-compensation are required in many

different tasks. Implementing the same controller repeatedly for different applications is nothing but duplication of efforts. To avoid this duplication of effort, it is necessary to have generic high-level controllers, which can be used for different robots as well as different applications. Hubo-ach provides a real-time and modular framework for controlling humanoid robots. It provides an elegant framework to develop controllers. However, no such complex controllers are a part of hubo-ach software package. With the intention to provide such a generic framework for high-level control of humanoid robots, hubo-motion-rt has been developed. Hubo-motion-rt implements a few high-level controllers that are common to a wide variety of tasks.

The motion planning for humanoid robots can be done by following one of the following two approaches:

1. Whole-body motion planning: In this approach, motion planning of the whole robot is done all together. Planning does not differentiate between any two joints [73].
2. Locomotion and manipulation planning: In this approach, the robot motion is divided into two motions. The trajectory of the lower-body joints is planned to achieve desired locomotion state of the robot. The trajectory of the upper-body joints is planned to successfully grasp objects [74].

Whole-body motion planning takes into account all the joints. Thus the planner has more variable (joint angles, angular velocities, and angular accelerations). It is more general than the other approach. On the other hand, dividing the motion into locomotion and manipulation planning simplifies the planning algorithm. During locomotion, the upper-body of the robot is not allowed to move. During manipulation, the lower-body of the robot is stationary. However, due to these assumptions, some of the tasks that require coordination between the upper-body and the lower-body of the robot cannot be achieved. In spite of the assumptions, due to its simplicity, the approach of dividing the motion into locomotion and manipulation is often employed. Hence, motion planning of humanoid robots is often done by splitting the robot

motion into upper-body motion for object manipulation and lower-body motion for balance and locomotion. Therefore, the controllers for humanoid robots should also be developed along these lines. Hubo-motion-rt is also built with the same philosophy.

With the intention of abstracting such necessary controllers, hubo-motion-rt was developed. Open-source implementation of commonly used controllers is expected to benefit many researchers all over the world.

4.3 Architecture of hubo-motion-rt

4.3.1 Overview of hubo-motion-rt

Hubo-motion-rt is based on the same principle as hubo-ach. Similar to hubo-ach, to ensure that the control software is real-time, modular, and abstracts out the built-in controllers, hubo-motion-rt also follows a multi-process architecture. Due to low latency in inter-process communication, Ach library is used to communicate within various processes of hubo-motion-rt and with hubo-ach. Other software packages that use hubo-motion-rt are also expected to interact with it using the same inter-process communication library.

Since motion planning generally splits the motion into upper-and-lower-body, hubo-motion-rt also follows the same philosophy. It has three processes, one for the upper-body motion, one for the lower-body motion, and the third merges them into one and sends the motion to hubo-ach. First process, called manipulation-daemon, controls the upper-body manipulation of the robot. It is responsible for providing high-level controllers that maneuver objects. Forward kinematics and inverse-kinematic solutions are also implemented in manipulation-daemon. The second process, called balance-daemon, is responsible to ensure the stability and locomotion of the robot. This process, using force/torque sensors and IMU sensors, ensures that the robot does not fall down. Balance-daemon moves the robot to the desired position and orientation. The third process, required to merge the commands from the first two processes, is called control-daemon. It interacts with hubo-ach and ensures that

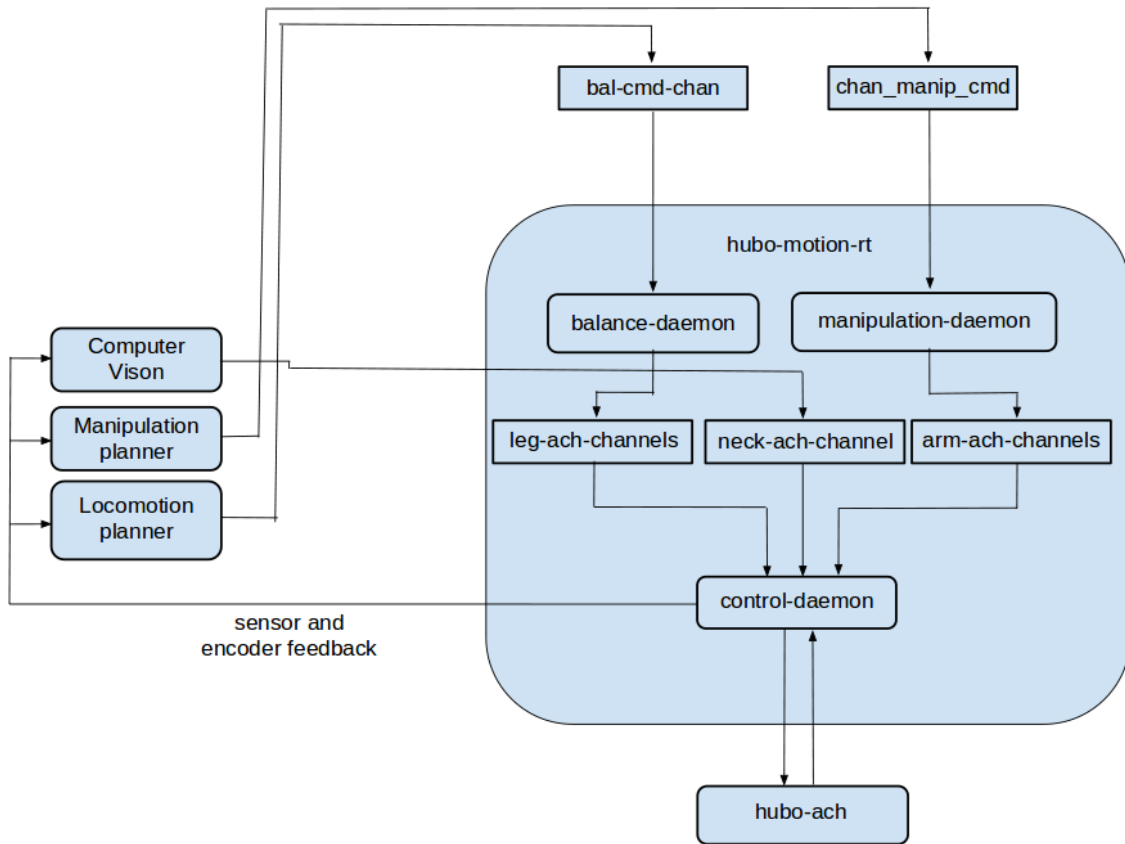


Fig. 4.1.: Architecture of `hubo-motion-rt`.

the desired trajectory sent to `hubo-ach` is smooth. It implements some commonly required controllers like joint velocity controller, torque controller, etc.

The `balance-daemon` via an ach channel called `bal-cmd-chan`, takes input from other process about the desired location of the waist and the locomotion state of the robot. When commanded, the `balance-daemon` executes the walking motion so that the robot reaches the desired location and then moves the waist to the desired angle. Then the `manipulation-daemon`, which obtains input via `chan_manip_cmd` ach channel, moves the upper-body of the robot to accomplish the required task. The `control-daemon` receives input via numerous ach channels. Ach channels for each of the four limbs and fingers on both the hands are created. Through these

ach channels, control-daemon obtains inputs about the desired motion and sends appropriate commands to hubo-ach. Figure 4.1 depicts the architecture of hubo-motion-rt.

4.3.2 Abstracting out ach communication for sensor feedback

Hubo-motion-rt is designed to provide a wide range of inbuilt features to a user. They range from basic manipulation controls to complex controllers. To obtain the latest sensor and encoder information, in hubo-ach a user had to open the hubo-state ach channel, read hubo_state data structure from it and parse it to obtain required values. Almost every controller needs to obtain the latest sensor readings and send the latest commands to the ach channels. Therefore, it is useful to abstract the ach communication. Hubo-motion-rt converts the above process into a single function, thus reducing the effort required by a user to implement a custom controller.

- `getJoint<param>`: This is a set of functions that take joint number as input and are used to obtain its information. For example, to obtain the encoder value of left elbow (LEB) joint, `getJointAngle(LEB)` function is used. Similarly, to obtain the joint velocity of the left shoulder pitch (LSP) joint, `getJointVelocity(LSP)` is used.

Often for humanoid robot controllers, obtaining a single joint angle is not as important as obtaining the joint angle for the whole limb. Thus, the following set of functions have been developed:

- `get<side><limb><param>`: This gives the information of a parameter of all joints in a limb. As an example, `getRightLegAngles()` would return a vector of the current motor board reference values of the right leg motors. Users can also obtain the encoder values or angular velocity of the joints as well.

Similarly, functions to get the force/torque sensor and the IMU sensor values have also been developed.

- `get<sensor><param>`: This returns the latest reading of a force/torque sensor's required parameter. For example, `getRightHandMx()` returns the moment about x -axis of the force/torque sensors on the right hand of the robot.
- `get<param><axis>`: This returns the IMU sensor reading of the waist IMU sensor about the input axis. For example, `getRotAngleX()` returns the angular velocity about the x -axis of the waist IMU sensor.
- `get<side>Tilt<axis>`: This returns the inclination with respect to a horizontal surface of the said axis with the IMU sensor mounted on the said foot.

4.3.3 Joint velocity control and torque control

Joint velocity control and torque control are important features required in various manipulation tasks. Therefore by abstracting them, the efforts required by researchers to develop new controllers or implement new planning algorithms can be significantly reduced. Just like functions mentioned in Section 4.3.2, abstracted out the reading the latest `hubo_state` data structure from `hubo-state` ach channel, wrapper functions to abstract out controlling a joint and thus moving the robot have been developed. `Hubo-ach` allows a user to only send position commands using the position-control mode or PWM-control mode to the robot. `Hubo-motion-rt` provides a much more elegant and powerful way to control the joints. It not only provides those features, but also lets the user control the velocity of a joint or the torque applied by the joint. For robust control, each joint is associated with two parameters:

1. Nominal (angular) speed
2. Nominal (angular) acceleration

Whenever a joint is commanded to move, in order to ensure smooth motion, `hubo-motion-rt` accelerates the joint at nominal acceleration until it reaches nominal speed. Then it continues to move at that speed for a while and finally decelerates

at the nominal acceleration value to reach a stop at desired location. Thus even for big jumps in joint angles, the joint motion is smooth (Fig. 4.2). Often robot controllers need to move a joint at constant angular velocity. Hubo-motion-rt has a joint velocity controller. When using it, a joint accelerates at nominal acceleration and reaches the desired angular velocity. For robot safety, the desired joint angular velocity is bounded by the nominal speed.

- `setJoint<param>`: This is used to change the angle, angular velocity, nominal speed or the nominal acceleration of a joint.

It is important to note here that the Hubo2+ robot's hardware does not inherently support joint velocity control. Thus, to move a joint at a certain velocity, hubo-motion-rt sends out new position commands at appropriate times. If however, another humanoid robot's hardware supports velocity control, then the joint velocity controller and hubo-ach should be appropriately modified.

To move a joint at a constant angular velocity, new position commands are sent to the robot. The position commands increment by the product of the desired angular velocity and the time before which the last command was sent out. For safety, the joint velocity does not abruptly change. The joint velocity increments at the rate of nominal acceleration until the desired velocity is obtained. Equations (4.1)-(4.5) mathematically describe how joint velocity control is achieved

$$dV_{initial} = desiredVelocity - u \quad (4.1)$$

$$dt = currentTime - lastTimeInstance \quad (4.2)$$

$$dV = \begin{cases} -a & , \text{if } dV < -a * dt \\ a * dt, & \text{if } dV > a * dt \\ dV_{initial}, & \text{otherwise} \end{cases} \quad (4.3)$$

$$newVelocity = u + dV \quad (4.4)$$

$$desiredAngle = originalAngle + newVelocity * dt \quad (4.5)$$

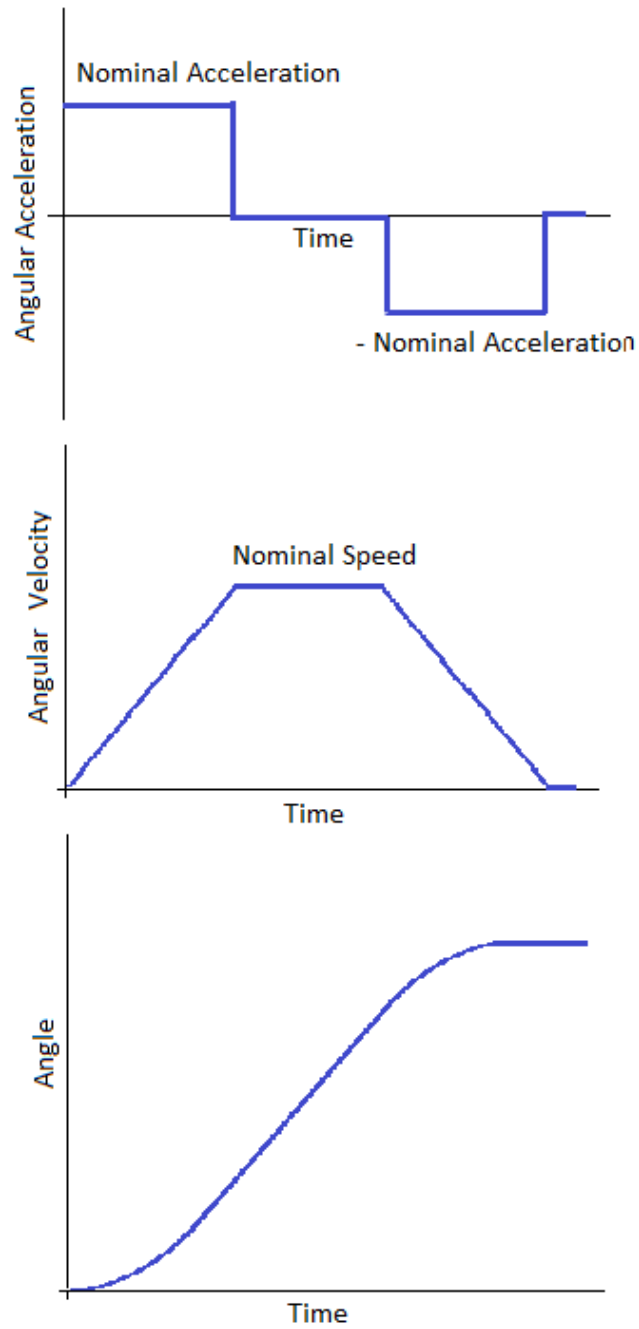


Fig. 4.2.: Schematic of joint motion in position-control mode using hubo-motion-rt.

where *desiredVelocity* is the desired angular velocity of the joint, *u* is the current joint angular velocity, $dV_{initial}$ is the difference between desired angular velocity and current

angular velocity of the joint, *currentTime* is the current system, *lastTimeInstance* is the last system time instance when this loop executed, *dt* is the difference between *currentTime* and *lastTimeInstance*, *dV* bounds *dV_{initial}*, *a* is the nominal acceleration of the joint, *newVelocity* is the new angular velocity of the joint, *originalAngle* is the current joint angle, and *desiredAngle* is the angular position that should be sent to the motor in order to obtain desired joint angular velocity.

Similarly, in the position-control mode of hubo-motion-rt, the joint accelerates at nominal acceleration to attain nominal velocity, cruises at that velocity, then decelerates at nominal acceleration and halts. Equations (4.6)-(4.11) mathematically describe the calculations for achieving joint position control.

$$dR = desiredAngle_{initial} - currentAngle \quad (4.6)$$

$$dV_{initial} = desiredVelocity - u \quad (4.7)$$

$$dt = currentTime - lastTimeInstance \quad (4.8)$$

$$dV = \begin{cases} -a * dt, & \text{if } dV < -a * dt \\ a * dt, & \text{if } dV > a * dt \\ dV_{initial}, & \text{otherwise} \end{cases} \quad (4.9)$$

$$newVelocity = \begin{cases} sqrt(2 * a * dR) - u, & \text{if } u > sqrt(2 * a * dR) \\ u + dV, & \text{otherwise} \end{cases} \quad (4.10)$$

$$desiredAngle = \begin{cases} desiredAngle_{initial}, & \text{if } newVelocity * dt > dR \\ currentAngle + newVelocity * dt, & \text{otherwise} \end{cases} \quad (4.11)$$

where *desiredAngle_{initial}* is the desired angular position of the joint, *originalAngle* is the current angular position of the joint, *dR* is the difference between *desiredAngle* and *originalAngle*, *desiredVelocity* is the desired angular velocity of the joint, *u* is the current joint angular velocity, *dV_{initial}* is the difference between desired angular velocity and current angular velocity of the joint, *currentTime* is the current system, *lastTimeInstance* is the last system time instance when this loop executed, *dt* is

the difference between *currentTime* and *lastTimeInstance*, dV bounds $dV_{initial}$, a is the nominal acceleration of the joint, *newVelocity* is the new angular velocity of the joint, *sqrt* is the standard mathematic function to obtain square root of a number, and *desiredAngle* is the angular position that should be sent to the motor in order to smoothly reach the desired angular position.

For various manipulation tasks, applying an appropriate amount of force is required. Drilling through a wall, throwing a ball along a desired trajectory, implementing gravity-compensation, etc. require robust torque control. Torque control, which is equally significant as position and velocity control, is discussed next. As mentioned in Chapter 2, Hubo2+ robot's hardware does not provide torque control or the PWM-control mode. But, DRC-Hubo robot's hardware supports the PWM-control mode. Using the PWM-control mode, a torque controller has been developed.

- *setJointTorque* : This function is used to apply a particular amount of torque on a joint.

To implement this, first a look-up table called Torque-PWM table is created. This table lists torques generated by various PWM values for all the joints, and stores them into a file that is loaded during the startup of *hubo-motion-rt*. During the run, when a torque control is required, the software looks into the file to find the desired value of torque. If the value is not in the look-up table, then the software chooses values directly higher and directly lower than the desired torque and finds the duty cycle to produce them. Using linear interpolation, the duty cycle to obtain the required torque is computed. Due to the friction and stiction, a joint does not move for low PWM values. So the offset PWM to move the joint is added. This computed PWM is finally applied to the robot through *hubo-ach* (Eq. (4.12)).

$$dutyCycle = \frac{(dutyUpper - dutyLower) * (Torque - TorqueLower)}{(TorqueUpper - TorqueLower)} + offset \quad (4.12)$$

where *Torque* is the required amount of torque, *TorqueUpper* is the torque directly higher than *Torque* in the Torque-PWM look-up table, *TorqueLower* is the torque directly lower than *Torque* in the Torque-PWM look-up table, *dutyUpper* is PWM-width required to produce *TorqueUpper*, *dutyLower* is PWM-width required to produce *TorqueLower*, *offset* is the offset required to counter frictional and stictional forces, and *dutyCycle* is the desired PWM-width required to produce *Torque*.

If the humanoid robot's hardware supports torque control, then instead of PWM commands, appropriate torque-control messages should be sent to hubo-ach.

4.3.4 Gravity compensation

The need for gravity-compensation

As discussed in Chapter 3, the PWM-control mode for the joints is very important for grasping rigid and immovable objects. However, the PWM-control affects accurate motion of the joint. The error in commanded and actual position increases with lower gains.

It is necessary to quantify the error in following the desired motion trajectory during different control modes. To estimate the error, the reference angles and the encoder angles of all the joints of the robot were logged using hubo-read. The data was parsed and the difference between the two was plotted for all joints. The experiment was conducted for the two control modes: the PWM-control and the position-control mode. Error was defined as follows:

$$error = \sum_{i=1}^n abs(encoder - reference) \quad (4.13)$$

where n is the number of trajectory setpoints, *encoder* is the current angular position of joint i of the robot, *reference* is the commanded angular position of joint i of the robot, *abs* is the standard mathematical function to obtain absolute value of a number, and *error* is the sum of absolute values of difference in *encoder* and *reference* over the complete motion.

The error in various joints during the motion execution in the PWM-control and the position-control mode is plotted in Fig. 4.3 and Fig. 4.4, respectively. From the figures, it is clear that the error in the PWM-control mode is significantly higher than in the position-control mode. To quantify the error, the term *errorDuringTrajectory* was defined as the norm of vector obtained when the error for each joint is added over the entire time series.

$$\text{errorSumVector} = \text{A vector of length number of joints.} \quad (4.14)$$

$$i^{\text{th}} \text{index of errorSumVector} = \text{error in joint } i \quad (4.15)$$

$$\text{errorDuringTrajectory} = \text{norm}(\text{errorSumVector}) \quad (4.16)$$

where *errorSumVector* is the vector of size equal to the number of joints in the robot, i^{th} index of *errorSumVector* is the *error* in joint i during the motion (Eq. (4.13)), and *errorDuringTrajectory* is the standard mathematical function to compute *norm* of a vector. Larger error in tracking the desired motion results in larger *errorDuringTrajectory* and vice-versa.

The *errorDuringTrajectory* was computed for ladder-climbing motion run in the PWM-control as well as the position-control mode.

PWM-control mode:

$$\text{errorDuringTrajectory} = 0.2372$$

Position-control mode:

$$\text{errorDuringTrajectory} = 0.0036$$

$$\frac{\text{errorDuringTrajectory (PWM-control mode)}}{\text{errorDuringTrajectory (Position-control mode)}} = 65.486$$

where the *errorDuringTrajectory* is calculated as in Eq. (4.16).

The error in position of various joints when controlled using the PWM-control mode and the position-control mode, has been plotted against time in Fig 4.3 and Fig 4.4. respectively. As can be seen from Fig 4.3, Fig. 4.4 and the computation of errors, the error in case of the PWM-control mode is significantly higher than the

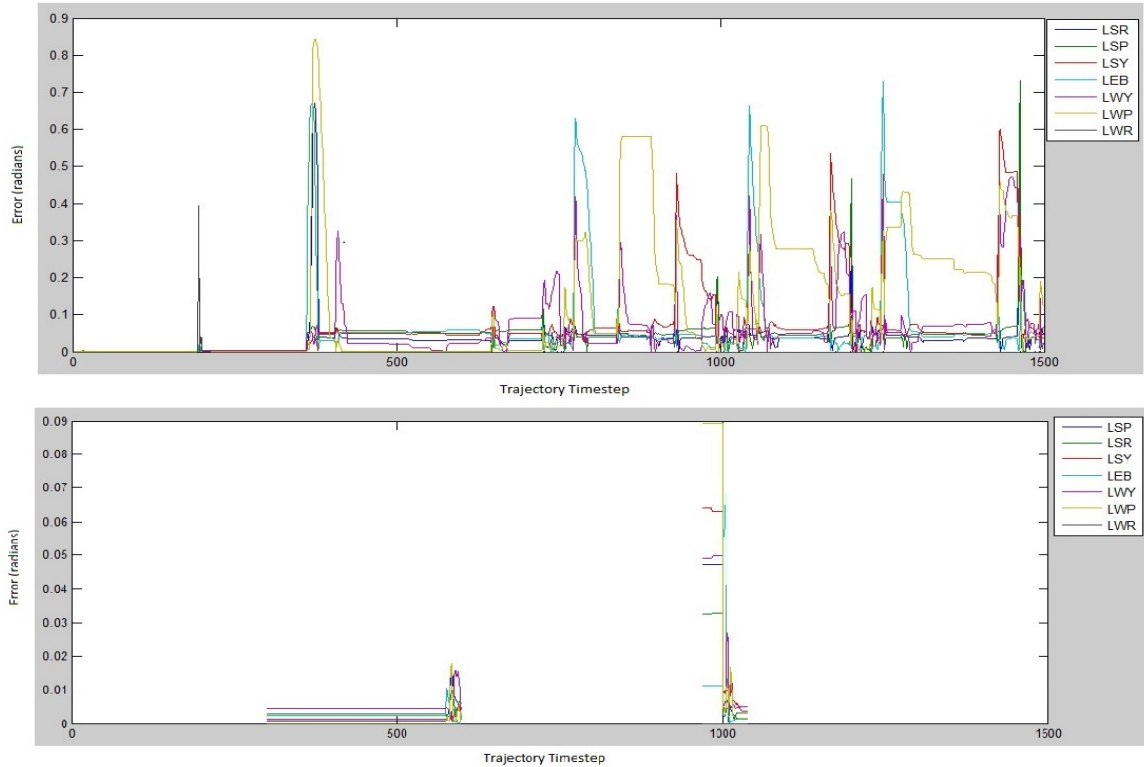


Fig. 4.3.: Error in the joints of left arm when the motion is executed using the PWM-control mode (top) and the position-control mode (bottom).

error in case of position-control mode. This error during the PWM-control mode can result in failure to grasp or improper grasping.

Designing gravity-compensation controller

To solve this problem, gravity-compensation has been designed. In a gravity-compensation controller, the torque to balance the robot in its current configuration is calculated. This term is not computed in the basic PWM-control mode. The motion control is based on a proportional-derivative (PD) controller. The torque to move from the current location to the new location is obtained using k_P (proportional) and k_D (differential) gains of the controller. The sum of the two torques is applied to the joints. Due to more accurate modeling, this leads to much more accurate motion.

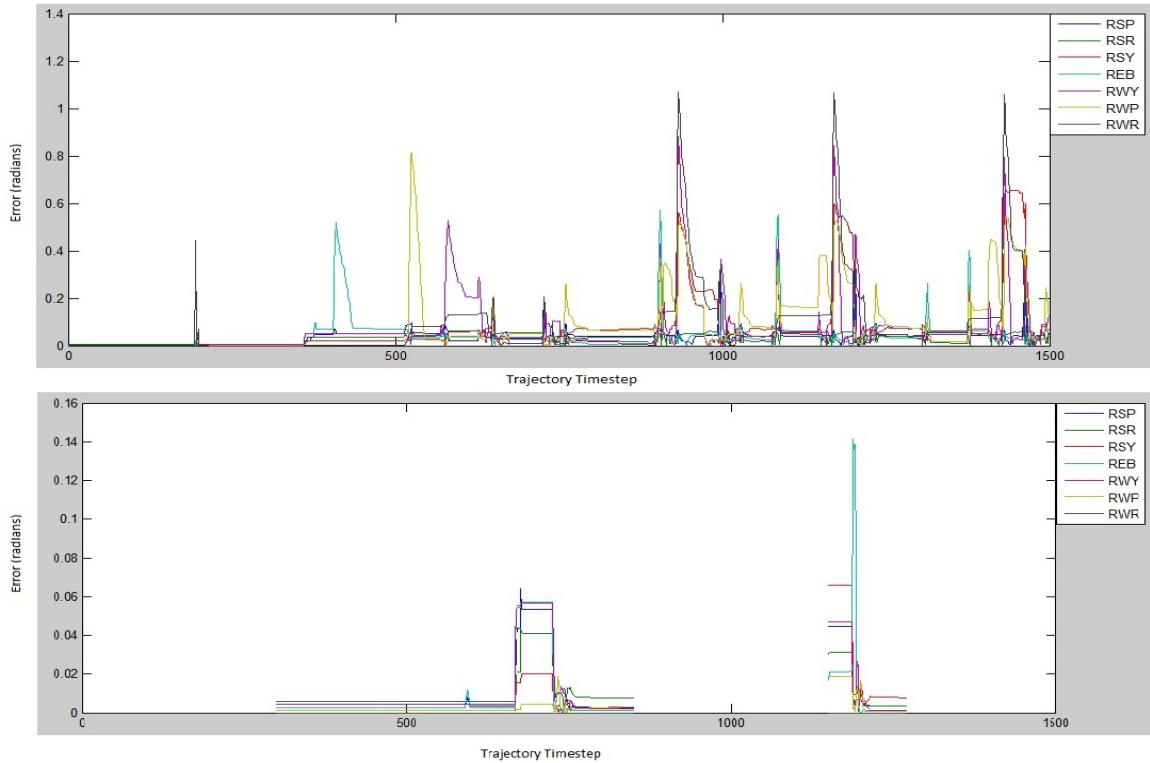


Fig. 4.4.: Error in the joints of right arm when the motion is executed using the PWM-control mode (top) and the position-control mode (bottom).

In order to compute the torque required at each joint, it is assumed that the robot is stable in the current configuration. Thus, the feet of the robot are fixed. Then the center of mass, moments of inertia of each link and the D-H parameters are obtained from the model of the robot. The robot's inclination with gravity is obtained from the IMU sensor mounted at the waist. Using the Newton-Euler recursive computation method, the torque required to counter the gravity is computed. Next, using Jacobian matrix, the joint torque required to apply requisite amount of force at the wrist to perform a manipulation task is computed. The PWM value corresponding to the sum of the two torques is found from the Torque-PWM look-up table and sent as the PWM parameter to hubo-ach. In the default mode, the commanded angle is also set as the current angle. Thus there is no correction torque and the robot stays in the

set position until moved by external forces. When external forces move the robot, the robot maintains the configuration at which external forces cease to exist. Figure 4.5 is the data-flow diagram of gravity-compensation controller.

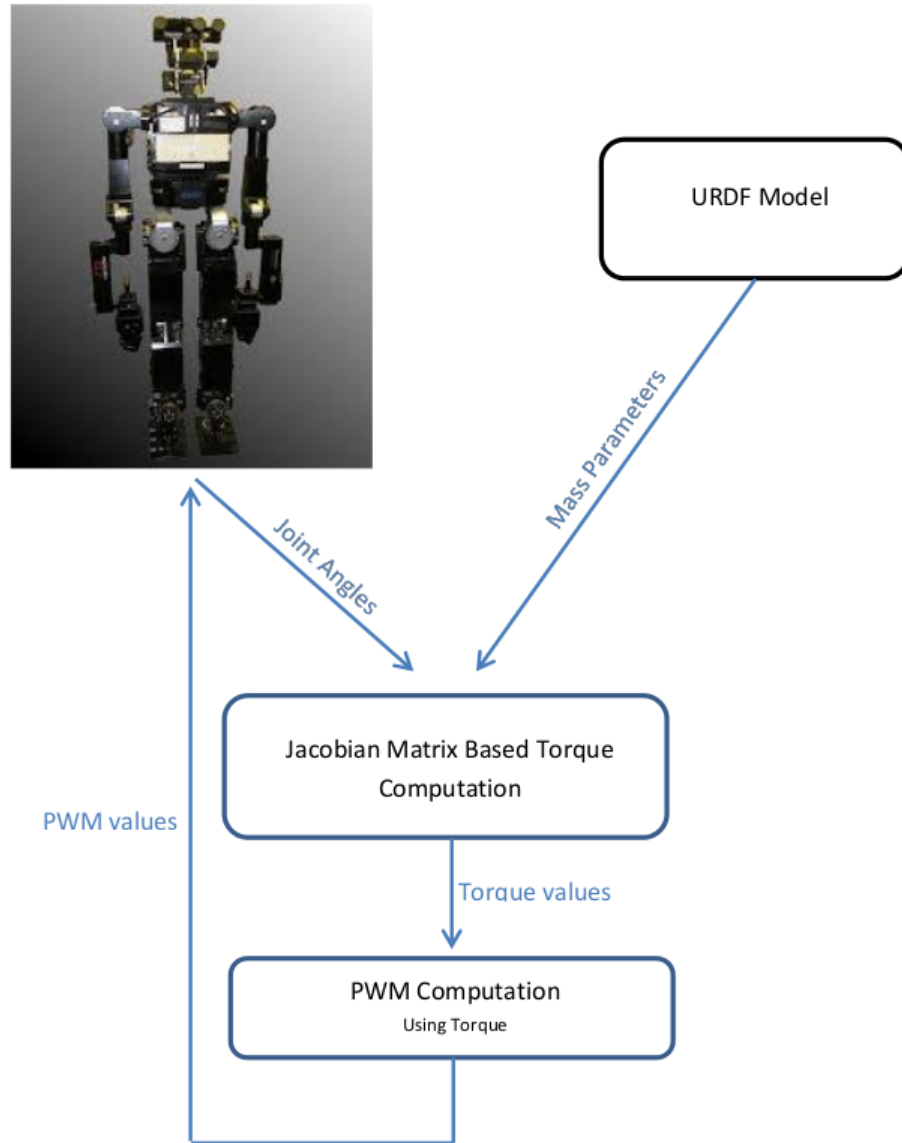


Fig. 4.5.: Block diagram of gravity-compensation controller.

4.3.5 Balance controller

To successfully complete any manipulation task, the humanoid robot has to remain stable. For a rigid object, a support polygon is defined as the smallest convex polygon that includes all the points of contact with the ground. It is also the region over which the center of mass of the object should lie to achieve static stability. To ensure balance of the robot, there is a need to have a controller that ensures that the center of mass of the robot lies within the support polygon and the feet are in complete contact with the ground. Satisfying these two conditions ensures that the robot does not topple down. The balance controller is designed to achieve this. The lower-body joints affect the center of mass position more than the upper-body joints. Constraining the upper-body joints affects the robot's manipulation capabilities. So under normal operation, the balance controller is designed to manipulate only the lower-body joints. Only in special circumstances, like during locomotion, the balance controller manipulates the upper-body joints as well. Another point to note is that to have a dynamic balance controller, the joint-trajectory of the robot must be known. Since this balance controller is designed to be stand-alone, therefore, it is a static balance controller. That means, the motion of the robot should be executed at a very slow speed.

A basic strategy to successfully manipulate an object is to move the robot within the object's reach and then use the upper-body to grasp and move the object. For the object to be within reach of the arms, the hips position is restricted. The location of hips is dependent upon the leg joints and so is controlled by the balance controller. In short the balance-controller achieves the following goals:

- Center of mass is within the support polygon.
- The feet are aligned and level with the ground.
- The hips reach the desired location.

The controller first obtains the x-offset and the height of the hips as inputs. Both the hips are always maintained at the same height. Though technically it is possible to let the two hips be at different heights, keeping the hips in the same horizontal plane simplifies the balance control without significantly affecting reachability of the robot. X-offset is the distance by which the weight of the robot should be shifted forward or backward. The location of center of mass of the robot largely depends on the hip position. Thus these inputs together with the current joint angles are used to calculate the linear velocity of the hip required to bring the center of mass within the support polygon and in particular at the desired x-offset location. The required hip, knee and ankle's angular velocities are calculated from the hip velocity. Using the torque reading of the force/torque sensors mounted on both feet, the ankle roll and pitch velocity to align feet with the ground are calculated.

The issues for the robot stability have thus been discussed. To move the hips to a desired height, the difference between current and desired knee positions are calculated. Using appropriate gains, knee velocity is calculated. To keep the center of mass in the same vertical line, half of this velocity should be subtracted from the ankle and hip pitches. Thus the new velocities are calculated and set as the desired joint velocities. Figure 4.6 is the data-flow diagram of the balance controller.

4.3.6 Integration of all controllers into one package

Section 4.3.1 provided an overview of hubo-motion-rt. Sections 4.3.2, 4.3.3, 4.3.4, and 4.3.5 described some features and controllers that are a part of the software. This section discusses the integration of all the controllers and features to form a single software package.

As mentioned in Section 4.3.1, the manipulation-daemon is responsible for the upper-body control. The balance-daemon is responsible to ensure the robot balance and attain the required hip location. The control-daemon obtains input from both these or any other processes and sends it to hubo-ach. To elegantly divide the motion

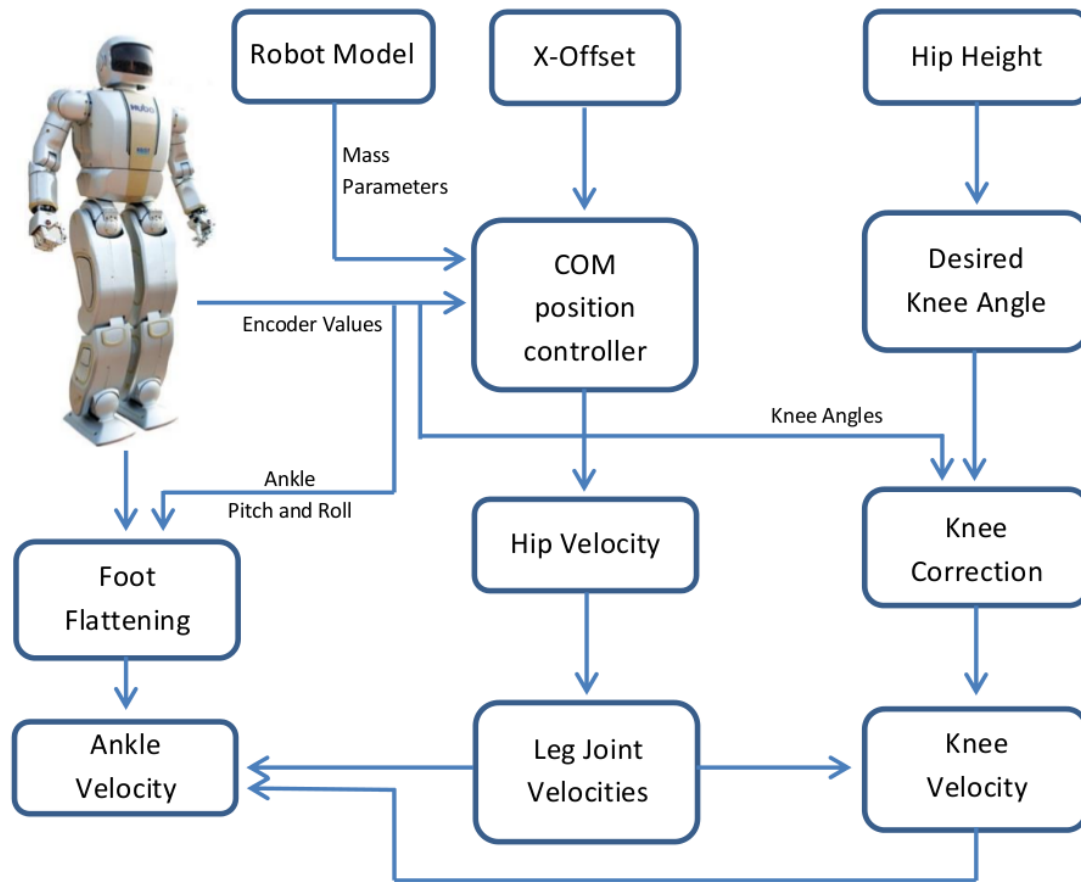


Fig. 4.6.: Data-flow diagram of the balance controller implemented in balance-daemon.

of joints, six ach-channels are created. Four of them are for the limbs (left leg, right leg, left arm, and right arm) and two are for fingers (left hand fingers and right hand fingers). Corresponding to the ach channel is also a data structure used to store desired position, desired velocity, nominal velocity and acceleration, PWM-control gains parameters, various error flags, joint indices in hubo-ach, the number of joints in that limb and a flag to store whether the limb is active or not. In general the manipulation-daemon writes only to the arm and finger channels and the balance-daemon writes to the legs channel. However, in some special circumstances like during

locomotion, the balance-daemon can take control of the upper-body joints as well. In that case, the balance-daemon sends an override signal to manipulation-daemon. The manipulation-daemon, after receiving the override signal, sends an acknowledgment signal back to the balance-daemon and stalls its operations. The balance-daemon then controls the upper-body joints as well. When done, it stops the override signal and both the manipulation-daemon and the balance-daemon resume normal operation. Meanwhile the control-daemon, reads the latest data in these channels, process that and sends it to hubo-ach. Figure 4.7 depicts how the various processes are integrated into one software package.

- **Manipulation Daemon:**

The manipulation-daemon is responsible for the upper-body motion of the robot. Thus, it writes only to the arms, fingers and waist of the Hubo2+ robot. To allow third-party software to easily integrate with hubo-motion-rt, the manipulation-daemon creates and reads from the `manipulation_command` ach channel. The data in this channel is shared through a data structure called `hubo_manip_cmd`. The data structure stores various manipulation related parameters like:

- KinematicMode: The mode of manipulation. Is it the desired pose or desired joint angles?
- PWM-control Settings: Should the arms be in the position-control mode or the PWM-control mode?
- Pose: The desired pose of the arms. This is used only when the manipulation mode is ‘pose’.
- Arm_angles: The desired joint angles. This is used only if the KinematicMode is chosen as desired joint angles.
- Grasp: At what point should the grasping be done.

Apart from these, there are other parameters as well, like waist angle, the allowed error limit is position etc.

The manipulation-daemon in a loop reads the latest data in manipulation-channel. Based on the various parameters, it fills in the data structure to store arm related data. In order to reach the desired pose, inverse kinematic computations need to be done using the equations discussed in Appendix A. These calculations are also done within this process. Finally it writes the desired joint angles or velocities and the mode of operation (PWM-control/position-control mode) to the ach channels for the arms. The control-daemon processes the data and sends it to hubo-ach.

- **Balance Daemon**

The balance-daemon is responsible to keep the lower-body of the robot stable. It controls the lower-body joints. It implements the balance-controller discussed in Section 4.3.5. To obtain the information about desired `x.offset` and height parameters from an ach-channel called `bal.cmd.channel`, any third-party software using the balance-daemon can just write to the ach channel and achieve the robot balancing.

- **Control Daemon**

The control-daemon reads the latest data in the ach channels to share arm, leg and finger data. It clubs all that data together and performs wide range of checks like ensuring that the trajectory is smooth, joints are within the range, etc. It also converts the joint velocity control commands to appropriate position commands. Similarly, it converts torque-control commands to appropriate PWM-width commands. Finally, it writes the computed data into the `hubo_ref` data structure and shares with with hubo-ach using the `hubo-ref ach` channel. Then hubo-ach reads the new ref data and moves the robot.

4.4 Writing custom software using **hubo-motion-rt**

Controllers written in `hubo-motion-rt` may not be sufficient for users. Users would like to design and develop their own controllers. Like `hubo-ach`, `hubo-motion-rt` is

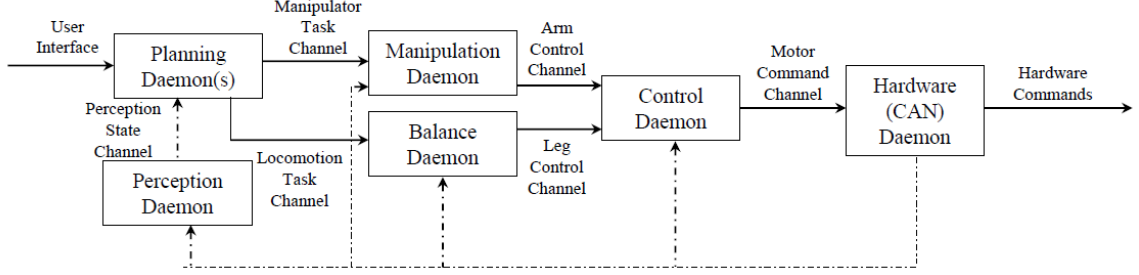


Fig. 4.7.: Communication between various processes in hubo-motion-rt [57].

designed in a way such that other software packages can easily integrate it as a submodule. From the architecture, it is clear that hubo-motion-rt takes two inputs. One is the hip location that goes to the balance-daemon and the other is the commands to move the upper-body via the manipulation-daemon. Thus a custom software that uses hubo-motion-rt has to open two ach channels: manipulation-daemon and bal_cmd_channel. Then based on sensor data and planner parameters, the data structures should be filled and sent to the ach channel. If the custom-software does not require the features in balance-daemon and manipulation-daemon, it can also directly write to the control-daemon. The pseudo-code of the controller is provided in Algorithm 4.

Algorithm 4 Custom controller using hubo-motion-rt.

Access the manipulation_cmd, bal_cmd_chan, arm, leg, and finger control ach-channels

Plan the manipulation trajectory and the hip location

while trajectory is not over **do**

Update the x_offset and hip height

Update the manipulation commands

Wait for the robot to reach close to the desired location

If required, write directly to the arm, leg, or finger ach-channels

end while

4.5 Summary

This chapter first discussed the need for a robot controlling software to be more abstract than `hubo-ach`. Section 4.3.1 provided an overview of the architecture of this software package. Sections 4.3.2, 4.3.3, 4.3.4, and 4.3.5 discussed various controllers and features available as a part of `hubo-motion-rt` and how they are implemented. The discussion ranged from some simply abstracting the `ach` communication to complex controllers like `gravity-compensation` and `balance-controller`. How all these controllers are integrated together was analyzed in Section 4.3.6. In Section 4.4, a generic method to develop other software packages that use `hubo-motion-rt` was provided. Chapter 5 discusses some software packages developed using `hubo-ach` and `hubo-motion-rt`. Their significance and usage are also discussed. Analyzing these software packages assists in understanding how multi-process architecture helps users to easily design new software.

5. APPLICATIONS AND USAGE OF HUBO-ACH AND HUBO-MOTION-RT

5.1 Introduction

Chapters 3 and 4 of this thesis focused on the architecture and design of hubo-ach and hubo-motion-rt, respectively. In both chapters, the development of custom controllers using hubo-ach and hubo-motion-rt was also discussed. This chapter discusses some controllers developed using hubo-ach and hubo-motion-rt. “Hubo-read-trajectory”- a software used to run pre-planned trajectories, “hubo-neck”- a software to control the neck joint for monitoring robot motion using the head-mounted camera, and “hubo-init”- a software to remotely control and monitor the robot and gravity-compensation based trajectory following are discussed in order. How hubo-ach can be integrated with ROS is analyzed using the software package “ros_hubo_ach”. This chapter is concluded with a discussion on how various features of hubo-ach and hubo-motion-rt are useful in design and development of a custom software for controlling humanoid robots.

5.2 Teleoperation of a robot

Teleoperation indicates operation of a robot from a distance. Robots may be expected to operate in inhospitable and dangerous areas. It is necessary to oversee the operation of the robot to ensure that it performs the required task remotely. In case of an error, the operator is expected to take corrective measures. Teleoperation has been used not only for controlling robots remotely, but also for monitoring unmanned air vehicles (UAVs), satellites, and controlling machines inside a nuclear reactor.

A wide variety of teleoperation tools has been designed and developed for monitoring and controlling robots. For example, some teleoperation tools have also been developed for humanoid robots [75–77]. However, such tools are specific to the type of robots. Therefore there is a need to develop a set of teleoperation tools, which are generic for humanoid robots and are not task-specific. Hubo-read-trajectory, hubo-neck, hubo-init, gravity-compensation-based trajectory following, and ROS_hubo_ach are five generic software packages developed for humanoid robots. Sections 5.3, 5.4, 5.5, 5.6, and 5.7 discuss each of them in detail.

5.3 Hubo-read-trajectory

Autonomous robots perceive the environment around them, plan the motion, and execute the motion. However before executing the planned trajectory, the software package used for planning (planner) needs to be tested. One of the methods to test the planner is to generate a motion plan, simulate it, and then execute it on the robot. After the planner has been tested, the controller can be modified for “planning on the fly”. It is also often useful to log the generated plan, so that the motion can be analyzed in case of failures. Thus, executing a pre-planned trajectory is a very important step to verify and debug a planner. Also, often the robot may be used to execute the same motion repeatedly. Motions in which robot does not physically interact with the environment are examples where motions can be re-used. Gesture motions like waving goodbye, clapping, fist pumping, etc. are some examples. Motions which are independent of the environment can be planned, stored on the body-computer of the Hubo2+ robot, and executed when required.

The need of executing a pre-planned motion is apparent from the above-mentioned situations. Hubo-read-trajectory was developed to fulfill this requirement. This package reads a file in which the desired motion of the robot is stored. Hubo-read-trajectory expects that the file is stored using space-separated values format. In a space-separated values file, the data items are separated using space as a delimiter.

The data items represent the desired joint angles. Therefore, each line of the file corresponds to one configuration of the robot.

The software is implemented using hubo-ach. Since the software just sends motion and does not expect any feedback, writing to the hubo-ref ach channel is sufficient. The program opens and initializes the hubo-ref channel, and then it opens the trajectory file. While the program has not reached the end of file, the program reads a line, parses it to fill hubo_ref data structure and sends the filled data structure to hubo-ach via the ach channel. It then waits for 5 ms (i.e. 200 Hz). Waiting for at least 200Hz is necessary as due to CAN card's bandwidth limitation, hubo-ach executes at that frequency. Sending hubo_ref data structure faster than 200Hz will not be useful as hubo_ach will ignore some of them. In case the package is used for running pre-planned motions on other humanoid robots, the frequency to write the motion to hubo-ref channel should be updated to that supported by the hardware. However, this is a very basic skeleton of hubo-read-trajectory. Adding some more features will make it a much more powerful tool. Figure 5.1 depicts the working of hubo-read-trajectory.

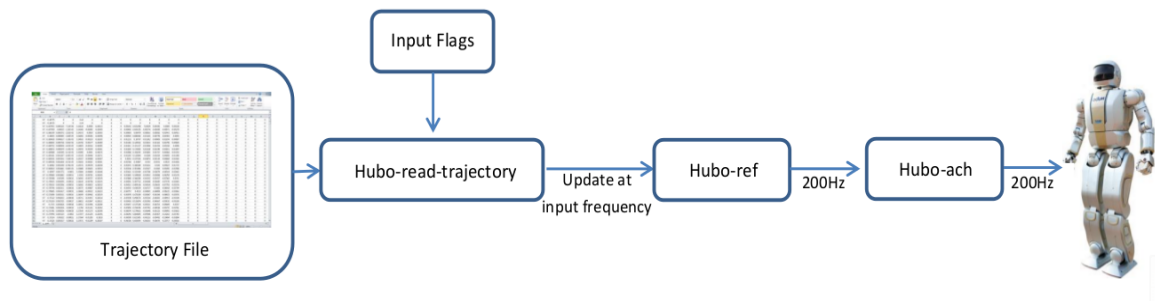


Fig. 5.1.: Schematic depicting data flow when using hubo-read-trajectory.

Some characteristics of hubo-read-trajectory are:

- The first set-point in the trajectory may not be the current robot position. So, if the trajectory is executed, sudden jerky motion can damage the robot. To ensure a smooth beginning of trajectory execution, the software should slowly

move the robot to the first configuration in the trajectory file. A feature called “goto init” feature achieves exactly the same. Hubo-read-trajectory executes a densely interpolated motion to move from current position to the first configuration in the trajectory file. The time taken to reach the first trajectory set point is 3 seconds by default.

- The user may want to run the motion in the PWM-control mode or in the normal position-control mode. So the ability to choose the control mode should also be available to the user. Also, for some tasks, the robot’s one arm may be holding a rigid immovable object, while the other arm will be free. Ladder climbing is one such example when one arm is grasping the ladder’s rail, while the other arm is reaching to a higher grasping position. In such a situation, it is desirable to have one arm in the PWM-control mode and the other arm in the position-control mode. Therefore the feature to put only one arm or both arms in the PWM-control mode or to execute the motion in the position-control mode is also required and provided in the software.
- The trajectory may be coarse or sparse; that is, the consecutive trajectory set points may or may not have huge jumps. A sparse motion can be executed without any danger to the robot. However, executing a coarse motion produces jerks, which are harmful to the robot. If the planned motion is coarse then the user should be able to execute the motion using the internal filter of hubo-ach described in Section 3.4.3. With this intention, hubo-read-trajectory provides a feature which allows the user to enable or disable the internal motion filtering of hubo-ach.
- Pausing a motion while it is being executed is a critical requirement. Not only it is required in case of emergency, but also, to analyze the cause for failure. When remotely operating, due to network delays, the user may want to wait for visual or sensory information. Pause feature is also critical in such a situation. So a feature to pause the motion and continue to play it from same set-point is

provided. A word of caution is to use this feature only if the robot is stable at every trajectory set point; otherwise, the use may pause the motion when the robot is not in a stable orientation, causing the robot to fall down.

- There could be instances where the user may want to test the performance of the trajectory with varying speeds. Generating the same motion with varying speeds is not desirable. To let the user use the same motion but execute it at different speeds, a feature to change the frequency of motion is provided.

5.4 Hubo-neck

A humanoid robot generally executes two different sets of algorithms. One set of algorithms is responsible for motion planning and control while the second is responsible for computer vision. Hubo-ach is designed so that the various algorithms run as different processes. In general, it is expected that the motion planning and control processes control the robot motion. However, the neck joints are an exception to this. Since the neck joints control the camera orientation, the computer vision algorithms should control these joints and position the camera.

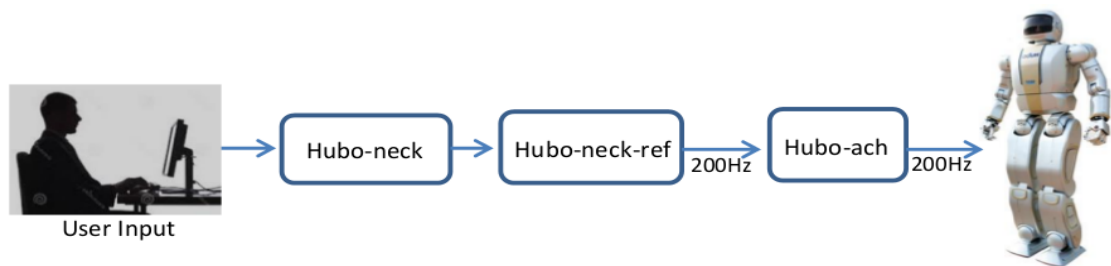


Fig. 5.2.: Schematic depicting data flow when using hubo-neck.

The user oversees the operation and pauses or stops the motions whenever the safety of the robot is compromised and uses the camera to monitor that the robot has grasped the object properly. Thus, there is a need to have a simple interface to control the neck. Hubo-neck is a software package designed so that the user can

easily control the orientation of the neck. The module opens the hubo-ref-neck ach channel and waits for the user input. To provide an interface that is familiar to most of the users, standard first person gaming control keys have been chosen. ‘w’, ‘a’, ‘s’, ‘d’ move the neck up, left, down, and right, respectively, by an angle of 0.3 radians. The angle was chosen to be 0.3 radian to ensure that there is a slight overlap between the last view and the current view. The neck pitch does not have a large rotation angle. It has a rotation range of only 0.7 radian. Hence two jumps of 0.3 radian are sufficient to cover the complete range of the neck-pitch motion. However, the neck yaw covers approximately 5 radians and thus, bigger rotation jumps in neck yaw may be required. Therefore ‘q’ and ‘e’ are used to move the neck leftwards and rightwards by 1 radian, respectively.

5.5 Hubo-init

A remote operation software can be used in two ways. In the first approach, the user can remotely access the robot’s body-computer using some standard software like TeamViewer or Microsoft Remote Desktop. A remote login software does not transfer the application specific data, but transfers the user-interface information. Thus the software consumes a lot of bandwidth and is not very efficient. The second approach is to develop two different software packages: a client side application that runs on the user’s computer and a server side application that runs on the robot. This approach shares only the application-specific data across the two computers. Hence, it is not data-intensive and should be used. Hubo-ach, as discussed in Chapter 3, is based on the inter-process communication library ‘Ach’. The library also provides with a feature to share process data across two computers. Since hubo-ach just reads from the ach-channels and is blind where the data is coming from, just using the remote data sharing feature of ach library in conjugation with hubo-ach running on the robot’s body-computer is sufficient for remote robot operation. Table 5.1 compares some of the standard software packages for remotely access.

Table 5.1: Comparison of various remote access software.

Remote Access Software Comparison	Data sharing	Bandwidth	Variable Update Frequency
Ach	Yes	Shared data size dependent	Yes
Team Viewer	No	Screen resolution dependent	No
SSH (X-11)	No	GUI size dependent	Yes
Microsoft Remote Desktop	No	Screen resolution dependent	No

Revisiting Ach

In Chapter 3, a brief introduction to Ach library for inter-process communication was provided. How Ach can be used to share data across different processes was discussed in Section 3.4.2. This section discusses the usage of Ach library to share data across two or more computers. When connected via a Local Area Network (LAN) to other computers, Ach library allows one computer to push its ach channel to the other or pull an ach channel from other computers. Thus, if Computer 1 is pushing ach channel C1 to Computer 2, then the latest update in C1 from Computer 1 will always be sent to Computer 2. Similarly if Computer 1 is pulling ach channel C2 from Computer 2, then the latest update in C2 from Computer 2 will be received in Computer 1. This remote synchronization feature of ach is very useful in developing software to remotely control the robot. It is important to note that since Ach uses TCP for remote synchronization, this feature is not real-time.

Hubo-init, a remote robot control software, is designed to help users remotely operate the robot. The graphical user interface (GUI) provided allows the user to monitor the joint motors and the sensors. The user can also initialize JMC boards, “home” the joints, and calibrate the sensors. Thus, if any joint goes into an error or warning state, or the sensor values are not as expected, the user can immediately notice the warning or the error, and take corrective measures.

Since hubo-init sends the homing or initializing sensor commands to the robot, the hubo-board-cmd channel of hubo-ach needs to be initialized. Similarly, to receive the encode values, the joint-error flags, and the sensor information, the hubo-state channel has to be pulled from the robot. Thus hubo-init creates these two channels and initializes them: hubo-board-cmd in the ‘push’ mode and hubo-state in the ‘pull’ mode. It is followed by the initialization of the GUI, which has multiple tabs. Upon clicking the home, reset or other commands for the joints, the hubo-board-cmd channel is appropriately filled and pushed to the robot. The hubo-ach running on the body-computer of the Hubo2+ robot, receives the latest ach messages from user’s computer. Hubo-ach running on the body-computer of the Hubo2+ robot sends appropriate commands to the JMC boards. Similarly, at a set frequency (default is 10Hz), the hubo-ach on the user’s computer updates the hubo-state channel. Hubo-init reads the latest hubo-state channel and displays the encoder and sensor information on the GUI. Screenshots of three different tabs of hubo-init are shown in Fig. 5.3, 5.4, and 5.5. The organizational flow of hubo-init is depicted in Fig. 5.6.

5.6 Gravity-compensation-based trajectory following

Though hubo-read-trajectory, based on hubo-ach, is a very useful and robust software, it shares the drawbacks of hubo-ach. The most significant of those is the lack of gravity-compensation. As discussed in Chapter 4, using only the PWM-control mode causes up to 65 times more error in joint angle than the position-control mode. Thus, for more accurate trajectory following, gravity-compensation should be used.

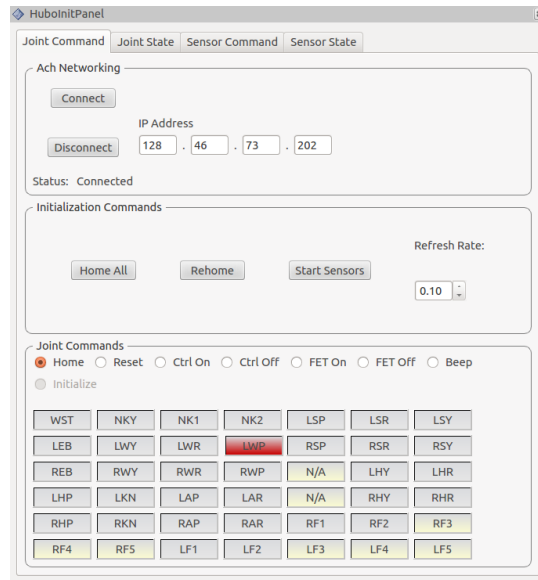


Fig. 5.3.: Screenshot of Hubo-init in use to home all the joints.

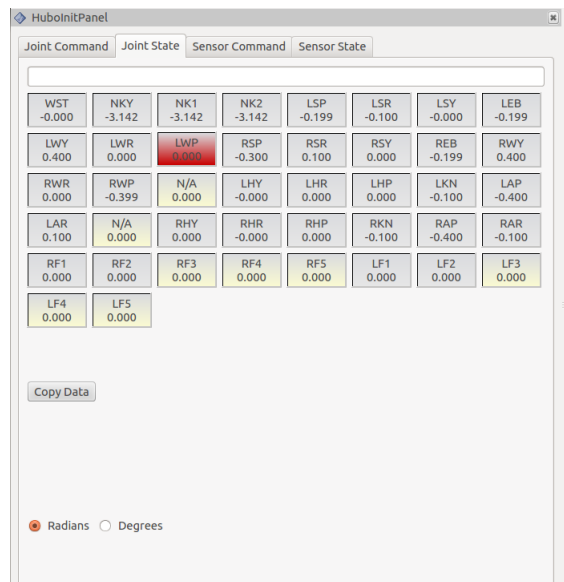


Fig. 5.4.: Screenshot of Hubo-init in use to monitor the robot joints.

For the sake of backward-compatibility, functionality similar to that of hubo-read-trajectory should be provided. Therefore, the software architecture is the same. However, instead of sending the joint-angle values directly to hubo-ref channel, they



Fig. 5.5.: Screenshot of Hubo-init in use to monitor the sensor values.

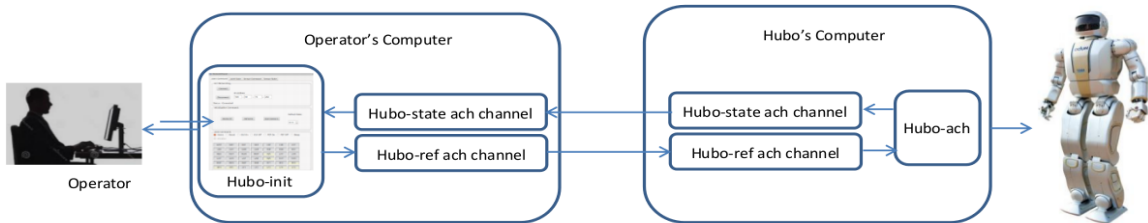


Fig. 5.6.: Working of hubo-init.

are sent to hubo-motion-rt. Hubo-motion-rt has three daemons. The balance-daemon modifies the leg-joint angles to ensure the stability of the robot. The manipulation-daemon computes the joint angles in the upper-body of the robot to successfully grasp objects. The pre-planned motion has the desired joint angles. When executed at a desired frequency, the motion is stable. The time to grasp and release grasping is also pre-planned. Hence, the balance-daemon and manipulation-daemon should not be used when using this software as they would modify the sent trajectory. The data should be directly written to the ach channels corresponding to arms, legs, and fingers.

Thus, hubo-motion-rt based trajectory follower reads and parses the trajectory files exactly as hubo-read-trajectory, but writes them to the arm, leg, and finger ach channels created by hubo-motion-rt. Hubo-motion-rt, after calculating the torque for gravity-compensation and the torque for moving the joints to desired position, adds them up and applies the resultant torque by following the method discussed in Chapter 4. Figure 5.7 depicts the organizational flow of gravity-compensation-based hubo-read-trajectory.

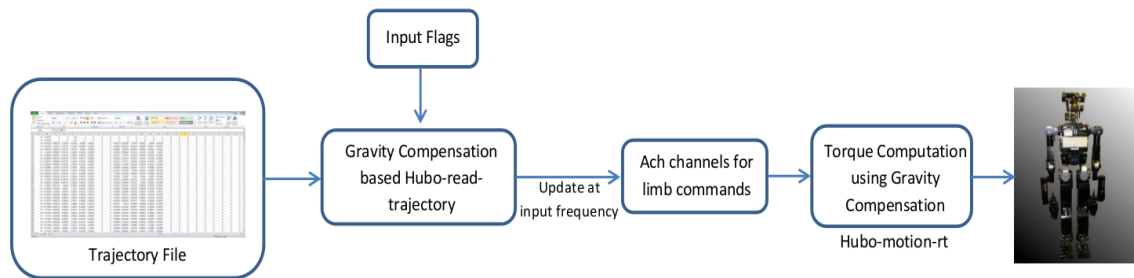


Fig. 5.7.: Schematic depicting execution of gravity-compensation based trajectory following.

5.7 ROS_hubo_ach

ROS is the most widely used existing software for controlling robots. Since it is not real-time software, it cannot be used for controlling humanoid robots. Due to the size of ROS users community, not only support for many robots is provided, but also a wide variety of controllers, motion planners, and features are available. Since a large number of users use the existing ROS packages, therefore they have undergone significant amount of testing. Thus, the likelihood of failure is low. Hence, for applications where real-time control of humanoid robots is not required, using ROS is a desirable option.

Different nodes of ROS communicate through ROS messages and ROS services. This architecture is similar to that of Ach communication. The difference is in pro-

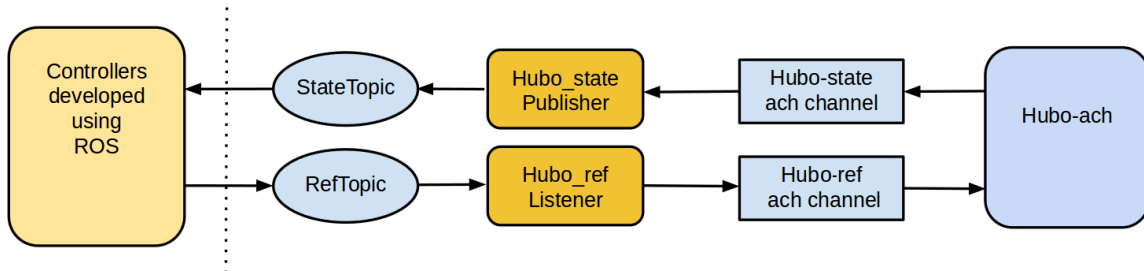


Fig. 5.8.: Schematic depicting how using hubo-ach, ROS can be used to control humanoid robot.

protocol for communication. ROS uses TCP whereas Ach uses mutex-based memory sharing. Thus, in order to use ROS for controlling a humanoid robot, it is sufficient to write ROS nodes which communicate with other ROS nodes through ROS messages, and communicate with hubo-ach through an ach channel. ROS_hubo_ach implements this design. The schematic of the design is provided in Fig. 5.8. Two ROS nodes called ‘Hubo_statePublisher’ and ‘Hubo_refListener’ are created. Hubo_statePublisher, obtains the latest hubo_state data structure from hubo-ach and publishes to ‘StateTopic’. StateTopic is a ROS topic which stores the latest hubo_state data structure. Similarly, Hubo_refListener reads the latest commands from ‘RefTopic’, and sends them to hubo-ach via hubo-ref ach channel. RefTopic is the ROS topic that stores the latest commands to be sent to the robot. Controllers developed using ROS communicate with Hubo_statePublisher and Hubo_refListener via the two topics.

Instead of using hubo-ach as the base controller, some researchers prefer to use hubo-motion-rt, then a similar architecture suffices. A ROS node to write the manipulation and balance commands to their respective ach channels (chan_manip.cmd and bal-cmd-chan respectively) should be created. Controllers and motion planners developed using ROS should communicate with this node through a ROS topic. Similarly, another ROS node is required to obtain the feedback and publish it to a ROS topic. Motion planners and controllers can obtain sensor feedback by listening to this ROS topic.

5.8 Summary

Chapters 3 and 4 discussed the architecture of hubo-ach and hubo-motion-rt, respectively. This chapter discussed how a custom software that uses hubo-ach and hubo-motion-rt can be developed. Five different software packages were discussed. They are “hubo-read-trajectory”, “hubo-neck”, “hubo-init”, “gravity-compensation-based trajectory following”, and “ROS_hubo_ach”. First the motivation behind developing the five software packages was discussed. This was followed by discussing the architecture and applications of the five software packages. By analyzing these packages, the power of multi-process architecture is obvious. It is easy to observe that upgrading one software package does not affect other packages. It is also obvious that abstracting out the CAN communication and sharing data through inter-process communication makes the development of custom controllers extremely easy. It is expected that by understanding these examples, a researcher will be able to quickly develop custom software that are based on hubo-ach or hubo-motion-rt. Chapter 6 presents some experiments conducted using the robot and discusses the results obtained from those experiments.

6. EXPERIMENTAL WORK ON DRC-HUBO TO TEST HUBO-ACH AND HUBO-MOTION-RT

6.1 Introduction

Chapters 3, 4, and 5 discussed the architecture of hubo-ach, hubo-motion-rt, and the development of custom software packages that use hubo-ach and hubo-motion-rt. Several experiments were conducted on the DRC-Hubo robot to verify the performance, robustness, and the ease of use of hubo-ach, hubo-read-trajectory, and hubo-neck.

Sensory feedback is essential for interacting with the environment. Theoretically, information from force/torque sensors can be used to know if the robot has grasped an object or not. In the first experiment, we study if the variation in force/torque sensors caused by the reaction forces can be used to detect if the robot has successfully grasped an object or not.

Ladder climbing is an important, yet difficult task. Access to many different buildings is only through ladders and staircases. Thus, in disaster situations, ladder-climbing is a necessary task that robots must perform. Though researchers have demonstrated the ability of humanoid robots to climb ladders, it has been done by using small scale humanoid robots, or using computer simulation. In this chapter we study the application of hubo-ach, hubo-read-trajectory, and hubo-neck for ladder-climbing task. Using the three software packages, we were successful in controlling a DRC-Hubo robot to perform ladder-climbing. Planning algorithms employed to obtain desired trajectory that a robot should follow in order to climb a ladder are also described.

6.2 Analysing the use of force/torque sensors for detecting grasping

For successfully manipulating and moving objects, it is necessary to know if the robot has grasped the object or not. It can be done in various ways: by using an IR distance sensor on the wrist, a force/torque sensor on the wrist, or using computer vision algorithms on the data received from the camera mounted on the head of the Hubo2+ robot. Since Hubo2+ robot does not have IR distance sensors and computer vision is computationally intensive, using force/torque sensors can provide an easy and reliable way to confirm the grasping.

When grasping objects, to counter the weight of the object, the wrist has to exert vertical force and thus the force/torque sensor reading will change. Similarly, when the robot grasps immovable rigid objects like the hand-rails of a ladder, significant variation in the force/torque sensor readings due to the reaction forces is expected. In this experimental study, we used the force-sensor data to detect if the robot has successfully grasped a ladder's railing or not.

To perform the experiment, `hubo-read-trajectory` and `hubo-read` were used in conjunction with `hubo-ach`. Path planning algorithm required to climb a ladder was executed on the operator's computer. The planned motion was transferred to the hubo's computer and executed. In parallel, using `hubo-read`, the joint encoder, and sensor data were logged. After the experiment, the force/torque sensor data was plotted against time. Instances when the robot was grasping the ladder and when it was not were marked.

In Fig. 6.1 the normal force F_z and the moments M_x and M_y have been plotted against time. The sensor reads have been plotted at configurations when the robot arm was grasping a rail as well as configurations when the robot arm was not grasping a rail. In the left column the graph of sensor data obtained from the force/torque sensor mounted on the wrist of the left hand of the robot has been plotted. Similarly, in the right column, the graph of the sensor data obtained from the force/torque sensor mounted on the wrist of the right hand of the robot has been plotted. From

Fig. 6.1, it is clear that there is almost no change in M_x and M_y reading throughout the climbing. Though F_z changes significantly throughout the motion, there is no particular pattern. The F_z reading for both right and left hands is always less than zero when the rail is not grasped. But when the hands grasp the rail, the readings vary a lot. So even though it may be possible to detect grasping in hindsight (higher variance), immediately detecting grasping is not possible. On referring to the joint encoder values obtained from hubo-read, it can be inferred that the variation arises mainly due to the forces exerted while climbing and not due to grasping. Thus, it is not easy to detect grasping using 3-axis force/torque sensors.

Since the experiment did not yield positive results, the head camera was used to monitor the grasping. The remote operator using hubo-neck software controlled the camera angle and monitored the grasping. If the robot was successful in grasping the hand-rail of the ladder, the user continued the robot motion. In case of failure to grasp, corrective motion was executed before continuing the motion.

6.3 Ladder climbing

Today, access to most of the buildings- residential or official, is through ladders or stairs. To respond to disaster situations in such buildings, it is essential that humanoid robots should be able to climb stairs and industrial ladders.

Ladder-climbing is a challenging task. Researchers have demonstrated the ability of humanoid robots to climb ladders [58–60]. However, those experiments were performed in a controlled and indoor environment. We have demonstrated that a full scale humanoid robot can successfully climb different ladders in outdoor environment. Outdoor environments in contrast to indoor environments, deals with uncertainties like sunlight, fluctuating wind speeds, etc. Also earlier, the ability to climb ladders was demonstrated using small humanoid robots or in computer simulation. In these sets of experiments the ability of a full-scale humanoid robot to climb industrial ladders has been demonstrated.

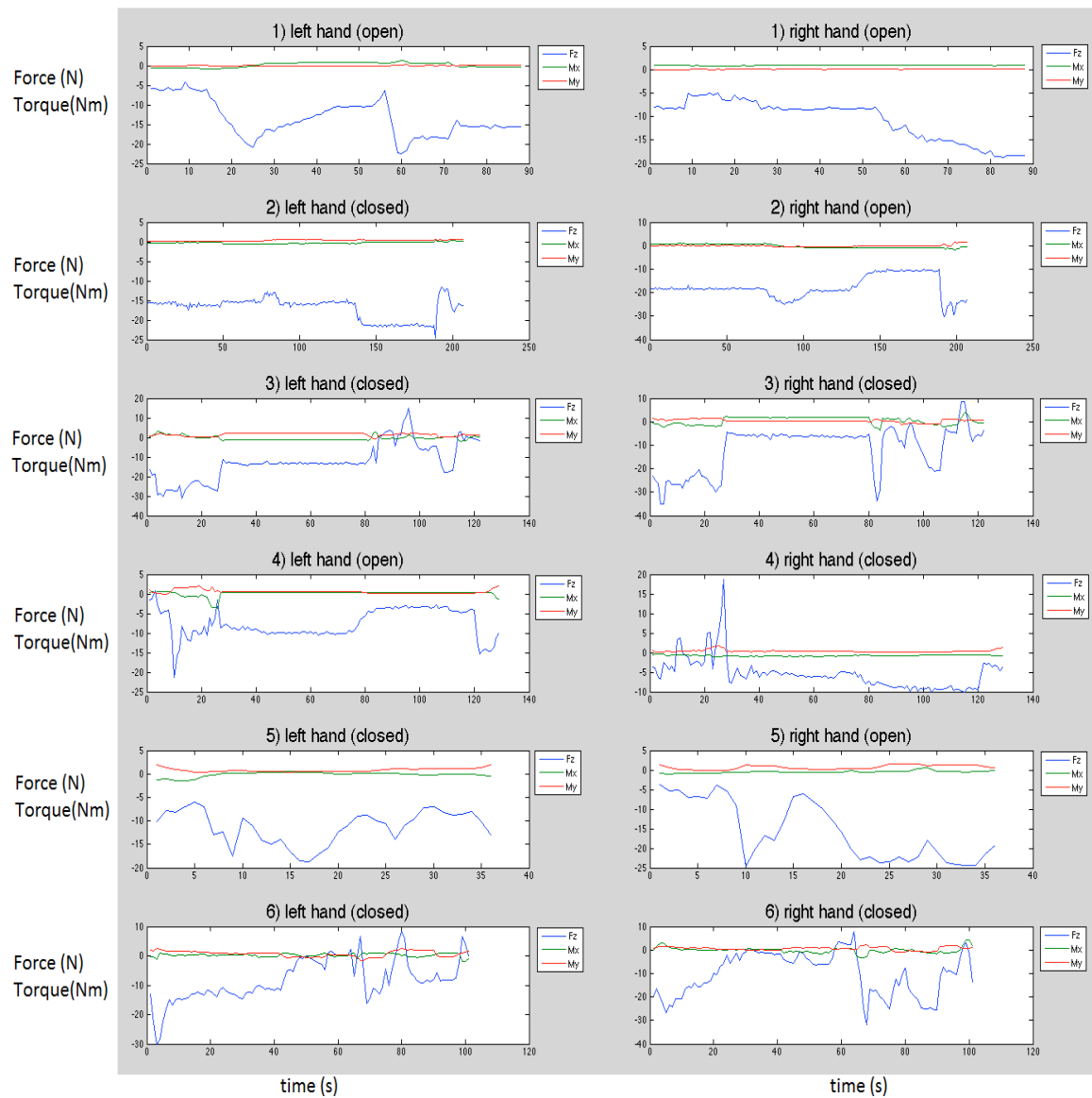


Fig. 6.1.: Force/torque sensor data when the rail is grasped (hand closed) and when it is not (hand open) during the climbing of the first step of the ladder.

For the set of experiments discussed in this thesis, the software systems discussed in earlier chapters were used by ARTLab at Purdue University and Intelligent Motion Laboratory at Indiana University to make the Hubo2+ and DRC-Hubo robots climb industrial ladders. The experiments were performed in two stages. Initially climbing experiments were performed on stairs with rails, followed by climbing two different ladders.

6.3.1 Motion planning for ladder climbing

A motion planner is required to obtain a trajectory that a robot should execute to finish a desired task. It takes the robot model, the environment model, a set of constraints, and the desired goal as inputs. In this case, the planner takes the robot model and the ladder specification as inputs. It outputs a motion plan that should be followed in order to make the robot climb the ladder. Motion primitives are the motions that change the state of contact of a limb of the robot. A motion planner for ladder-climbing motions based on the idea of motion primitives was designed and developed by Luo, Zhang, and Hauser, etc. [79–83].

A ladder consists of rungs and stringers. Rungs are equi-spaced and horizontal structures on which the robot places its feet. They can be cylindrical or cuboidal. Stringers can also be cylindrical or cuboidal and are supports for the rungs. Along with the shape and dimension of rungs and stringers, inclination of ladder with the floor and the number of rungs completely specify the ladder model. A model of the ladder is depicted in Fig. 6.2(a). Currently, this simple structure is used to model a ladder, but if required a more complex ladder can also be defined.

The term ‘hold’ is defined as the geometric region in which the robot and the environment are in contact with each other. ‘Stance’ is a set of holds. When standing on the ground, each foot of the robot touches the floor. So the stance of the robot consists of two holds. During climbing, a stance may consist of two, three or four holds (i.e., either two feet, two feet and a hand, two hands and a foot, or all four

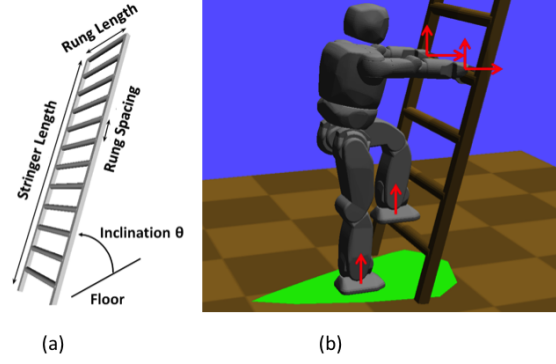


Fig. 6.2.: (a) Parameter specification of a 3D ladder model. (b) Point-contacts and their normals (red arrow). A support polygon (green region) can be calculated based on the contacts to check the stability of the robot.

limbs in contact). Since a stance with two holds has a smaller support polygon and will have a greater chance of the robot falling down- such a stance should be avoided. To model a contact region, a finite number of point-contacts are used. Let the points on the robot that touch the ground be r_1, \dots, r_k . Let the points of contact in the environment be x_1, \dots, x_k .

Hand holds are modeled as two point-contacts:

- Vertically oriented hold to allow the hand to push down.
- Horizontally oriented hold to allow the fingers to pull back.

The foot is expected to apply only vertical force to push the robot upwards and so foot holds are modeled as only one vertically oriented point contact. Coulomb friction is assumed with a known coefficient of friction ($\mu = 0.4$). Figure 6.2(b) shows the robot model, the support polygon, and the holds of the robot.

A feasible robot configuration q at a stance σ must satisfy the following constraints:

1. From the definition of contact points provided above, it is clear that a constraint that the points $r_1(q), \dots, r_k(q)$ meet the points x_1, \dots, x_k , for all holds in σ should be created.

2. To ensure feasible robot motion, the joints must be within their limits: $q \in [q_{min}, q_{max}]$.
3. The robot must not be in a state of collision with any objects in the environment.
4. There should not be collisions between different parts of the robot.
5. Since the planner is expected to generate a statically stable trajectory, then at every point the center of mass of the robot should lie within the support polygon formed of the robot.

A plan to climb the ladder consists of a sequence of stances $\sigma_1, \dots, \sigma_n$ and a continuous sequence of single-step paths of feasible configurations p_1, \dots, p_n that are feasible at their corresponding stance. During the execution of a motion primitive, a limb first loses contact with the environment and regains contact at a new location. This leads to two changes in the stance of the robot. In order to ensure a feasible motion plan, during the transition from stance σ_i to σ_{i+1} , the robot must pass through a configuration that meets the constraints of both stances.

Based on the contact with the ladder, a ladder climbing motion can be decomposed into following seven motion primitives:

1. placeHands: place two hands on a (chosen) rung.
2. placeLFoot: place left foot on the first rung.
3. placeRFoot: place right foot on the first rung.
4. moveLHand: lift left hand to the next higher rung.
5. moveRHand: lift right hand to the next higher rung.
6. moveLFoot: lift left foot to the next higher rung.
7. moveRFoot: lift right foot to the next higher rung.

Primitives 1–3 mount a robot onto the ladder, primitives 4–7 make the robot climb a rung. Therefore primitives 4–7 are repeated for every rung until the robot reaches close to the top of the ladder. Each motion primitive is designed to contain prior knowledge for solving that portion of the climbing task. It contains a set of point-contacts, robot poses, and intermediate way-points tailored to finish the desired action. Currently a human expert designs the motion primitives. Seeds are the motion primitives provided to the planner in order to find natural-looking contracts, poses and paths for various ladders. These seeds are the starting points for trajectory optimization. Since the planner is based on optimization, a good starting point is critical to avoid local minima and reduce the cost of optimization. Experiments showed that good seeds can greatly speed up the process of finding collision-free and stable paths [82, 83].

How to plan and utilize primitives 1–7 in sequence to climb up two rungs is described next. To climb multiple rungs, repeating primitives 4–7 is sufficient. This method is a randomized sequential descent in which each primitive is slightly perturbed from the seed values at random in order to help find successful solutions. To ensure that paths stay close to the seed primitives, the radius of perturbation starts at zero and increases upon subsequent iterations. This procedure is described by Algorithm 5.

Algorithm 5 Motion planning using motion primitives

```

1: while No solution is found and algorithm has not reached time limit do
2:   Let  $q_0 \leftarrow q_{cur}$  and  $\sigma_0 \leftarrow \sigma_{cur}$  .
3:   for Motion primitives, 1, 2,  $\dots$ , 7, do
4:     Sample a desired hold  $h_d$  near the seed hold.
5:     Let  $\sigma_i$  replace the current hold in  $\sigma_{i-1}$  with  $h_d$ .
6:     Sample a feasible destination configuration  $q_i$  at  $\sigma_i$ .
7:     Find a feasible path connecting  $q_{i-1}$  to  $q_i$  at  $\sigma_{i-1}$ .
8:   end for
9: end while

```

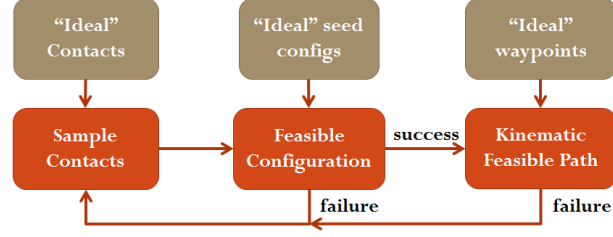


Fig. 6.3.: Three steps in motion planning for ladder climbing based on motion primitives. If one step fails, the planning process will trace back to the previous step. Prior information is being used to assist in each step.

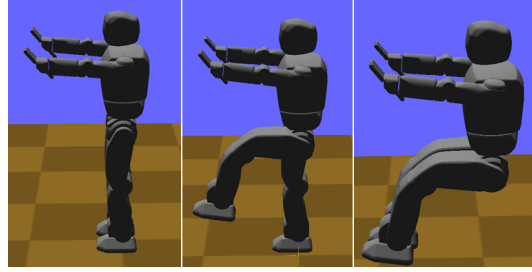


Fig. 6.4.: Seed examples for motion planner- from left to right: placeHands, liftLFoot, liftRFoot.

The innermost loop samples holds, configurations, and paths in that order. Failure of any step in the innermost loop leads to the planner restarting from step 2. Each innermost sampling step is run for n samples, where n controls the balance of putting more effort on one action or backtracking to get a better start. For this implementation, n is set to 50 after tuning.

Starting from a seed configuration q_{seed} , using numerical inverse kinematics (IK) solver, a configuration that satisfies IK and joint limit constraints is obtained. If this fails, using a perturbation function, q_{init} is modified. The perturbation is drawn uniformly from 0 to some empirically chosen radius c . The perturbation radius increases with the number of failures. The process stops when either a feasible configuration is found or the implementation reaches its iteration limit. Algorithm 6 describes the configuration-finding procedure.

Algorithm 6 Finding feasible configuration

```

for  $i = 0, 1, \dots, n$ : do
     $q_{init} = q_{seed} + \text{perturb}(i)$ 
    if find  $q$  from IK solver starting from  $q_{init}$ : then
        if no self-collision and no environment-collision and stable: then
            return  $q$ 
        end if
        if no self-collision and has envr-collision: then
            if retract( $q$ ) succeeds and  $q$  is stable: then
                return  $q$ 
            end if
        end if
    end if
end for
  
```

To generate trajectories that connect the starting and ending configurations of motion primitives, intermediate way-points to avoid collisions with the ladder rungs are added. By interpolating the endpoints of the moved limb along an arc in world space and perturbations to push way-points into the feasible space the way-points are obtained.

A linear interpolation in joint space does not suffice, because the robot fails to maintain contact at intermediate configurations. Instead a recursive interpolation is used to ensure that the supporting limbs remain in contact. Given two endpoint configurations, q_1 and q_2 , first the middle point $q = \frac{1}{2}(q_1 + q_2)$ in the joint space is computed. Then its projection q' in the contact space is calculated. The projection function uses numerical IK to find q' , which satisfies the IK constraints that $r_1(q'), \dots, r_k(q')$ meet x_1, \dots, x_k . Then the following two sub-problems are recursively solved:

1. Interpolation between q_1 and q' .

2. Interpolation between q' and q_2 .

The algorithm terminates the recursion once q_1 and q_2 are closer than the required distance (obtained from the user). Thus, using this planner, feasible and smooth ladder climbing motions are generated.

6.3.2 Execution of the planned trajectory

In order to perform ladder-climbing, the planned trajectory is pre-loaded into the robot. Remote operator executes the trajectory, monitors the joint warning states, the error states, and grasping using the head-camera. In case of any error, appropriate corrective trajectory is executed. After correction, the original trajectory should be resumed to make the robot climb a ladder.

Using the planner described in Section 6.3.1, motions to climb stairs and two different ladders were obtained. The specifications of these ladders are provided in Table 6.1. To verify that the motions are feasible, the generated trajectories were first executed in simulation (using Klamp) and then on the real robot. The experiments were conducted in three stages using DRC-Hubo robot (an upgraded version of Hubo2+ robot). In the first stage, based on the observation that the PWM-control mode leads to higher error, the motion was executed completely in position-control mode. Since this was not suitable for robot safety, during the next stage of experiments, the motion was executed with both arms being in the PWM-control mode throughout the climb. This posed no risk to the robot. But the errors in position often caused failure to grasp the ladder rails and thus led to failure in climbing the ladder without correction. In the final stage, a mixture of first two strategies was employed. To reach the rails, the position-control mode was used, but while grasping the PWM-control mode was enabled. This was not only safe for the robot, but also led to good success rate. Table 6.2 presents a comparison of the three strategies, discussing the grasping accuracy, damage frequency and the overall success rate.

Table 6.1: Comparison of various ladders.

Parameter	Stair	Ship ladder	DRC ladder
Slope (in $^{\circ}$)	40	58.1	60
Number of rungs	4	6	9
Distance between rails (m)	1.27	0.79	1.35
Rail height from rung (m)	0.99	1.02	0.91
Rung spacing (m)	0.203	0.254	0.305
Distance of first rung from ground (m)	0.172	0.264	0.305
Rung width (m)	1.22	0.762	0.812
Rung length (m)	0.25	0.178	0.102
Rung thickness(m)	0.04	0.038	0.032

6.3.3 Climbing using only position-control mode

In this set of experiments, the planned trajectory to climb the ladder was executed with all the joints in the position-control mode. The motion was pre-planned and the robot was manually (by human judgment) placed as close as possible to the starting pose in the planner. However, only in about 30% of the experiments, the robot was successful in climbing to the first rung of the ladder. On inspection, it was realized that the wrist roll joint turned off. It is important to recollect that the wrist the wrist roll joint of DRC-Hubo robot cannot be controller using the PWM-control mode. Since the fingers and the wrist roll joint are controlled by the same JMC board, the fingers turned off too. Thus they failed to hold onto the rail and the robot failed to climb the ladder. In addition to this, failure to place the robot perfectly at the initial position, resulted in extreme stress on some joints of the robot. This often physically damaged the wrists and the finger joints. Considering these problems, motion for ladder climbing was not executed using only the position-control mode.

6.3.4 Climbing using only PWM-control mode in the upper-body of the robot

During this set of experiments, the planned trajectory was executed with all upper-body joints in the PWM-control mode. Even in this case, the wrist roll joint turned off causing the fingers to turn off as well. To solve the problem, the wrist roll joint was turned off from the very beginning of climbing motion. Since the motors of the wrist roll joint were not powered, external forces could move the wrist roll joint. This essentially put the wrist roll joint in ‘complete compliance’. Since the grasping angle was the same for every step, changing the wrist roll joint angle was not required. Thus, turning off the wrist joint is a feasible solution. In case of error, the robot operator can turn on the wrist and move it to the desired angle. This strategy led to about 80% accuracy in climbing four steps of a ladder.

However, always using the PWM-control mode in the upper-body of the robot has its own drawbacks. As discussed in Chapter 4, the PWM-control mode has higher errors in joint position. Also the PWM-control depends on the battery voltage. This sometimes caused error in grasping the rails. Failure to grasp the rails led to failure in climbing the ladder.

6.3.5 Climbing partly using position-control and partly using PWM-control mode

The PWM-control mode in joint space is required when the joints are interacting with rigid and immovable objects. The ladder climbing motion planner can be divided into five stages:

1. Place left hand on a higher position on the rail.
2. Place right hand on a higher position on the rail.
3. Place left foot on the next rung.
4. Place right foot on the next rung.

5. Straighten the legs.

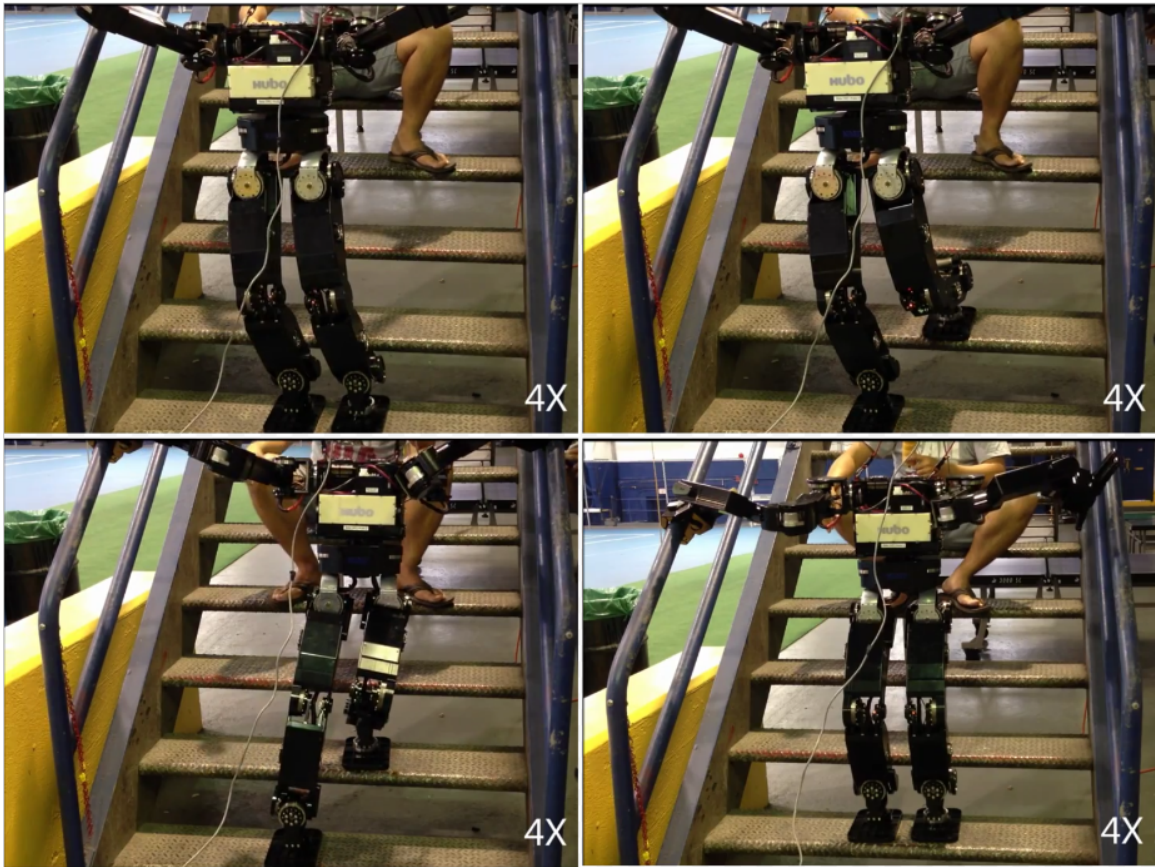


Fig. 6.5.: Snapshots of DRC-Hubo robot climbing stairs in the Armoury, Drexel University, Philadelphia [84].

During stages 3, 4 and 5, both the hands are grasping the rail and so they should be in the PWM-control mode. During stage one, while moving the left hand upwards, it is in the air. Thus for accurately placing it as the desired position, the position-control mode should be used, and not the PWM-control mode. Right hand should be in the PWM-control mode as it is in contact with the rail. Similarly, for stage 2, left hand has to be in the PWM-control mode and right hand should be in the position-control mode.

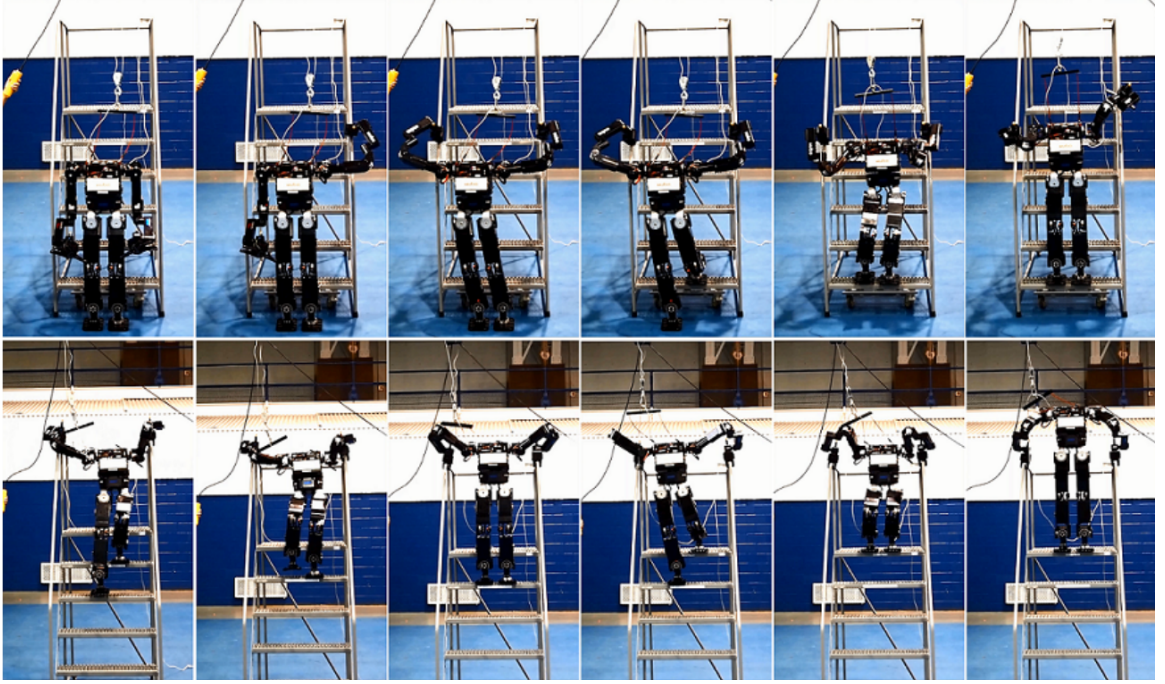


Fig. 6.6.: Snapshots of DRC-Hubo robot climbing a ship-ladder [80,85,86].

Table 6.2: Comparison of the three ladder climbing strategies.

Strategy Comparison	Using only position-control mode	Using only PWM-control mode	Using both modes
Grasping Accuracy	High	Low	High
Damage frequency	High	Low	Low
Overall Success Rate	about 30%	about 80%	>95%

In this set of experiments, the above mentioned strategy was employed. An arm was in the PWM-control mode only when it is in the vicinity of the rails or is grasping the rails. This strategy led to almost 100% success rate in climbing four steps of the ladder. Analysis of instances of failure led to the conclusion that failure, mainly occurred due to improper initialization of the joints. Photographs of the DRC-Hubo



Fig. 6.7.: Snapshots of DRC-Hubo robot climbing the DRC ladder [87].

robot climbing stairs in Armoury of Drexel University, a ship ladder and two different industrial ladders are shown in Fig. 6.5, 6.6, and 6.7, respectively.

6.4 DARPA Robotics Challenge- 2013 Trials

Humans are vulnerable to natural and man-made disasters. In such situations, there is hardly much that humans can do. Robots have the potential to be useful assistants in situations in which humans cannot safely operate. However, the robots of today are not yet robust enough to function in many disaster zones and help mitigate the crisis situation. In order to reduce casualties and save lives during a disaster, it is essential to promote groundbreaking research and development in hardware and software of robots. This will ensure that in future, robots can perform most hazardous activities in disaster zones. Disasters are very unpredictable in their manifestation and effects. Therefore to aid in these situations, robots must be adaptable and require four key capabilities to be effective [78]:

- Mobility and dexterity to maneuver in the degraded environments typical of disaster zones.
- Ability to manipulate and use a diverse assortment of tools designed for humans.
- Ability to be operated by humans who have had little to no robotics training.
- Semi-autonomy in task-level decision-making based on operator commands and sensor inputs.

To ensure that robots should be able to assist humans in disaster and relief operations, DARPA organized the DARPA Robotics Challenge (DRC). To test the mentioned capabilities of a robot, a robot's performance was judged based on the following challenging tasks:

1. Vehicle driving
2. Rough terrain walking
3. Debris clearing
4. Door opening

5. Ladder climbing
6. Wall breaking
7. Hose installation
8. Valve turning

We participated in the DRC 2013 Trials held in Homestead, Florida on 20th and 21st December 2013. ARTLab at Purdue University and Intellignet Motion Laboratory at Indiana University worked together to accomplish the ladder-climbing task. The ladder model for the competition was same as the DRC Ladder used for experiments described in Section 6.3.2 (Table 6.1). The strategy described in Section 6.3.5- to use the position-control mode in the arms to reach the desired position on hand-rails of the ladder, and the PWM-control mode when grasping the ladder- was employed. There was no error in climbing the ladder until the 7th rung. To reach the 8th rung, the robot had to first grasp the hand-rail of the ladder at a higher point. The robot was then expected to place its right foot, followed by the left foot on the 8th rung, and finally straighten its knees. Grasping the hand-rails is essential to ensure that the robot does not fall down. However, while climbing from the 7th rung to the 8th, the robot's right arm was unsuccessful in grasping the handrail of the ladder. Since the remote operator failed to notice that the grasp was unsuccessful, the corrective trajectory was not executed. Though, the robot was able to successfully climb to the 8th rung of the ladder, due to faulty grasp, the robot toppled down when straightening its knees. A photograph of the DRC-Hubo robot climbing the ladder during DRC Trials 2013 is shown in Fig. 6.8.

6.5 Summary

This chapter presented the experiments conducted using hubo-ach and hubo-motion-rt. The results of these experiments were discussed, throwing light on the performance of the packages. The intricacies of the complex task of ladder climbing



Fig. 6.8.: Snapshots of DRC-Hubo robot climbing the industrial ladder during the DARPA Robotics Challenge- Trials 2013 [88].

was described. The motion planning algorithm executed to climb a ladder was discussed. This was followed by the analysis of different strategies employed to execute the generated motion. The next chapter discusses the positives and negatives of the developed software packages. Based on the results of experiments discussed in this chapter, suggestions to improve the hardware and the software are provided.

7. CONCLUSION

7.1 Introduction

Chapter 3, Chapter 4, and Chapter 5 discussed the architecture, usage and applications of hubo-ach and hubo-motion-rt. Chapter 6 discussed various experiments performed using those software packages. This chapter discusses the performance of the developed software packages. We analyze if the developed software packages meet our requirements. The advantages and disadvantages of hubo-ach and hubo-motion-rt are pointed out. The direction to mitigate the disadvantages of hubo-ach and hubo-motion-rt is also discussed. We also summarize the results of the experiments discussed in Chapter 6.

7.2 Analysis of the performance of the software packages

Chapter 3, Chapter 4, and Chapter 5 discussed various software packages developed for controlling a humanoid robot. Hubo-ach was developed to fulfill the requirement of a generic real-time robot control software package. ROS, though very powerful tool, is not real-time. In this section, the developed software packages, particularly hubo-ach and hubo-motion-rt are analyzed. The section discusses if these software packages meet the requirements for controlling a humanoid robot or if they do not.

In order to make the software packages modular, multi-process architecture is used. The processes communicate with each other, or with other software packages, using Ach library. The Ach library has low latency and so is suited for real-time applications. Figures 7.1 and 7.2 provide the architecture of hubo-ach and hubo-motion-rt,

respectively. Understanding the architecture helps in analysing the advantages and disadvantages of the software packages.

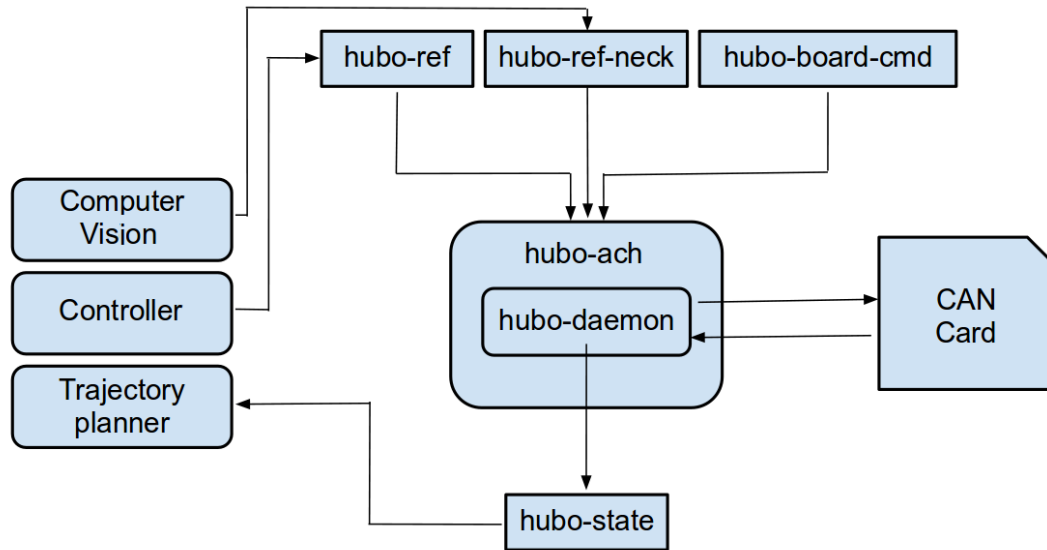


Fig. 7.1.: Architecture of hubo-ach.

1. Real-time performance: The latency when using hubo-ach or hubo-motion-rt for controlling a humanoid robot depends upon the delay in inter-process communication. Both hubo-ach and hubo-motion-rt use Ach library for inter-process communication. Ach library results in communication delay of approximately $20 \mu s$. Thus, the delay to send the desired command to hubo-ach (or hubo-motion-rt) and receiving the latest sensor data from it, is less than 200 Hz, the frequency of execution of hubo-daemon. Therefore, hubo-ach and hubo-motion-rt can both be used for controlling a humanoid robot in real-time.
2. Generic software package: Hubo-ach and hubo-motion-rt are designed to be generic software packages. They can be used for controlling any humanoid robot. The only changes required are:

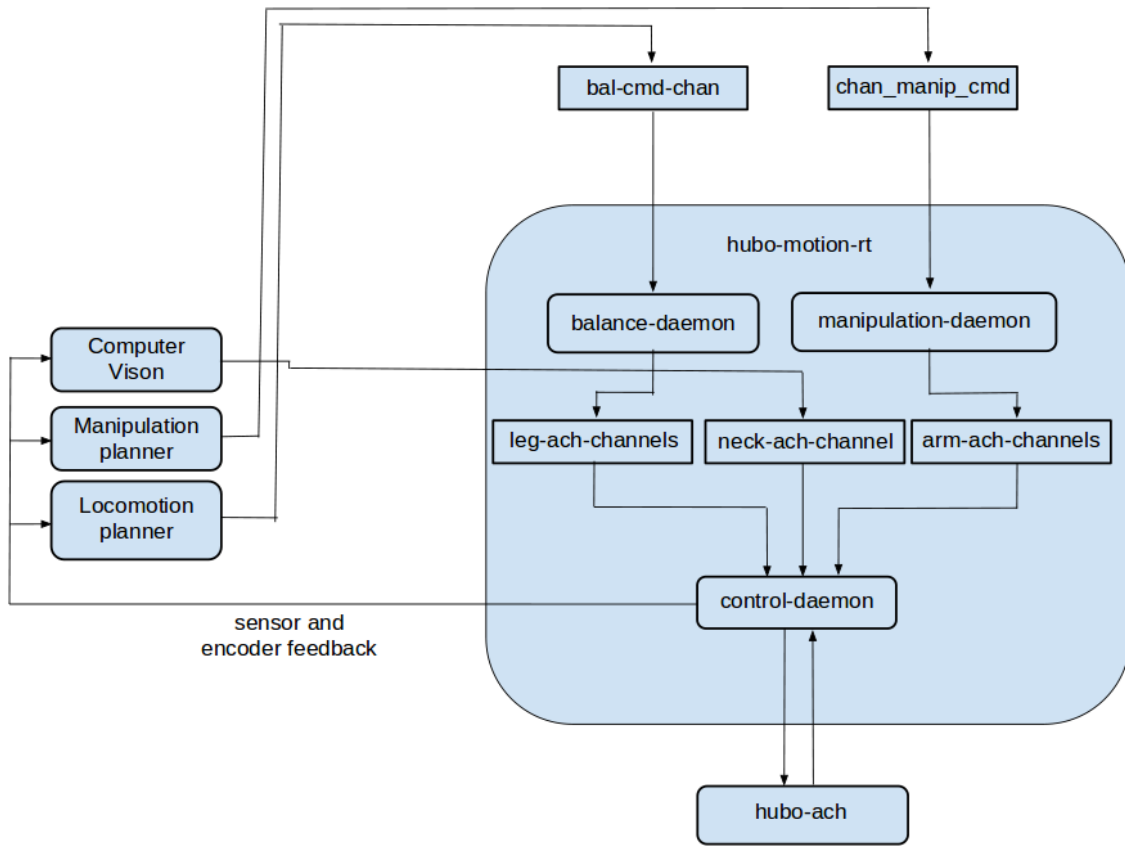


Fig. 7.2.: Architecture of `hubo-motion-rt`.

- The new humanoid robot may not use CAN communication with same heading information. Thus, the module for communication with motor controller boards and sensor should be updated.
- The new humanoid robot may have different sensors than Hubo2+ robot. The robot may also provide different modes of joint control like torque-control. The `hubo_state` and `hubo_ref` data structures need to be updated to incorporate the new features.

The generality of the robot has been tested on various robots manufactured by Rainbow Company. The software has been successfully used to control miniature versions of Hubo2+ robot, KHR-3 robot, and the DRC-Hubo robot [?, 80].

3. **Modularity:** Multi-process architecture directly leads to modularity of the software packages. Many software packages like `hubo-read-trajectory`, `gravity-compensation-based trajectory following`, `hubo-init`, `hubo-neck`, etc. have been developed using `hubo-ach` and `hubo-motion-rt`. It was observed that updating one software package does not affect the others. As long as the data structures are the same, updating `hubo-ach` and `hubo-motion-rt` does not affect other software packages either. Thus, the
4. **Easy integration with other software packages:** It has been demonstrated that `hubo-ach` and `hubo-motion-rt` can be easily integrated with other software packages. `Klamp`, `OpenRAVE`, and `ROS` have been integrated with `hubo-ach`. In a similar fashion, custom controllers can also be easily integrated with `hubo-ach` and `hubo-motion-rt`.
5. **Multi-lingual:** `Hubo-ach` and `hubo-motion-rt` have been developed using C and C++ programming languages. Custom software packages that use `hubo-ach` and `hubo-motion-rt` can also be developed using C or C++. Additionally, `hubo-ach` and `hubo-motion-rt` also provide libraries for development using Python programming language. Thus either of C, C++, or Python languages can also be used developing a custom controller.
6. **Open-source and thin development model:** `Hubo-ach` and `hubo-motion-rt` are both open-source software packages. Many other packages developed using them like `hubo-read-trajectory`, `hubo-init`, etc. are also open-source. Developers all over the world can use these software packages for controlling their humanoid robots. Users are also encouraged to use the thin development model. Users should use existing open-source software packages to develop new controllers. This will not only reduce the development time, but also help in increasing the reliability of the software package.

It is clear that `hubo-ach` and `hubo-motion-rt` fulfill the requirement of generic, real-time, modular, open-source, and multi-lingual software package. They can be

used to control any humanoid robot for real-time application. However, every design approach has its own advantages and disadvantages. Some of the disadvantages of `hubo-ach` and `hubo-motion-rt` are listed below:

1. Number of users: `Hubo-ach` and `hubo-motion-rt`, have been recently developed. The number of users of `hubo-ach` and `hubo-motion-rt` is much smaller than the size of ROS user's community. The number of controllers available within `hubo-ach` or `hubo-motion-rt` is very small. Therefore, it is necessary to develop commonly required controllers compatible with `hubo-ach` and `hubo-motion-rt`. This will also help to increase the number of users of `hubo-ach` and `hubo-motion-rt`.
2. Calibration of Torque-PWM lookup table: Gravity compensation is very sensitive to the motor performance. The PWM-Torque characteristic for different motors varies significantly. Thus, whenever a motor is replaced, the PWM-Torque look-up table needs to be manually re-calibrated for that joint. The characteristic may change over time as well. Thus, an automatic calibration procedure needs to be developed. The calibration could be done by applying various PWM values for a joint and storing all joint angles. Using Newton Euler computation, the torque required at joints to maintain that steady state can be computed and thus the PWM-Torque look-up table can be automatically corrected.
3. Request based data broadcasting: In the current framework, the computer on `hubo` sends a request to the sensor or encoder to publish the information. Upon receiving the request, the sensor board or JMC board transmits the information, which is read by the computer. In the existing solution, to update the `hubo_state`, `hubo-ach` sends a request to all JMC boards and sensor boards. Instead, the firmware on JMC boards can be modified, so that they continuously transmit the information at 200Hz. Thus, the `hubo-computer` can save computation cycles as well as CAN bandwidth.

7.3 Analysis of the experimental work

Chapters 3, 4, and 5 of this thesis discussed the architecture, design, and applications of `hubo-ach`, `hubo-motion-rt`, and other software packages designed using them. Chapter 6 studied the application of `hubo-ach`, `hubo-read-trajectory`, and `hubo-neck` for ladder-climbing task. We successfully demonstrated that DRC-Hubo robot can climb stairs and industrial ladders. A planner based on the idea of motion primitives was used to plan the trajectory of the robot. The features of `hubo-ach` and `hubo-read-trajectory` were used to control the robot and ensure that it follows the desired trajectory. `Hubo-neck` and `hubo-init` were used to monitor the performance of the robot. From the experiment it is clear that the `hubo-ach` is suitable for controlling humanoid robots. Though the motion was quasi-static, the software packages can also be used for executing trajectories which are not quasi-static. `Hubo-ach` has been used to execute motions which are not quasi-static like throwing a baseball, walking, etc. [31]. This experimentally demonstrates that `hubo-ach` is suitable for controlling humanoid robots for real-time applications.

7.4 Future work

We discussed the pros and cons of the existing software packages like ROS and MRDS which are used for controlling robots. To overcome the shortcomings of the existing software, we proposed developing a real-time, open-source, and multi-lingual software package. However, there is always a scope for improvement. We saw in Section 7.3, the developed software comes with a few disadvantages. The future work should be directed towards mitigating these drawbacks.

1. Proving the generality of the `hubo-ach` and `hubo-motion-rt`: The packages are developed to be generic. The architecture as well as implementation of the software packages is not specific to any robot. However to be more confident of their generality, it is necessary to demonstrate and validate that they can be used for controlling other humanoid robots like Darwin, NAO, etc.

2. Integrating existing humanoid robot controllers: To control humanoid robots like NAO and Atlas, ROS packages are available. However, since ROS is not real-time software, the motion of these robots is quasi-static. Nevertheless, the developed packages, if generic, can be used for other humanoid robots as well. Therefore it is necessary to integrate existing humanoid robot planning and control software packages with hubo-ach and hubo-motion-rt. The integration will greatly enhance the capabilities of hubo-ach and hubo-motion-rt.
3. Application of hubo-ach and hubo-motion-rt for different tasks: It is expected that in the near future, humanoid robots will be used for wide range of applications like: search-and-rescue operations, cooperating with human beings, space exploration, etc. To work towards this goal, it is necessary to develop new algorithms for motion-control, motion-planning, computer vision, etc. The designed algorithms should be integrated with hubo-ach and hubo-motion-rt. The integration with hubo-ach and hubo-motion-rt will smoothen and quicken the research. Constant effort in this direction will ensure that robots assist humans in daily lives.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] W. Heginbotham, "Assembly automation-past, present and future," *Assembly Automation*, vol. 1, no. 1, pp. 14–20, 1980.
- [2] Sabre-Autonomous-Solutions, "Industrial robotic systems." <http://sabreautonomous.com.au/about-us/our-technology/>.
- [3] Xinhua-agencies, "Robot put into use in lushan quake rescue." <http://www.globaltimes.cn/DesktopModules/DnnForge%20-%20NewsArticles/Print.aspx?tabid=99&tabmoduleid=94&articleId=777763&moduleId=405&PortalID=0>, 2013.
- [4] N. U. Ali, "Doctor versus algorithms." <http://in-training.org/doctor-versus-algorithm-which-would-you-trust-4298>, 2014.
- [5] Nvidia, "3d vision helps nasa explore mars." <http://www.3dvisionlive.com/content/3d-vision-helps-nasa-explore-mars>, 2012.
- [6] D. Cassella, "Robovie-ii assists the elderly in supermarkets." <http://www.digitaltrends.com/cool-tech/robovie-ii-assists-the-elderly-in-supermarkets/#!Ag9UQ>, 2010.
- [7] DARPA, "Darpa robotics challenge (drc)." http://www.darpa.mil/Our_Work/TT0/Programs/DARPA_Robotics_Challenge.aspx, 2012.
- [8] CareObotII, "Features of care-o-bot ii." <http://www.care-o-bot.de/en/care-o-bot-3/history/care-o-bot-ii.html>, 2002.
- [9] Cargnews-Asia, "Amazon hopes idea of drone delivery will fly." <http://www.cargonewsasia.com/secured/article.aspx?article=32352>, 2013.
- [10] J.-I. Yamaguchi, A. Takanishi, and I. Kato, "Development of a biped walking robot compensating for three-axis moment by trunk motion," in *Intelligent Robots and Systems' 93, IROS'93. Proceedings of the 1993 IEEE/RSJ International Conference on*, vol. 1, pp. 561–566, IEEE, 1993.
- [11] J.-I. Yamaguchi, A. Takanishi, and I. Kato, "Development of a biped walking robot adapting to a horizontally uneven surface," in *Intelligent Robots and Systems' 94. Advanced Robotic Systems and the Real World, IROS'94. Proceedings of the IEEE/RSJ/GI International Conference on*, vol. 2, pp. 1156–1163, IEEE, 1994.
- [12] I.-W. Park, J.-Y. Kim, J. Lee, and J.-H. Oh, "Mechanical design of the humanoid robot platform, hubo," *Advanced Robotics*, vol. 21, no. 11, pp. 1305–1322, 2007.

- [13] I.-W. Park, J.-Y. Kim, J. Lee, and J.-H. Oh, "Mechanical design of humanoid robot platform khr-3 (kaist humanoid robot 3: Hubo)," in *Humanoid Robots, 2005 5th IEEE-RAS International Conference on*, pp. 321–326, IEEE, 2005.
- [14] H. Surmann, A. Nüchter, and J. Hertzberg, "An autonomous mobile robot with a 3d laser range finder for 3d exploration and digitalization of indoor environments," *Robotics and Autonomous Systems*, vol. 45, no. 3, pp. 181–198, 2003.
- [15] J. W. Weingarten, G. Gruener, and R. Siegwart, "A state-of-the-art 3d sensor for robot navigation," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 3, pp. 2155–2160, IEEE, 2004.
- [16] K. Kosuge and Y. Hirata, "Human-robot interaction," in *Robotics and Biomimetics, 2004. ROBIO 2004. IEEE International Conference on*, pp. 8–11, IEEE, 2004.
- [17] J. Lee, J.-Y. Kim, I.-W. Park, B.-K. Cho, M.-S. Kim, I. Kim, and J.-H. Oh, "Development of a humanoid robot platform hubo fx-1," in *SICE-ICASE, 2006. International Joint Conference*, pp. 1190–1194, IEEE, 2006.
- [18] S. Rodríguez-Jiménez and M. Abderrahim, "3-dimensional object perception for manipulation tasks using the atlas robot," in *ROBOT2013: First Iberian Robotics Conference*, pp. 359–368, Springer, 2014.
- [19] N. Levi, G. Kovelman, A. Geynis, A. Sintov, and A. Shapiro, "The darpa virtual robotics challenge experience," in *Safety, Security, and Rescue Robotics (SSRR), 2013 IEEE International Symposium on*, pp. 1–6, IEEE, 2013.
- [20] E. Garcia, M. Ocaña, L. M. Bergasa, M. Ferre, M. Abderrahim, J. C. Arevalo, D. Sanz-Merodio, E. J. Molinos, N. Hernandez, Á. Llamazares, *et al.*, "Competing in the darpa virtual robotics challenge as the sarbot team," in *ROBOT2013: First Iberian Robotics Conference*, pp. 381–396, Springer, 2014.
- [21] P. Csonka and K. Waldron, "A brief history of legged robotics," in *Technology Developments: the Role of Mechanism and Machine Science and IFToMM*, pp. 59–73, Springer, 2011.
- [22] M. Hirose and K. Ogawa, "Honda humanoid robots development," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 365, no. 1850, pp. 11–19, 2007.
- [23] Y. Sakagami, R. Watanabe, C. Aoyama, S. Matsunaga, N. Higaki, and K. Fujimura, "The intelligent asimo: System overview and integration," in *Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on*, vol. 3, pp. 2478–2483, IEEE, 2002.
- [24] M. Diftler and R. O. Ambrose, "Robonaut: a robotic astronaut assistant," in *ISAIRAS conference*, vol. 2001, 2001.
- [25] M. A. Diftler, J. Mehling, M. E. Abdallah, N. A. Radford, L. B. Bridgwater, A. M. Sanders, R. S. Askew, D. M. Linn, J. D. Yamokoski, F. Permenter, *et al.*, "Robonaut 2-the first humanoid robot in space," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 2178–2183, IEEE, 2011.

- [26] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier, "Mechatronic design of nao humanoid," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, pp. 769–774, IEEE, 2009.
- [27] D. Gouaillier, V. Hugel, P. Blazevic, C. Kilner, J. Monceaux, P. Lafourcade, B. Marnier, J. Serre, and B. Maisonnier, "The nao humanoid: a combination of performance and affordability," *CoRR abs/0807.3223*, 2008.
- [28] K. Kaneko, "Towards emergency response humanoid robots," in *Mechatronics (MECATRONICS), 2012 9th France-Japan & 7th Europe-Asia Congress on and Research and Education in Mechatronics (REM), 2012 13th Int'l Workshop on*, pp. 504–511, IEEE, 2012.
- [29] K. Bouyarmane, J. Vaillant, F. Keith, and A. Kheddar, "Exploring humanoid robots locomotion capabilities in virtual disaster response scenarios," in *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, pp. 337–342, IEEE, 2012.
- [30] H. Dang, Y. Jun, P. Oh, and P. K. Allen, "Planning complex physical tasks for disaster response with a humanoid robot," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013.
- [31] D. M. Lofaro, *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots*. PhD thesis, Drexel University, 2013.
- [32] S. Behnke, M. Schreiber, J. Stuckler, R. Renner, and H. Strasdat, "See, walk, and kick: Humanoid robots start to play soccer," in *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, pp. 497–503, IEEE, 2006.
- [33] J. Mller and T. Rfer, "Kicking a ball modeling complex dynamic motions for humanoid robots, in robocup 2010: Robot soccer world cup xiv, ser," in *Lecture Notes in Artificial Intelligence*.
- [34] K. Nishiwaki, A. Ionno, K. Nagashima, M. Inaba, and H. Inoue, "The humanoid saika that catches a thrown ball," in *Robot and Human Communication, 1997. RO-MAN'97. Proceedings., 6th IEEE International Workshop on*, pp. 94–99, IEEE, 1997.
- [35] B. Bauml, F. Schmidt, T. Wimbock, O. Birbach, A. Dietrich, M. Fuchs, W. Friedl, U. Frese, C. Borst, M. Grebenstein, *et al.*, "Catching flying balls and preparing coffee: Humanoid rollin'justin performs dynamic and sensitive tasks," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 3443–3444, IEEE, 2011.
- [36] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mosenlechner, D. Pangeric, T. Ruhr, and M. Tenorth, "Robotic roommates making pancakes," in *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pp. 529–536, IEEE, 2011.
- [37] B. V. Adorno, A. P. L. Bó, P. Fraisse, and P. Poignet, "Towards a cooperative framework for interactive manipulation involving a human and a humanoid," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pp. 3777–3783, IEEE, 2011.

- [38] A. St Clair and M. J. Mataric, "Task coordination and assistive opportunity detection via social interaction in collaborative human-robot tasks," in *Collaboration Technologies and Systems (CTS), 2011 International Conference on*, pp. 168–172, IEEE, 2011.
- [39] S. G. McGill and D. D. Lee, "Cooperative humanoid stretcher manipulation and locomotion," in *Humanoid Robots (Humanoids), 2011 11th IEEE-RAS International Conference on*, pp. 429–433, IEEE, 2011.
- [40] C. Breazeal, "Socially intelligent robots," *interactions*, vol. 12, no. 2, pp. 19–22, 2005.
- [41] K. Dautenhahn, "Socially intelligent robots: dimensions of human-robot interaction," *Philosophical Transactions of the Royal Society B: Biological Sciences*, vol. 362, no. 1480, pp. 679–704, 2007.
- [42] R. Alazrai and C. G. Lee, "Real-time emotion identification for socially intelligent robots," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pp. 4106–4111, IEEE, 2012.
- [43] S. Behnke, "Humanoid robots-from fiction to reality?," *KI*, vol. 22, no. 4, pp. 5–9, 2008.
- [44] R. O. Ambrose, R. R. Burrige, W. Bluethmann, H. Aldridge, R. S. Askew, F. Rehnmark, M. Diftler, D. Magruder, and C. Lovchik, "Robonaut: Nasa's space humanoid," *IEEE Intelligent Systems*, vol. 15, no. 4, pp. 57–63, 2000.
- [45] T. Guardian, "Robots become reality." <http://www.theguardian.com/technology/gallery/2009/dec/02/robots-japan>, 2009.
- [46] E. M. Daily, "Japanese robo-chefs to feed us all." <http://www.eatmedaily.com/2009/06/japanese-robo-chefs-to-feed-replace-us-all-video/>, 2009.
- [47] Rediff, "Future of robots: Rj can catch balls, make coffee." <http://www.rediff.com/money/slide-show/slide-show-1-tech-a-robot-that-can-play-catch-and-make-coffee/20110608.htm>, 2011.
- [48] A. Liesowska, "Siberia to host cyber football world cup in 2018 - for robots." <http://siberiantimes.com/sport/football/news/siberia-to-host-cyber-football-world-cup-in-2018-for-robots/>, 2013.
- [49] B. Durán and S. Thill, "Robs robot: Current and future challenges for humanoid robots," in *The Future of Humanoid Robots-Research and Applications, InTech*, 2012.
- [50] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, 2009.
- [51] A. Makarenko, A. Brooks, and T. Kaupp, "On the benefits of making robotic software frameworks thin," in *International Conference on Intelligent Robots and Systems*, vol. 2, 2007.

- [52] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics & Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007.
- [53] R. Diankov, *Automated Construction of Robotic Manipulation Programs*. PhD thesis, Carnegie Mellon University, Robotics Institute, August 2010.
- [54] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes,"
- [55] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 2, pp. 2766–2771, IEEE, 2003.
- [56] H. Bruyninckx, "Open robot control software: the orocos project," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 3, pp. 2523–2528, IEEE, 2001.
- [57] M. Grey, N. T. Dantam, D. M. Lofaro, P. Oh, A. Bobick, M. Egerstedt, and M. Stilman, "Multi-process control software for humanoid robots," in *IEEE International Conference on Technologies for Practical Robot Applications*, pp. 190–195, 2013.
- [58] H. Yoneda, K. Sekiyama, Y. Hasegawa, and T. Fukuda, "Vertical ladder climbing motion with posture control for multi-locomotion robot," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pp. 3579–3584, IEEE, 2008.
- [59] K. Harada, K. Hauser, T. Bretl, and J.-C. Latombe, "Natural motion generation for humanoid robots," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pp. 833–839, IEEE, 2006.
- [60] K. Hauser, T. Bretl, and J.-C. Latombe, "Non-gaited humanoid locomotion planning," in *Humanoid Robots, 2005 5th IEEE-RAS International Conference on*, pp. 7–12, IEEE, 2005.
- [61] M. Wang, Y. Yang, R. R. Hatch, and Y. Zhang, "Adaptive filter for a miniature mems based attitude and heading reference system," in *Position Location and Navigation Symposium, 2004. PLANS 2004*, pp. 193–200, IEEE, 2004.
- [62] K. S. Fu, R. Gonzalez, and C. G. Lee, *Robotics: Control Sensing Vision and Intelligence*. Tata McGraw-Hill Education, 1987.
- [63] D. L. Peiper, "The kinematics of manipulators under computer control," tech. rep., DTIC Document, 1968.
- [64] H. A. Park, M. A. Ali, and C. G. Lee, "Closed-form inverse kinematic position solution for humanoid robots," *International Journal of Humanoid Robotics*, vol. 9, no. 03, 2012.
- [65] M. A. Ali, H. A. Park, and C. G. Lee, "Closed-form inverse kinematic joint solution for humanoid robots," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pp. 704–709, IEEE, 2010.
- [66] B.-K. Cho, S.-S. Park, and J.-h. Oh, "Controllers for running in the humanoid robot, hubo," in *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, pp. 385–390, IEEE, 2009.

- [67] A. Watanabe and S. Yuta, "Efficient feedforward current control method of brushless dc motor: By using non-complementary switching in driver circuit," in *Advanced Motion Control, 2010 11th IEEE International Workshop on*, pp. 768–773, IEEE, 2010.
- [68] A. R. Golding and N. Lesh, "Indoor navigation using a diverse set of cheap, wearable sensors," in *Wearable Computers, 1999. Digest of Papers. To the Third International Symposium on*, pp. 29–36, IEEE, 1999.
- [69] J. N. Pires, J. Ramming, S. Rauch, and R. Araújo, "Force/torque sensing applied to industrial robotic deburring," *Sensor Review*, vol. 22, no. 3, pp. 232–241, 2002.
- [70] M. Farsi, K. Ratcliff, and M. Barbosa, "An overview of controller area network," *Computing & Control Engineering Journal*, vol. 10, no. 3, pp. 113–120, 1999.
- [71] N. Dantam and M. Stilman, "Robust and efficient communication for real-time multi-process robot software," in *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, pp. 316–322, IEEE, 2012.
- [72] C. E. Agüero, J. M. Canas, F. Martín, and E. Perdices, "Behavior-based iterative component architecture for soccer applications with the nao humanoid," in *5th Workshop on Humanoids Soccer Robots. Nashville, TN, USA*, 2010.
- [73] E. Yoshida, M. Poirier, J.-P. Laumond, O. Kanoun, F. Lamiriaux, R. Alami, and K. Yokoi, "Whole-body motion planning for pivoting based manipulation by humanoids," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 3181–3186, IEEE, 2008.
- [74] J. Kuffner, K. Nishiwaki, S. Kagami, M. Inaba, and H. Inoue, "Motion planning for humanoid robots," in *Robotics Research*, pp. 365–374, Springer, 2005.
- [75] R. O’Flaherty, P. Vieira, M. Grey, P. Oh, A. Bobick, M. Egerstedt, and M. Stilman, "Humanoid robot teleoperation for tasks with power tools," in *IEEE International Conference on Technologies for Practical Robot Applications*, pp. 119–124, 2013.
- [76] M. Stilman, K. Nishiwaki, and S. Kagami, "Humanoid teleoperation for whole body manipulation," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pp. 3175–3180, IEEE, 2008.
- [77] N. Miller, O. C. Jenkins, M. Kallmann, and M. J. Mataric, "Motion capture from inertial sensing for untethered humanoid teleoperation," in *Humanoid Robots, 2004 4th IEEE/RAS International Conference on*, vol. 2, pp. 547–565, IEEE, 2004.
- [78] DARPA, "Darpa robotics challenge." <http://www.theroboticschallenge.org/>, 2013.
- [79] K. Hauser, T. Bretl, J.-C. Latombe, K. Harada, and B. Wilcox, "Motion planning for legged robots on varied terrain," *The International Journal of Robotics Research*, vol. 27, no. 11-12, pp. 1325–1349, 2008.

- [80] J. Luo, Y. Zhang, K. Hauser, A. Park, M. Paldhe, C. S. G. Lee, M. Grey, M. Stilman, J. H. Oh, J. Lee, I. Kim, and Y. O. Paul, "Robust ladder-climbing with a humanoid robot with application to the darpa robotics challenge," in *Accepted at Robotics and Automation (ICRA), 2014 IEEE International Conference on*, IEEE, 2014.
- [81] Y. Zhang, J. Luo, K. Hauser, A. Park, M. Paldhe, C. S. G. Lee, R. Ellenberg, B. Killen, P. Oh, J. H. Oh, J. Lee, and I. Kim, "Motion planning and control of ladder climbing on drc-hubo for darpa robotics challenge" (video contribution)," in *Accepted at Robotics and Automation (ICRA), 2014 IEEE International Conference on*, IEEE, 2014.
- [82] Y. Zhang, J. Luo, K. Hauser, R. Ellenberg, P. Oh, H. A. Park, and M. Paldhe, "Motion planning of ladder climbing for humanoid robots," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013.
- [83] K. Hauser, Y. Zhang, and J. Luo, "Planner-aided design of ladder climbing capabilities for a darpa robotics challenge humanoid,"
- [84] Y. Zhang, "Drc-hubo climbing stairs in armoury, drexel university, philadelphia." http://www.youtube.com/watch?v=1L2bjh2TS_Q, 2013.
- [85] Y. Zhang, "Drc-hubo ship ladder during testing." <http://www.youtube.com/watch?v=ZEUVQRrKfjo>, 2013.
- [86] D. Lofaro, "Drc-hubo climbing ladder during the team's dry run." <http://www.youtube.com/watch?v=2ft83Cs7k6M>, 2013.
- [87] D. Lofaro, "Drc-hubo climbing ladder during final practice test." <https://www.youtube.com/watch?v=VALLGcg0zg>, 2013.
- [88] M. Paldhe, "Drc-hubo climbing ladder at the drc trials 2013." <https://www.youtube.com/watch?v=wjt97BPREDc>, 2013.
- [89] M. Zucker, Y. Jun, B. Killen, T.-G. Kim, and P. Oh, "Continuous trajectory optimization for autonomous humanoid door opening," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–5, IEEE, 2013.
- [90] Y. Zheng, H. Wang, S. Li, Y. Liu, D. Orin, K. Sohn, Y. Jun, and P. Oh, "Humanoid robots walking on grass, sands and rocks," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013.
- [91] N. Alunni, C. Phillips-Grafftin, H. B. Suay, D. Lofaro, D. Berenson, S. Chernova, R. W. Lindeman, and P. Oh, "Toward a user-guided manipulation framework for high-dof robots with limited communication," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013.
- [92] C. Rasmussen, K. Yuvraj, R. Vallett, K. Sohn, and P. Oh, "Towards functional labeling of utility vehicle point clouds for humanoid driving," in *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, pp. 1–6, IEEE, 2013.

APPENDICES

A. D-H REPRESENTAION AND TRANSFORMATION MATRICES

Appendix-A will discuss the D-H representation of the Hubo2+ robot and inverse kinematic solutions for the limbs. Though, for simplicity of usage instead of using the D-H representation, all the axes are aligned in same direction, understanding the D-H representation and the inverse kinematic solution is useful.

A.1 D-H Representation and forward kinematics

In D-H representation, position of a link of the robot is described with respect to another link. In case of a robotic arm, the base of the robot is stationary and thus is the constant in global coordinates. However a humanoid robot moves around. Thus, for a humanoid robot a coordinate base is set at the neck joint. This is considered as the global frame for the robot. All the limb bases are related to this base through the waist joint. Thus, points in limb base coordinate system can be easily transformed to the global base by a transformation matrix.

In D-H representation relation between link i 's coordinate system to link $(i - 1)$'s coordinate system for rotary joint i is given by the following matrix:

$${}^{i-1}\mathbf{A}_i = \begin{bmatrix} \cos \theta_i & -\cos \alpha_i \sin \theta_i & \sin \alpha_i \sin \theta_i & a_i \cos \theta_i \\ \sin \theta_i & \cos \alpha_i \cos \theta_i & -\sin \alpha_i \cos \theta_i & a_i \sin \theta_i \\ 0 & \sin \alpha_i & \cos \alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The joint parameters α_i , a_i and d_i are provided in Table A.1. From the joint parameters, the transformation matrices for each joint on right arm of the Hubo2+ robot are obtained.

Table A.1: D-H Parameter of Hubo2+ robot's right arm. These parameters are used to compute forward and inverse kinematic solutions.

Joint i	θ_i	α_i	\mathbf{a}_i	\mathbf{d}_i
1	θ_1	90°	0	0
2	θ_2	-90°	0	0
3	θ_3	90°	0	d_3
4	θ_4	90°	0	0
5	θ_5	90°	0	d_5
6	θ_6	0°	a_6	0

$${}^0\mathbf{A}_1 = \begin{bmatrix} C_1 & 0 & S_1 & 0 \\ S_1 & 0 & -C_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^1\mathbf{A}_2 = \begin{bmatrix} C_2 & 0 & -S_2 & 0 \\ S_2 & 0 & C_2 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.1})$$

$${}^2\mathbf{A}_3 = \begin{bmatrix} C_3 & 0 & S_3 & 0 \\ S_3 & 0 & -C_3 & 0 \\ 0 & 1 & 0 & d_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}^3 \quad \mathbf{A}_4 = \begin{bmatrix} C_4 & 0 & S_4 & 0 \\ S_4 & 0 & -C_4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.2})$$

$${}^4\mathbf{A}_5 = \begin{bmatrix} C_5 & 0 & S_5 & 0 \\ S_5 & 0 & -C_5 & 0 \\ 0 & 1 & 0 & d_5 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^5\mathbf{A}_6 = \begin{bmatrix} C_6 & -S_6 & 0 & a_6 C_6 \\ S_6 & C_6 & 0 & a_6 S_6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.3})$$

Similarly, the transformation matrices for the right leg can be computed from table A.2.

Table A.2: D-H Parameter of Hubo2+ robot's right leg. These parameters are used to compute forward and inverse kinematic solutions.

Joint i	θ_i	α_i	\mathbf{a}_i	\mathbf{d}_i
1	θ_1	90°	0	0
2	θ_2	-90°	0	0
3	θ_3	0°	\mathbf{a}_3	0
4	θ_4	0°	\mathbf{a}_4	0
5	θ_5	90°	0	0
6	θ_6	0°	\mathbf{a}_6	0

$${}^0\mathbf{A}_1 = \begin{bmatrix} C_1 & 0 & S_1 & 0 \\ S_1 & 0 & -C_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^1\mathbf{A}_2 = \begin{bmatrix} C_2 & 0 & -S_2 & 0 \\ S_2 & 0 & C_2 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.4})$$

$${}^2\mathbf{A}_3 = \begin{bmatrix} C_3 & -S_3 & 0 & a_3 C_3 \\ S_3 & C_3 & 0 & a_3 S_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}^3 \quad \mathbf{A}_4 = \begin{bmatrix} C_4 & -S_4 & 0 & a_4 C_4 \\ S_4 & C_4 & 0 & a_4 S_4 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.5})$$

$${}^4\mathbf{A}_5 = \begin{bmatrix} C_5 & 0 & S_5 & 0 \\ S_5 & 0 & -C_5 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad {}^5\mathbf{A}_6 = \begin{bmatrix} C_6 & -S_6 & 0 & a_6 C_6 \\ S_6 & C_6 & 0 & a_6 S_6 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.6})$$

By multiplying all the six transformation matrices in order, the position and orientation of the limb end point (fingers in case of hand and foot location in case of legs) can be easily obtained.

$$\mathbf{T} = {}^0\mathbf{A}_1\mathbf{A}_2\mathbf{A}_3\mathbf{A}_4\mathbf{A}_5\mathbf{A}_6 = \begin{bmatrix} \mathbf{n} & \mathbf{s} & \mathbf{a} & \mathbf{p} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} n_x & a_x & p_x & p_x \\ n_y & a_y & p_y & p_y \\ n_z & a_z & p_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (\text{A.7})$$

where,

$$\begin{aligned} n_x &= -C_6(S_1S_{345} - C_1C_2C_{345}) - C_1S_2S_6 \\ n_y &= C_6(C_1S_{345} + C_2C_{345}S_1) - S_1S_2S_6 \\ n_z &= C_2S_6 + C_{345}C_6S_2 \\ s_x &= S_6(S_1S_{345} - C_1C_2C_{345}) - C_1C_6S_2 \\ s_y &= -S_6(C_1S_{345} + C_2C_{345}S_1) - C_6S_1S_2 \\ s_z &= C_2C_6 - C_{345}S_2S_6 \\ a_x &= C_{345}S_1 + C_1C_2S_{345} \\ a_y &= C_2S_1S_{345} - C_1C_{345} \\ a_z &= S_2S_{345} \\ p_x &= l_{L3}C_1C_2C_3 - (C_3S_1 + C_1C_2S_3)(l_{L4}S_4 + l_{L5}C_6S_{45}) \\ &\quad - (S_1S_3 - C_1C_2C_3)(l_{L4}C_4 + l_{L5}C_6C_{45} - l_3S_1S_3 - l_{L5}C_1S_2S_6) \\ p_y &= (C_1S_3 + C_2C_3S_1)(l_{L4}C_4 + l_{L5}C_6C_{45}) + (C_1C_3 - C_2S_1S_3)(l_{L4}S_4 + l_{L5}C_6S_{45}) + l_{L3}C_1S_3 \\ &\quad + l_{L3}C_2C_3S_1 - l_{L5}S_1S_2S_6 \\ p_z &= l_{L3}C_3S_2 + l_{L5}C_2S_6 + C_3S_2(l_{L4}C_4 + l_{L5}C_6C_{45}) - S_2S_3(l_{L4}S_4 + l_{L5}C_6S_{45}) \end{aligned}$$

In this notation:

S_i denotes $\sin(\theta_i)$, C_i denotes $\cos(\theta_i)$,

S_{ij} denotes $\sin(\theta_i + \theta_j)$, C_{ij} denotes $\cos(\theta_i + \theta_j)$,

S_{ijk} denotes $\sin(\theta_i + \theta_j + \theta_k)$, and C_{ijk} denotes $\cos(\theta_i + \theta_j + \theta_k)$

A.2 Inverse kinematics

Right Leg

In case of Hubo2+ robot's legs the three hip joints (hip roll, pitch and yaw joints) intersect at a point. Thus, a closed form inverse kinematic (IK) solution can be obtained. To obtain the IK solution, the base is moved to the foot, instead of the hip joint. Then the joint solutions are obtained by solving the new IK problem. On changing the base, new \mathbf{n} , \mathbf{s} , \mathbf{a} and \mathbf{p} vectors are obtained by computing the T^{-1} as:

$$\begin{aligned}
 n'_x &= -C_1(S_2S_6 - C_2C_6C_{345}) - S_1C_6S_{345} \\
 n'_y &= S_1S_6S_{345} - C_1(C_2S_6C_{345} + C_6S_2) \\
 n'_z &= S_1C_{345} + C_1C_2S_{345} \\
 s'_x &= C_1C_6S_{345} - S_1(S_2S_6 - C_2C_6C_{345}) \\
 s'_y &= -C_1S_6S_{345} - S_1(C_2S_6C_{345} + C_6S_2) \\
 s'_z &= C_2S_1S_{345} - C_1C_{345} \\
 a'_x &= C_2S_6 + S_2C_6C_{345} \\
 a'_y &= C_2C_6 - S_2S_6C_{345} \\
 a'_z &= S_2S_{345} \\
 p'_x &= -l_{L5} - l_{L3}C_6C_{45} - C_5C_6l_{L4} \\
 p'_y &= l_{L3}S_6C_{45} + l_{L4}C_5S_6 \\
 p'_z &= -l_{L4}S_5 - l_{L3}S_{45}
 \end{aligned}$$

The closed form solution is:

$$\begin{aligned}
 C_4 &= \frac{(p'_x + l_{L5})^2 + p'^2_y + p'^2_z - l_{L3}^2 - l_{L4}^2}{2l_{L3}l_{L4}}, & S_4 &= \text{KNEE} \cdot \sqrt{1 - C_4^2} \\
 \theta_4 &= \text{atan2}(S_4, C_4) \\
 \psi &= \text{atan2}(S_4l_{L3}, C_4l_{L3} + l_{L4}) \\
 \theta_5 &= \text{atan2}(-p'_z, \text{ANKLE} \cdot \sqrt{(p'_x + l_{L5})^2 + p'^2_y}) - \psi \\
 \theta_6 &= \begin{cases} \text{atan2}(p'_y, -p'_x - l_{L5}) + \pi & \text{if } (C_{45}l_{L3} + C_5l_{L4}) < 0 \\ \text{atan2}(p'_y, -p'_x - l_{L5}) & \text{otherwise} \end{cases}
 \end{aligned}$$

$$\begin{aligned}
C_2 &= S_6 s'_x + C_6 s'_y & S_2 &= \text{HIP} \cdot \sqrt{1 - C_2^2} \\
\theta_2 &= \text{atan2}(S_2, C_2) \\
\theta_1 &= \text{atan2}(-S_6 s'_x - C_6 s'_y, -S_6 n'_x - C_6 n'_y) \\
\theta_{345} &= \text{atan2}(a'_z, C_6 a'_x - S_6 a'_y) \\
\theta_3 &= \theta_{345} - \theta_4 - \theta_5
\end{aligned}$$

Left Leg

But for a small change, the solution for the right leg also applies to left leg. To obtain the IK solution for left leg, replace l_{L1} with $-l_{L1}$.

Right Arm

Similar to the IK computation of right leg, the IK computation of right arm can be easily done. The vectors \mathbf{n}' , \mathbf{s}' , \mathbf{a}' and \mathbf{p}' are:

$$\begin{aligned}
n'_x &= C_1(C_2(C_3(S_5S_6 + C_4C_5C_6) + C_6S_3S_5) + S_2(C_4S_6 - C_5C_6S_5)) \\
&\quad - S_1(S_3(S_5S_6 + C_4C_5C_6) - C_3C_6S_5) \\
n'_y &= C_1(C_2(C_3(C_6S_5 - C_4C_5S_6) - S_3S_5S_6) + S_2(C_4C_6 + C_5S_5S_6)) \\
&\quad - S_1(S_3(C_6S_5 - C_4C_5S_6) + C_3S_5S_6) \\
n'_z &= -C_1(C_2(C_5S_3 - C_3C_4S_5) + S_2S_5S_5) - S_1(C_3C_5 + C_4S_3S_5) \\
s'_x &= C_1(S_3(S_5S_6 + C_4C_5C_6) - C_3C_6S_5) + S_1(C_2(C_3(S_5S_6 + C_4C_5C_6) \\
&\quad + C_6S_3S_5) + S_2(C_4S_6 - C_5C_6S_5)) \\
s'_y &= C_1(S_3(C_6S_5 - C_4C_5S_6) + C_3S_5S_6) + S_1(C_2(C_3(C_6S_5 - C_4C_5S_6) \\
&\quad - S_3S_5S_6) + S_2(C_4C_6 + C_5S_5S_6)) \\
s'_z &= C_1(C_3C_5 + C_4S_3S_5) - S_1(C_2(C_5S_3 - C_3C_4S_5) + S_2S_5S_5) \\
a'_x &= S_2(C_3(S_5S_6 + C_4C_5C_6) + C_6S_3S_5) - C_2(C_4S_6 - C_5C_6S_5) \\
a'_y &= S_2(C_3(C_6S_5 - C_4C_5S_6) - S_3S_5S_6) - C_2(C_4C_6 + C_5S_5S_6) \\
a'_z &= C_2S_5S_5 - C_5S_2S_3 + C_3C_4S_2S_5 \\
p'_x &= -l_4 - l_2(C_4S_6 - C_5C_6S_5) - l_3S_6 \\
p'_y &= -l_3C_6 - l_2C_4C_6 - l_2C_5S_5S_6 \\
p'_z &= l_2S_5S_5
\end{aligned}$$

Using these computations, the IK solution is computed as:

$$\begin{aligned}
C_4 &= \frac{(p'_x + l_{A4})^2 + p'^2_y + p'^2_z - l_{A2}^2 - l_{A3}^2}{2l_{A2}l_{A3}}, & S_4 &= \text{ELBOW} \cdot \sqrt{1 - C_4^2} \\
\theta_4 &= \text{atan2}(S_4, C_4) \\
C_5 &= \text{WRIST} \cdot \sqrt{1 - S_5^2}, & S_5 &= p'_z / (S_4 l_{A2}) \\
\theta_5 &= \text{atan2}(S_5, C_5) \\
\psi &= \text{atan2}(p'_y, p'_x + l_{A4}) \\
\theta_6 &= \text{atan2}(-(C_4 l_{A2} + l_{A3}), S_4 C_5 l_{A2}) - \psi \\
C_2 &= a'_z S_4 S_5 - a'_y (C_4 C_6 + S_4 C_5 S_6) - a'_x (C_4 S_6 - S_4 C_5 C_6), \\
S_2 &= \text{SHOULDER} \cdot \sqrt{1 - C_2^2} \\
\theta_2 &= \text{atan}(S_2, C_2) \\
g_{413} &= a'_x (C_4 C_5 C_6 + S_4 S_6) + C_4 S_5 a'_z + (S_4 C_6 - C_4 C_5 S_6) a'_y \\
g_{433} &= S_5 C_6 a'_x - C_5 a'_z - S_5 S_6 a'_y \\
\theta_3 &= \text{atan}(g_{433}, g_{413}) \\
g_{421} &= (S_4 C_5 C_6 - C_4 S_6) n'_x + S_4 S_5 n'_z - (S_4 C_5 S_6 + C_4 C_6) n'_y \\
g_{422} &= (S_4 C_5 C_6 - C_4 S_6) s'_x + S_4 S_5 s'_z - S_4 C_5 S_6 + C_4 C_6 s'_y \\
\theta_1 &= \\
\theta_1 &= \begin{cases} \text{atan2}(-g_{422}, -g_{421}) + \pi & \text{if } S_2 < 0 \\ \text{atan2}(-g_{422}, -g_{421}) & \text{otherwise} \end{cases}
\end{aligned}$$

Left Arm

But for a small change, the solution for the right arm also applies to left arm. To obtain the IK solution for left arm, replace l_{A1} with $-l_{A1}$.

Decision Equations

The decision equations for the legs and arms IK solutions are obtained as follows:

$$\begin{aligned}
\text{KNEE} &= \text{sign}(S_4) & \text{HIP} &= \text{sign}(S_2) & \text{ANKLE} &= \text{sign}(5_\psi) & \text{SHOULDER} &= \\
& \text{ARM} \cdot \text{sign}(S_2) & \text{ELBOW} &= \text{sign}(S_4) & \text{WRIST} &= \text{ARM} \cdot \text{sign}(C_5) \\
\text{ARM} &= \begin{cases} +1 & \text{if solution is for right arm} \\ -1 & \text{if solution is for left arm} \end{cases}
\end{aligned}$$

B. DYNAMIC COMPUTATIONS

B.1 Jacobian matrix

Jacobian matrix is used to compute the torque required at the joints so that required amount of force can be applied at the end-effector.

$$\mathbf{T} = \mathbf{J}^T \mathbf{F}$$

The Jacobian matrix is obtained as follows:

$$\mathbf{J}_i = \begin{cases} \begin{bmatrix} \mathbf{z}_{i-1} \times^{i-1} \mathbf{p}_n \\ \mathbf{z}_{i-1} \end{bmatrix} & \text{if joint } i \text{ is rotational} \\ \begin{bmatrix} \mathbf{z}_{i-1} \\ 0 \end{bmatrix} & \text{if joint } i \text{ is prismatic} \end{cases}$$

$$\mathbf{J} = \begin{bmatrix} \mathbf{J}_1 & \mathbf{J}_2 & \dots & \mathbf{J}_n \end{bmatrix}$$

In the above equations, all the vectors are represented in base coordinates and n is the number of joints in the robot.

Using the above equation, the Jacobian matrix for Hubo2+ robot's right arm is calculated. As in the IK solutions, the Jacobian for left arm can be computed by reversing the sign of l_{A1} .

Jacobian Matrix for right arm of Hubo2+ robot

$$\mathbf{J} = \begin{bmatrix} J_{11} & J_{12} & J_{13} & J_{14} & J_{15} & J_{16} \\ J_{21} & J_{22} & J_{23} & J_{24} & J_{25} & J_{26} \\ J_{31} & J_{32} & J_{33} & J_{34} & J_{35} & J_{36} \\ J_{41} & J_{42} & J_{43} & J_{44} & J_{45} & J_{46} \\ J_{51} & J_{52} & J_{53} & J_{54} & J_{55} & J_{56} \\ J_{61} & J_{62} & J_{63} & J_{64} & J_{65} & J_{66} \end{bmatrix}$$

where,

$$J_{11} = -S_1(S_2(l_{A2}+C_4(l_{A3}+l_{A4}S_6)-l_{A4}C_5C_6S_4)+C_2(C_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)+l_{A4}C_6S_3S_5)) \\ -C_1(S_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)-l_{A4}C_3C_6S_5)$$

$$J_{21} = C_1(S_2(l_{A2}+C_4(l_{A3}+l_{A4}S_6)-l_{A4}C_5C_6S_4)+C_2(C_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)+l_{A4}C_6S_3S_5)) \\ -S_1(S_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)-l_{A4}C_3C_6S_5)$$

$$J_{31} = 0; \quad J_{41} = 0; \quad J_{51} = 0; \quad J_{61} = 1$$

$$J_{12} = C_1C_2(l_{A2}+C_4(l_{A3}+l_{A4}S_6)-l_{A4}C_5C_6S_4)-C_1S_2(C_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)+l_{A4}C_6S_3S_5)$$

$$J_{22} = C_2S_1(l_{A2}+C_4(l_{A3}+l_{A4}S_6)-l_{A4}C_5C_6S_4)-S_1S_2(C_3(S_4(l_{A3}+l_{A4}S_6)+l_{A4}C_4C_5C_6)+l_{A4}C_6S_3S_5)$$

$$J_{32} = l_{A2}S_2+l_{A3}C_4S_2+l_{A3}C_2C_3S_4+l_{A4}C_4S_2S_6+l_{A4}C_2C_3S_4S_6 \\ +l_{A4}C_2C_6S_3S_5-l_{A4}C_5C_6S_2S_4+l_{A4}C_2C_3C_4C_5C_6$$

$$J_{42} = S_1; \quad J_{52} = -C_1; \quad J_{62} = 0$$

$$J_{13} = l_{A4}C_1C_2C_3C_6S_5-l_{A3}C_1C_2S_3S_4-l_{A4}C_3S_1S_4S_6-l_{A4}C_6S_1S_3S_5$$

$$-l_{A3}C_3S_1S_4-l_{A4}C_3C_4C_5C_6S_1-l_{A4}C_1C_2S_3S_4S_6-l_{A4}C_1C_2C_4C_5C_6S_3$$

$$J_{23} = l_{A3}C_1C_3S_4+l_{A4}C_1C_3S_4S_6+l_{A4}C_1C_6S_3S_5-l_{A3}C_2S_1S_3S_4$$

$$+l_{A4}C_1C_3C_4C_5C_6+l_{A4}C_2C_3C_6S_1S_5-l_{A4}C_2S_1S_3S_4S_6-l_{A4}C_2C_4C_5C_6S_1S_3$$

$$J_{33} = -S_2(l_{A3}S_3S_4-l_{A4}C_3C_6S_5+l_{A4}S_3S_4S_6+l_{A4}C_4C_5C_6S_3)$$

$$J_{43} = -C_1S_2; \quad J_{53} = -S_1S_2; \quad J_{63} = C_2$$

$$J_{14} = l_{A3}C_1C_2C_3C_4-l_{A3}C_4S_1S_3-l_{A3}C_1S_2S_4-l_{A4}C_1S_2S_4S_6$$

$$-l_{A4}C_4S_1S_3S_6+l_{A4}C_1C_2C_3C_4S_6-l_{A4}C_1C_4C_5C_6S_2+l_{A4}C_5C_6S_1S_3S_4-l_{A4}C_1C_2C_3C_5C_6S_4$$

$$J_{24} = l_{A3}C_1C_4S_3-l_{A3}S_1S_2S_4+l_{A3}C_2C_3C_4S_1+l_{A4}C_1C_4S_3S_6$$

$$-l_{A4}S_1S_2S_4S_6+l_{A4}C_2C_3C_4S_1S_6-l_{A4}C_4C_5C_6S_1S_2-l_{A4}C_1C_5C_6S_3S_4-l_{A4}C_2C_3C_5C_6S_1S_4$$

$$J_{34} = l_{A3}C_2S_4+l_{A3}C_3C_4S_2+l_{A4}C_2S_4S_6+l_{A4}C_2C_4C_5C_6+l_{A4}C_3C_4S_2S_6-l_{A4}C_3C_5C_6S_2S_4$$

$$J_{44} = C_3S_1+C_1C_2S_3; \quad J_{54} = C_2S_1S_3-C_1C_3; \quad J_{64} = S_2S_3$$

$$J_{15} = l_{A4}C_6(C_3C_5S_1+C_1C_2C_5S_3+C_1S_2S_4S_5+C_4S_1S_3S_5-C_1C_2C_3C_4S_5)$$

$$J_{25} = -l_{A4}C_6(C_1C_3C_5-C_2C_5S_1S_3+C_1C_4S_3S_5-S_1S_2S_4S_5+C_2C_3C_4S_1S_5)$$

$$J_{35} = -l_{A4}C_6(C_2S_4S_5 - C_5S_2S_3 + C_3C_4S_2S_5); \quad J_{45} = C_1C_4S_2 - S_4(S_1S_3 - C_1C_2C_3)$$

$$J_{55} = S_4(C_1S_3 + C_2C_3S_1) + C_4S_1S_2; \quad J_{65} = C_3S_2S_4 - C_2C_4$$

$$J_{16} = l_{A4}C_1C_4C_6S_2 - l_{A4}C_6S_1S_3S_4 - l_{A4}C_3S_1S_5S_6 + l_{A4}C_1C_2C_3C_6S_4 - l_{A4}C_1C_2S_3S_5S_6 \\ + l_{A4}C_1C_5S_2S_4S_6 + l_{A4}C_4C_5S_1S_3S_6 - l_{A4}C_1C_2C_3C_4C_5S_6$$

$$J_{26} = l_{A4}C_4C_6S_1S_2 + l_{A4}C_1C_6S_3S_4 + l_{A4}C_1C_3S_5S_6 + l_{A4}C_2C_3C_6S_1S_4 - l_{A4}C_1C_4C_5S_3S_6 \\ - l_{A4}C_2S_1S_3S_5S_6 + l_{A4}C_5S_1S_2S_4S_6 - l_{A4}C_2C_3C_4C_5S_1S_6$$

$$J_{36} = l_{A4}C_3C_6S_2S_4 - l_{A4}C_2C_4C_6 - l_{A4}C_2C_5S_4S_6 - l_{A4}S_2S_3S_5S_6 - l_{A4}C_3C_4C_5S_2S_6$$

$$J_{46} = -S_5(C_4(S_1S_3 - C_1C_2C_3) + C_1S_2S_4) - C_5(C_3S_1 + C_1C_2S_3)$$

$$J_{56} = S_5(C_4(C_1S_3 + C_2C_3S_1) - S_1S_2S_4) + C_5(C_1C_3 - C_2S_1S_3)$$

$$J_{66} = S_5(C_2S_4 + C_3C_4S_2) - C_5S_2S_3$$

B.2 Newton-Euler computations

The Newton-Euler computation is used to compute the torque required at joints to move the robot. The method is basically used to solve the equation:

$$\mathbf{T}(t) = \mathbf{D}(\theta)\ddot{\theta}(t) + \mathbf{h}(\theta, \dot{\theta}) + \mathbf{c}(\theta)$$

where $\mathbf{D}(\theta)$ is the inertial matrix of the robot, $\mathbf{h}(\theta, \dot{\theta})$ is the coriolis and centrifugal vector, and $\mathbf{c}(\theta)$ is the gravity term.

If the motion is very slow, the the first two terms on the right hand side can be neglected. Thus, the torque computed is the torque required to counter gravity. Let us first define a few variables.

m_i = total mass of link i ,

\mathbf{p}^*_i = the origin of i^{th} coordinate frame with respect to $(i-1)^{th}$ frame

$\bar{\mathbf{s}}_i$ = position of center of mass of i^{th} link with respect to $(i-1)^{th}$ frame

Using, the above-mentioned approximation and the fact that all joints in Hubo2+ robot are rotary joints, the following equations for torque computation are obtained:

Forward Equations: $i = 1, 2, \dots, n$

$$\begin{aligned}\omega_i &= 0; \quad \dot{\omega}_i = 0 \\ \dot{v}_i &= \bar{a}_i = (g_x, g_y, g_z)^T\end{aligned}$$

Backward Equations: $i = n, n-1, \dots, 1$

$$\begin{aligned}\mathbf{f}_i &= m_i \bar{a}_i + \mathbf{f}_{i+1} \\ \mathbf{n}_i &= \mathbf{f}_{i+1} + \mathbf{p}_i^* \times \mathbf{f}_{i+1} + (\mathbf{p}_i^* + \bar{\mathbf{s}}_i) \times m_i \bar{a}_i \\ \tau_i &= \mathbf{n}_i^T \mathbf{z}_{i-1}\end{aligned}$$

C. HUBO-ACH MANUAL

C.1 Installation

The hubo-ach software, being open-source is available at the address <https://github.com/hubo/hubo-ach>. It is recommended to install the master branch.

System Requirements

1. Operating system must be Ubuntu. All variations of version 12.04 and 12.10 are supported.
2. System must have at least 2GB of free space.

Download and Installation Steps

1. Open Terminal and go to the desired download location.
2. Type “git clone <https://github.com/hubo/hubo-ach>”.
3. Go into the hubo-ach folder using command “cd hubo-ach”.
4. Get the latest updated program using command “git pull”.
5. Download all dependent packages and install hubo-ach using “sudo ./first-time-install-hubo-ach.sh”. Enter password if prompted for.

C.2 Usage

Starting hubo-ach also starts the hubo-console, the wrapper to send commands to the robot.

1. Power on the Hubo2+ robot’s motors.
2. Open a terminal and type the command “hubo-ach start” if you are using Hubo2+ robot and “hubo-ach start drc” if using DRC-Hubo robot.

3. Hubo-ach will start and you can type commands in the console.
4. First home all the joint using command “homeAll”.
5. Check that all the joints have homed by physical inspection and by checking the joint state using the command “statusAll”.
6. Initialize sensors using “iniSensors”.
7. Move the joints (by default in rigid mode) using the command format “goto <JointName><DesiredAngle(radians)>”, e.g. “goto LSP -0.2” is to move Left Shoulder Pitch joint to angle -0.2 radians in rigid mode.
8. Move the joints in compliance mode using the command format “comp <JointName><DesiredAngle(radians)>”, e.g. “comp LSP -0.2” is to move Left Shoulder Pitch joint to angle -0.2 radians in rigid mode.
9. Use command “kp <JointName><DesiredKPGain>” to change the kp gain in PWM mode (for safety, it should be less than 70).
10. Use command “kd <JointName><DesiredKDGain>” to change the kd gain in PWM mode (for safety, it should be less than 10).
11. Use command “maxPWM <JointName><DesiredmaxPWM>” to change the maximum PWM value (for safety, it should be less than 8).
12. To obtain a joint’s error and warning flags use command “status <JointName>”.
13. To home a particular joint use command “home <JointName>”.
14. To quit hubo-console use command “quit”.

Hubo-read is another wrapper based around hubo-ach. To use hubo-read follow the step below:

1. Go into the folder that has hubo-ach.
2. In another terminal start hubo-ach (Not required if hubo-ach is already started).
3. In case user needs to log the data use command “script filename.log”.
4. Enter command “sudo ./hubo-read”.

5. Press “ctrl+c” to exit.
6. If user is logging the data, use command “exit” to stop logging.

C.3 Support

For any suggestions, or support in installing or using the software feel free to contact any of the following:

1. Manas Paldhe, mpaldhe@purdue.edu, ArtLab at Purdue University
2. Michael Grey, mxgrey@gatech.edu, Humanoid Robotics Lab at Georgia Institute of Technology
3. Daniel Lofaro, dan@danlofaro.com Drexel Autonomous Systems Lab at Drexel University

It is not easy to read the source code of hubo-ach. A lot of backward compatibility issues were also observed with hubo-ach. To make it more readable and backward compatible, ‘HuboCan’ is being developed. It follows the same architecture and design philosophy as hubo-ach, but is more easy-to-read, backward compatible, and easily maintained. Users can also more easily contribute to the source code. The repository is available at <https://github.com/golems/HuboCan>.

D. HUBO-MOTION-RT MANUAL

D.1 Installation

The hubo-motion-rt software, being open-source is available at the address <https://github.com/manaspaldhe12/hubo-motion-rt>. It is recommended to install the master branch.

System Requirements

1. Operating system must be Ubuntu. All variations of version 12.04 and 12.10 are supported.
2. System must have at least 2GB of free space.
3. Hubo-ach must be installed.
4. RobotKin package must be installed.

Download and Installation Steps

1. RobotKin package is required to use hubo-motion-rt. The package is available at <https://github.com/mxgrey/RobotKin>. The README.md file lists out the set of instructions to install the RobotKin package.
2. drchubo-v2.urdf is required to be in ‘/etc/hubo-ach’ folder. Download the model from the link <https://github.com/hubo/drchubo/tree/master/drchubo-v2/robots> and copy it to the hubo-ach folder.
3. Open terminal and download the package using command “git clone sudo apt-get install libeigen3-dev”.
4. Type in the following commands in order to install dependent libraries:
 - (a) sudo apt-get update
 - (b) sudo apt-get upgrade

- (c) `sudo apt-get install cmake`
- (d) `sudo apt-get install libeigen3-dev`
- 5. To download the software go to the desired location and use command “<https://github.com/manaspaldhe12/motion-rt/>”.
- 6. Next, to generate a makefile using cmake package, use the commands:
 - (a) Go into the hubo-motion directory
 - (b) `mkdir build`
 - (c) `cd build`
 - (d) `cmake ..`
 - (e) Use `ls` to ensure that a file named ‘Makefile’ exists
- 7. User is now ready to compile and install. Use the next set of commands in order to install the software.
 - (a) `make`
 - (b) `sudo make install`

D.2 Usage

To start hubo-motion-rt use the command “`sudo service hubo-motion start`”. This starts and initialized hubo-ach and then hubo-motion-rt. The hubo-console comes up can can be used to move the robots. Since hubo-ach has been started, all hubo-ach dependent software packages like hubo-read, hubo-init etc can be used as well.

Hubo-motion-rt cannot be used directly. Controllers and wrappers have to be written by the developer to use the software. <https://github.com/manaspaldhe12/examples-hubo-motion-rt/> provides a few examples codes to use the software.

D.3 Support

For any suggestions or support in installing or using the software feel free to contact any of the following:

1. Manas Paldhe, mpaldhe@purdue.edu, ArtLab at Purdue University
2. Michael Grey, mxgrey@gatech.edu, Humanoid Robotics Lab at Georgia Institute of Technology

E. HUBO-READ-TRAJECTORY MANUAL

The hubo-ach software, being open-source is available at the address <https://github.com/manaspaldhe12/hubo-read-trajectory>. It is recommended to install the master branch.

System Requirements

1. Operating system must be Ubuntu. All variations of version 12.04 and 12.10 are supported.
2. System must have at least 2GB of free space.
3. Hubo-ach must be installed.

Download and Installation Steps

1. Open terminal and go to the desired location of the software.
2. Use command “git clone <https://github.com/manaspaldhe12/hubo-read-trajectory>” to download the software.
3. Enter the hubo-read-trajectory folder.
4. Makefile already exists. Thus use “make” command to compile the software.

Note that the software does not ‘install’. Thus to use it the user has to go to the software location and run it.

E.1 Usage

Hubo-read-trajectory is used to run pre-planned motion. The motion must be stored in a file in ‘space-separated-value’ format. Each line of the file is the set of desired joint angles. The order of joints is:

RHY RHR RHP RKN RAP RAR LHY LHR LHP LKN LAP LAR RSP RSR RSY

REB RWY RWR RWP LSP LSR LSY LEB LWY LWR LWP NKY NK1 NK2 WST
 RF1 RF2 RF3 RF4 RF5 LF1 LF2 LF3 LF4 LF5

To use hubo-read trajectory follow the following steps:

1. Start hubo-ach in one terminal. Home the joints and initialize the sensors.
2. In another terminal, go to the hubo-read-trajectory folder.
3. Copy the desired trajectory file to this folder.
4. In default mode, use the command “sudo ./hubo-read-trajectory -f 200 -n <trajectory file name>” to run the motion.
5. The number after ‘-f’ flag is the frequency at which the motion should run. The accepted frequencies are 10, 25, 50, 100 and 200.
6. A wide range of flags can be added to the above command in any order. They are:
 - (a) -i: To ensure that the robot smoothly goes to the fist trajectory setpoint before staring the motion.
 - (b) -c: To run the upper body motion in compliance mode with default gains.
 - (c) -cl: To run the motion with only the left arm joints in compliance mode with default gains.
 - (d) -cr: To run the motion with only the right arm joints in compliance mode with default gains.
 - (e) -nf: To run the motion in direct reference mode (no internal motion filtering).
 - (f) -p: To enable the pause and play feature. When using this if ‘p’ is pressed then the motion pauses until ‘p’ is pressed again.
 - (g) -line <number>: To begin the motion from the given line number of the input file. The user must ensure that the current robot state is close to the the state corresponding to that line of the trajectory.

E.2 Support

For any suggestions or support in installing or using the software feel free to contact any of the following:

1. Manas Paldhe, mpaldhe@purdue.edu, ArtLab at Purdue University
2. Daniel Lofaro, dan@danlofaro.com Drexel Autonomous Systems Lab at Drexel University

F. HUBO-INIT MANUAL

The hubo-ach software, being open-source is available at the address https://github.com/hubo/hubo_init. It is recommended to install the master branch.

System Requirements:

1. Operating system must be Ubuntu. All variations of version 12.04 and 12.10 are supported.
2. System must have at least 2GB of free space.
3. Hubo-ach must be installed.
4. ROS groovy must be installed.

Download and Installation Steps:

1. ROS installation: In order to use hubo-init, ROS (Groovy) must be installed. To install ROS follow the following instructions mentioned at <http://wiki.ros.org/groovy/Installation/Ubuntu>.
2. After ROS is successfully installed, the user needs to create catkin workspace. Follow the commands below in order to setup catkin.
 - (a) `mkdir -p /catkin_ws/src`
 - (b) `cd /catkin_ws/src`
 - (c) `catkin_init_workspace`
 - (d) `cd /catkin_ws/`
 - (e) `catkin_make`
 - (f) `source /catkin_ws/devel/setup.bash`
 - (g) `echo "source /catkin_ws/devel/setup.bash" >> ~/.bashrc`
3. Before installing hubo-init, user needs to ensure that the hubo-ach on hubo-computer and operator's computer are the same version.

4. After the catkin workspace has been setup, one can download and install hubo-init using the commands (in order):
 - (a) `cd /catkin_ws/src`
 - (b) `git clone https://github.com/hubo/hubo_init.git`
 - (c) `cd /catkin_ws`
 - (d) `catkin_make`

F.1 Usage:

1. Ensure that the same version of hubo-ach are installed on hubo-computer and operator's computer.
2. To use hubo-init, hubo-ach must be started on the Hubo2+ robot's computer.
3. After starting hubo-ach on hubo-computer, run command "roscore" on operator's computer.
4. Run command "roslaunch rviz rviz" in a new terminal of operator's computer.
5. If using hubo-init for the first time, then follow the steps below:
 - (a) Click on "Panels" on top and choose "Add new Panel".
 - (b) Select and add "HuboInitPanel".
 - (c) In the IP put in the IP address of the hubo-computer.
 - (d) Click Connect.
 - (e) Using the GUI after this for controlling and monitor the robot is trivial.
 - (f) When quitting choose to "save" to avoid the first three steps when using hubo-init next time.

F.2 Support

For any suggestions or support in installing or using the software feel free to contact any of the following:

1. Manas Paldhe, mpaldhe@purdue.edu, ArtLab at Purdue University
2. Michael Grey, mxgrey@gatech.edu, Humanoid Robotics Lab at Georgia Institute of Technology