Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots

A Thesis

Submitted to the Faculty

of

Drexel University

by

Daniel Marc Lofaro

in partial fulfillment of the

requirements for the degree

of

Doctor of Philosophy in Electrical and Computer Engineering Engineering

May 2013

To Mommadet and Father. You made me who I am today. You prepared me to become a better man tomorrow. Thank you.

A message to my friends, family and colleagues:

Thank you for spending your time with me. I would not trade our experiences together for anything. You are all truly unique and wonderfully people and deserve the greatest of thanks.

Mommadet, Father, Andrew, Squirt, Aunt Genn, Uncle Matt, Big Bimmel, Momma, Bucky, and Burnidet Dr. Paul Oh, Dr. Youngmoo Kim, Dr. Tom Chmielewski, Dr. Timothy Kurzweg, Dr. Adam Fontecchio **DASL**: K. Sevcik, C. Korpela, R. Ellenberg, R. Gross, D. Lofaro, A. Alspach, S. Mason, B. Sherbert, J. Hing, K. Yuvraj, P. Brahmbhatt, B. Killen, R. Vallett, Y. Jun, K. Sohn, T. Kim, Jaemi Hubo, M. Orsag, D. Castley; **ECE:** Moshe, Tanita, Kathy, Amy, Delores, Chad, Tai, Dan, Wayne, Scott, Alyssa, Dave, Manu, MET Lab and the rest of the ECE Dept. **Friends:** Kevin, Jess, Liz, Sharon, Rob, MLE, Rachel, Nate, Sandy, Keyur, Trey, Shoko, Jon, and the rest of the *Elite Gang*; Louis, Maggie, Duck, Carl, Caroline, Chris, Alex, Bella, Andrew U., Ttalg, Sarah, Mayank, Will, and the rest of the *Goon Squad*. **Korea:** Dr. Lee, Dr. JH Oh, Inhyeok, Jungwoo, Chelsea, Hubo-Lab, Woojin, Kayla, Jonghee, Meejin, Mahin; **Special Thanks to:** Jaemi Hubo, Leoben, Dirc, Simon Cavil, D'Anna, Caprica, Aaron, Mini-0, Shoko Robot.

A message to everyone:

Robots are people too.

Video: http://danlofaro.com/phd/

If you see the image above use a **QR-Code reader** or enter the URL listed above to see the digital content. The digital content consists of videos and/or interactive demonstrations.

Table of Contents

List of Tables

List of Figures

Abstract:

The degrees of freedom (DOF) of robots and complex systems have been increasing increasing exponentially since the early 20th century. Today it is common place for complex control systems to have 40 DOF. This number is projected to be 70 DOF by the year 2020. Robots with high DOF allows for complex tasks such as tool manipulation, greater human-robot interaction and agile full-body locomotion. More DOF require greater attention to local communication delays, bandwidth, system configuration and stability. In addition different tasks being performed by separate parts of the robot in tandem bring on greater issues including controller timing and priorities. The increase in DOF on single system requires that the traditional methods of controller design be re-examined.

This dissertation describes a Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots that allows a user to develop controllers using a three tier infrastructure. The Unified Algorithmic Framework called Hubo-Ach is a multi-process based system that allows for robust multi-rate simultaneous control and seamless implementation between virtual, miniature, and full-size robots with no modification. The three tier infrastructure provides different levels of cost to entry and testing. Examples of this field tested framework functioning on simulated, miniature, and full-size high DOF robots is given as well as validation by external researchers.

1. Introduction

The degrees of freedom (DOF) of robots and complex systems have been increasing increasing exponentially since the early 20^{th} century. Today it is common place for complex control systems to have 40 DOF. This number is projected to be 70 DOF by the year 2020 (see Section E). Robots with high DOF allows for complex tasks such as tool manipulation[1–4], greater human-robot interaction such as music performances[5–8] and agile full-body locomotion[9–11]. More DOF require greater attention to:

- local communication delays
- system configuration
- bandwidth
- stability

In addition different tasks being performed by separate parts of the robot in tandem bring on greater issues including controller timing and priorities. The increase in DOF on a single system requires that the traditional methods of controller design be re-examined.

Experimental results in kinematic planning (Section J), end-effector velocity control (Section K) and human-robot interaction[8] resulted in specific additional requirements for a high DOF controller for complex systems and humanoids. These requirements include:

- Robust controller integration
- High gain position controlled joints move without creating an *over torque* condition
- Live control
- Synchronous control
- Run on onboard computer
- Run in real-time

- Allow for hardware out of the loop testing
- Refined programming methods

This work describes the creation of a controller architecture for high DOF robots that achieves all of the above requirements. The system is call Hubo-Ach and has the following key attributes:

- Real-Time Performance
- Inherently robust controller integration via multi-process architecture
- No-Head of Line Blocking scheme (newest data first)
- Low CPU usage (*lean and mean*, written in C)
- Compatible with almost any simulator
- C/C++, Python and Matlab bindings
- Built-in real-time networking support (up to 1khz)
- Maximum limiting bus bandwidth
- Robot agnostic

Hubo-Ach is verified via comprehensive experiments. It is validated via third party implementation.

The Hubo-Ach system is described in detail in Section 3. Full documentation on usage and programming examples of Hubo-Ach is given in Section 4. Verification of Hubo-Ach performance is given in Section 5. Third party validation of Hubo-Ach performance is given by Zucker et. al.[12], O'Flasherty et. al.[13], and Section I. Finally a survey of 17 independent Hubo-Ach users showing overwhelming positive results is given in Section M.

This document shows that the verified and validated Hubo-Ach system is truly a *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots* because of its ability to combine a verity of control algorithms on a robot agnostic system.

1.1 Critical Gap

Due to the high entry cost for high DOF robots the Hubo-Ach controller should be able to be tested on low/no *cost to entry* systems. This means the controller must be compatible with:

- virtual/simulated robot
- full-size robot
- kinematicly scaled robot

It is evident that the critical gap is needing a *unified algorithmic framework for high degree of freedom robots that allows for development on multiple platforms.* This unified algorithmic framework connects the three robots above in the *Three Tier Infrastructure*[14] for complex system development as described in Section 1.2. The idea for this infrastructure was first realized by a Partnerships for International Research and Education (PIRE) grant #0730206, sponsored by the the U.S. National Science Foundation (NSF). This is the same grant in which this work is sponsored. The three tiers include:

- Rapid Prototype (RP) phase with zero cost to entry (OpenHubo Platform Section 2.4.3)
- Test and Evaluation (T&E) phase with low cost to entry (Mini-Hubo Platform Section 2.4.2)

- Verify and Validate (V&V) phase with lease-time cost to entry (Hubo Platform Section 2.4.1)

The unifying algorithmic framework called *Hubo-Ach*[1] is described in Section 3.

As described above this work demonstrates that a multi-process, multi-rate control structure coupled with the proper timing mechanisms is conducive to creating this unified algorithmic framework. Through verification and validation Hubo-Ach is shown to be a viable unifying algorithmic framework conducive to collaborative work. A road map of how this work began is shown in Section 2.1.

An example of the three tier infrastructure being used to enable a high DOF robot to throw a ball is given in Section K. The methods used include a unique algorithm for end-effector velocity control called Sparse Reachable Maps (SRM) is explained in Section K.1. In addition an end-effector velocity control method is used in a live throwing experiment at a baseball game and described in Section L.1.

The Hubo-Ach system is verified under many circumstances including:

- Real-time closed form inverse kinematic controller (Section 3.6)
- Full body locomotive task of turning a valve (Section 3.6.1)
- Full body locomotive task of walking (Section 5.1.2)
- Visual seroving while performing full body locomotive task (Section 5.2)
- Active damping via force-torque feedback (Section 5.3)

Hubo-Ach is then independently validated by other researcher through the examples of:

- Door opening (Section 3.8)
- Dynamic walking (Section I)

A study/survey about how well the Hubo-Ach system performs as a *unifying algorithmic framework* is given. Results and the questions are given in Section M.

Lastly Section 6 discusses the results of the work and the future of this system.

Note: This work has already been validated by pears in the field through:

- Use as the primary control system for the DARPA Robotics Challenge Track-A Team DRC-Hubo, Section F.
- Used in the NSF-PIRE[1] and NSF-MIRR[2] projects.
- Various research being conducted using Hubo-Ach at MIT, WPI, Purdue, Ohio State, Swarthmore College, Georgia Tech, and Drexel University.

In addition more information on *why this problem is hard* is in Section 1.3.

For the remainder of this document the focus will be on implementing this Unifying Algorithmic Framework on the different platforms of the three tier infrastructure. These platforms are Hubo, Mini-Hubo, and OpenHubo. Detailed description of each of these robots are available in Section 2.4.

1.2 Three Tier Infrastructure

The three tier infrastructure allows for:

- Rapid Prototype (RP) phase with zero cost to entry (OpenHubo Platform Section 2.4.3)
- Test and Evaluation (T&E) phase with low cost to entry (Mini-Hubo Platform Section 2.4.2)

[1]NSF-PIRE: Partnerships for International Research and Education (PIRE) #0730206, sponsored by the the U.S. National Science Foundation (NSF)

[2]NSF-MIRR: Major Research Infrastructure Recovery and Reinvestment (MIRR) #CNS-0960061 sponsored by the the U.S. National Science Foundation (NSF)

- Verify and Validate (V&V) phase with lease-time cost to entry (Hubo Platform Section 2.4.1)

The OpenHubo[15] kinematic and dynamic model in OpenRAVE[16] is the RP for this document. The T&E phase is a miniature kinematically scaled structure to that of the Hubo platform. Mini-hubo[14] acts as the model for the T&E phase. The Hubo platform[17] is used as the full-size humanoid for this infrastructure. A key aspect is that a single controller commands all three of the phases. This controller is called Hubo-Ach and is discribed in Section 3. Fig 1.1 shows the three tier infrastructure for the Hubo platform.

The key point of the three tier infrastructure is allowing for testing on the RP. When the algorithms applied to RP works, moving to the T&E phase is the next step. If it does not work in T&E then the cycle states movement back to the RP phase. If it does work then movement to the V&V phase is the next step. If application to V&V is not successful then the cycle states movement back to T&E or RP phase. If it does work then the project is complete [14].

1.3 Challenges

This problem is hard when each controller has different frequencies, timing requirements (asyncronous vs. syncronous), latency restrictions, newest state data is more important then older state data and most basic of all languages the controller is written in. This is especially true for complete and complex autonomous systems. I define a complete and complex autonomous system as an electro mechanical mechanism with high degree of freedom (DOF) that is capable of making its own decisions through the use of sensor data processed by its artificial intelligence (AI). The combination of high DOF and the requirement for autonomy makes the work space broad and controllers complex. The overarching question becomes; What is the control

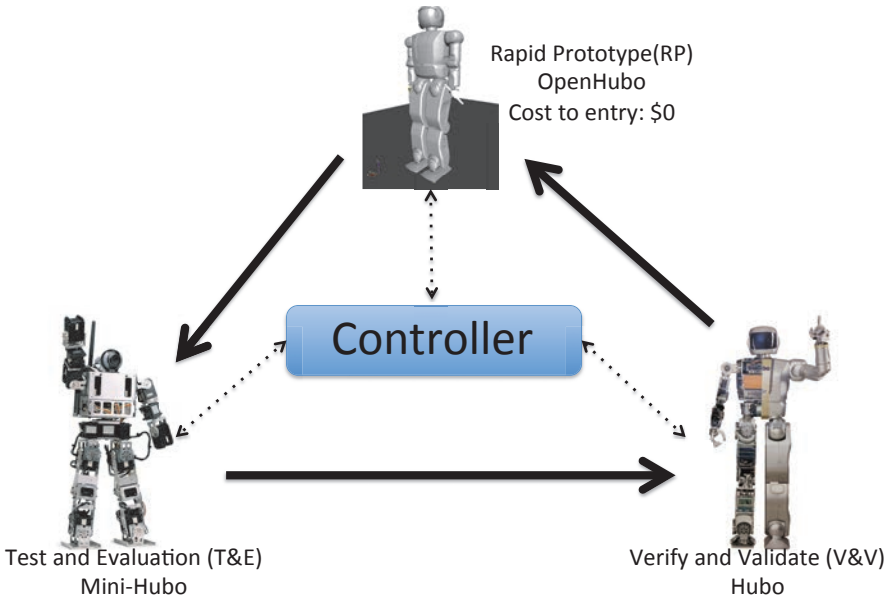


Figure 1.1: Three tier infrastructure. Tier 1: Rapid Prototype (RP) using OpenHubo. Tier 2: Test and Evaluation (T&E) using Mini-Hubo. Tier 3: Verify and Validate (V&V) using Hubo.

system structure for a complete and complex autonomous systems with high DOF, a multitude of sensors, AI performing high-level and low-level tasks all while keeping a stable system structure conducive to collaborative work? Current methods of solving the problem of controller synchrony and latest state data is to keep your critical control elements in the primary control loop. Inter-process communication (IPC) and/or network sockets to communicate between the high level and low level processes even if written in different languages. The majority of IPC have the problem of *head of line* blocking (HOL) which means you must read the older data in a buffer before you read the newest data. In the computer science field this is not a problem because all data being intact is typically desired. In the field of robotics and control the most recent state data is more important to a real-time control system to act on. This thesis shows that by expanding on the idea of multi-process controllers connected to high-speed low-latency IPC you can create a *robot layer* on a computer platform that will allow low-level controllers to run in separate processes while still allowing them access to the most recent data as the priority. The new technical idea is the *robot layer*, a control layer that allows external processes to run like normal and not deal with the specifics of the given robot system. The robot system can be replaced by a simulated system without any of the processes needing to be modified or even know of the change. This allows more mature controllers to be easily interfaced with this system without modifying control rates or timing. This *robot layer* must be:

- Have a IPC latency much less then that of the robot's inherent sampling period $t_{ipc} << T_r$
- Allow for command rates much slower then the inherent sampling period $T_{slow} >> T_r$
- Allow for command rates much faster then the inherent sampling period $T_{fast} <<$

T_r

- Allow for arbitrary command rates.
- Allow for real-time and non-real-time controllers to command actuators
- Allow for all processes to have access to the newest data first
- Allow for no more then one rt time step delay between command and robot actuator retrieval
- Commanded such that it is for an arbitrary robotic actuator.
- Triggering for process synchronization
- Triggering for simulator synchronization and holding

We can succeed now not only because the bleeding edge technology allows for the fast enough communication between processes with access to the latest data.

Results are measured quantitatively and qualitatively. Data showing proper loop rates, timings, controller implementation, simulation connections etc. show the viability of the system. User survey shows methodology is sound, useful, and practical.

1.4 Controbutions and Vertical Leap

The primary contributions and vertical leap to the field is the creation of a *unified algorithmic framework for high degree of freedom complex systems and humanoid robots*. The resulting framework allows seamless integration of:

- Controllers running at different loop rates
- Runs on multiple robots with no modification
- Runs on simulated robots with no modification

- Inherent structure makes it more robust
- Written in C for controller programing language interdependence (use C bindings in desired language)

The unified algorithmic frame work Hubo-Ach is an Open-Source BSD licensed software allowing for open use.

The contributions of Hubo-Ach have been independently verified by external parties[12, 13]. Is has been validated by multiple IEEE publications[2, 3], in review for a publication in the IEEE Robotics and Automation Society Magazine (RAM)[1] and has been the top featured video on the IEEE Spectrum *Video Friday* article on December 14^{th}, 2012[18].

2. Background and Results from Preliminary Experiments

This section gives a brief background and results from preliminary experiments of the methods used to complete the Hubo-Ach system. The motivation and timeline of experiments is given in Section 2.1. Different control system structures are discussed in Section 2.2. Section 2.3 describes why inter-process communication (IPC) is used for the Hubo-Ach control system and as well as a brief background of different IPC methods. Section 3 gives this background in greater detail. Finally Section 2.4 describes the different platforms used to validate Hubo-Ach the *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots*.

2.1 Motivation

This section provides context to the origin of the idea of a unified algorithmic framework for complex systems called Hubo-Ach.

In summer 2008 Daniel participated in the NSF-EAPSI (East Asian and Pacific Summer Institute) allowing him to study at the Hubo-Lab at KAIST to learn how to maintain, operate and program the Hubo series robot. In Fall of 2008 Daniel started his work on a Hubo KHR-4 model in the Drexel Autonomous Systems Lab (DASL) at Drexel University. Shortly after that in Spring of 2009 he had the robot performing interactive musical tasks such as listening to music and autonomously tapping its hand to the beat[8]. This was the first of many experiments in sensor integration on the Hubo. Later that spring Daniel and the rest of DASL showed Hubo to the public at a live demonstration at the Philadelphia Please Touch Museum. In winter of 2009 the first of the visual feedback methods was implemented [10]. In 2010 Daniel investigated brain machine interfacing with the robot as well as multi-modal sensing

using visual and auditory cues[7]. In late 2010/early 2011 Daniel started on his first throwing experiments which culminated in making the Hubo robot throw the first pitch at a Major League Baseball game[9]. A timeline of Daniel's work can be seen in Fig. 2.1.

Throughout this work Daniel quickly realized that there was no simple and robust way of integrating controllers on top of the existing Hubo control system. Hubo's original control system was written by Hubo-Lab in the Windows environment utilizing the Real-Time Extension (RTX) for Windows API. This controller is a typical single loop, single process, real-time controller that gets its high level input via flags and data fields located in shared memory. As the controller gets more complex it became more and more probable that something would throw a fault. If one part of the controller failed the whole system fails.

The Daniel worked with the Drexel Autonomous Systems Lab to create a linux based controller for the Hubo called ACES/Conductor. This controller designed by Sherbert et. al.[19] is a multi-threaded real-time controller that breaks each joint into individual devices. Each of these devices has multiple layers including the hardware, control, and command layer. Each of these layers runs their own real-time loop, sharing data via pointer passing for dynamic memory fields. Though the theoretical concept for this controller was sound, proper implementation would be suitable for an FPGA, GPU or other processors with many cores. Because each device had multiple real-time loops associated with it (one for each layer) and there are many devices (joints) on the Hubo the CPU usage was high. When running on a dual core 1.6 Ghz Intel Atom Processor a constant 100% CPU usage was recorded. This did not allow other processies to run in tandem. In addition there was an inherent memory leak in the dynamic memory. This was brought about from the system never being able to guarantee that another thread is not using a block of memory, thus it is never

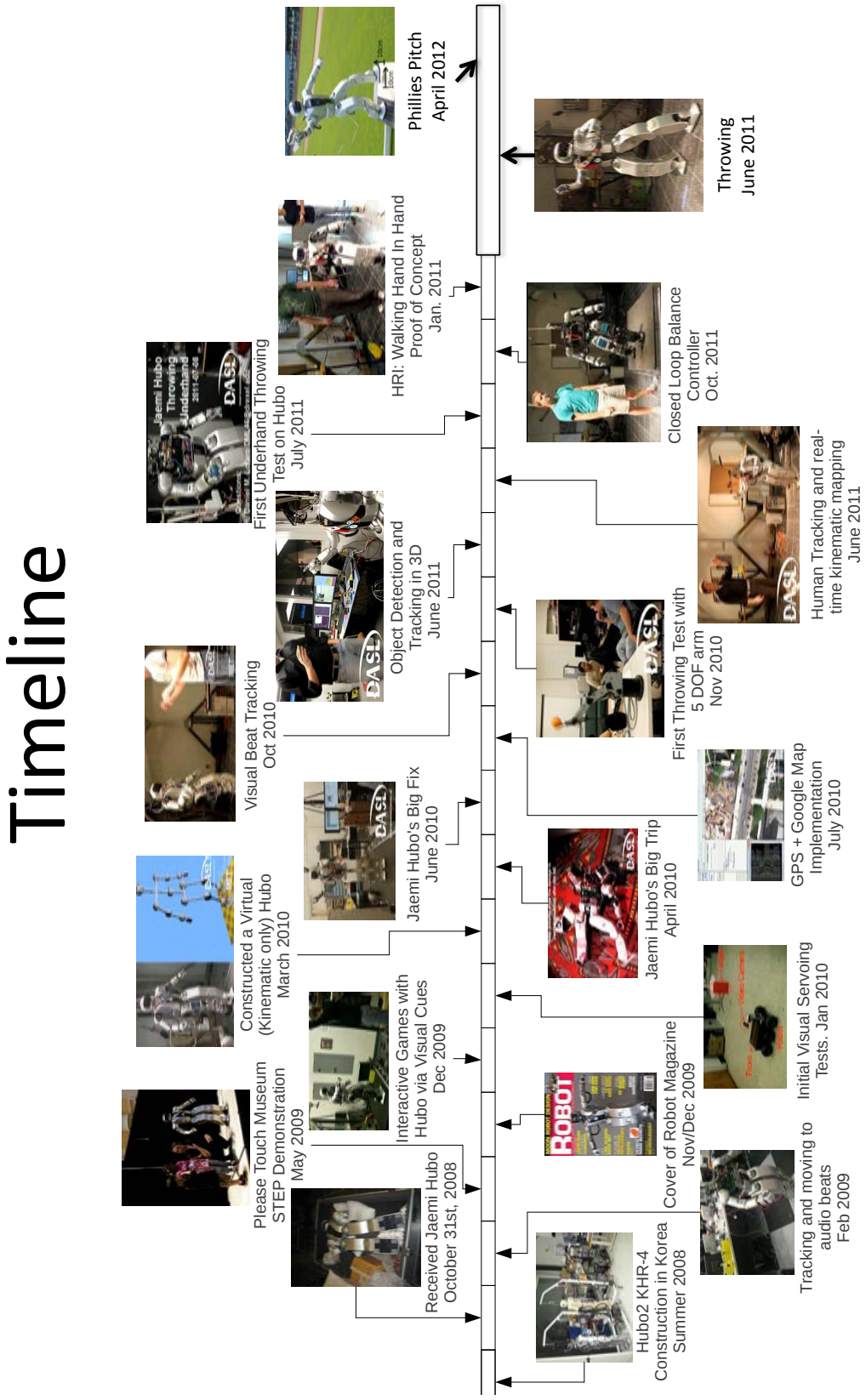


Figure 2.1: Timeline of Daniel M. Lofaro's research from 2008 to 2012

trashed. This caused the memory usage to increase at a predictable rate. This would cause a complete system crash when the memory size surpassed the available memory of the system. It was found that this system was not suitable to run the Hubo robot.

Learning from the past experiences Daniel sought out to create a controller that could:

- handel high degrees of freedom
- runs in real-time
- simple sensor integration
- low CPU usage

This goal was realized with the creation of Hubo-Ach as described in Section 3. Further examples of why Hubo-Ach was created can be found in Appendix K

2.1.1 Human Robot Interaction Preliminary Experiments

The initial goal was to have a humanoid robot become an interactive musical participant with humans. This spawned the creation of a visual method of tracking the beat in the absence of auditory cues[8]. This came from a modification of a method of allowing children to play interactive games with humanoid robots[10]. The resulting method was effective, but to increase the accuracy it was required to combine a pre-existing auditory beat tracker with the visual system. This calumniated with a multi process system that combine the auditory and visual beat trackers[5–7]. A human comparison was completed and found that this combined method was as accurate at detecting the beat in music as average humans.

Results from preliminary experiments

When collaborating with other to create a complex robot control systems integrating controllers is difficult because of the use of:

- different loop rates causing synchronization issues

- different programming languages making using the same libraries a challenge

It was found that it is best to keep each working systems *independent* allowing them to run at their native rate and on their native platforms[20].

2.1.2 High Degree of Freedom Kinematic Planning Preliminary Experiments

The next challenge was to perform kinematic planning for end effector velocity control. This resulted in the development of a method that is able to solve inverse kinematics (IK) for high degree of freedom (DOF) systems where there is no closed-form solution as well as create collision free trajectories for high DOF robots[15]. This is described in detail in Section L and K. This culminated in the verification and validation of the system by an experiment where Hubo full-size humanoid robot throw the first pitch at a Major League Baseball (MLB) game[9, 21].

Results from preliminary experiments

As best practice when controllers and planners are implemented it is important that low-level controllers such as balance and obstacle avoidance run at all times[1]. Non-priority controllers such as throwing trajectory planning can run in the background in a separate process. Keeping the processes separate allowed the system to be more resistant to lag and crashes of one or more of the controllers. This brought validation to the overarching plan for the unified algorithmic framework for complex systems and humanoid robots.

2.1.3 Lessons Learned

At this point creating these experiment it was required to *hacked* together pre-existing systems that allowed the robot to do the task. This is the point where it

was realized that a *unified algorithmic framework for complex systems and humanoid robots* was required for further development in the field. Key lessons learned from these experiments were:

- Must inherently decouple controllers loop rates and phases
- Must allow for collaborators not have to *inject* their code into existing source.
- Must work with multiple robots for testing, evaluation, validation, and verification.

This is where Hubo-Ach was born. The idea was to create a multi process architecture for humanoid control using state of the art high-speed low-latency Inter-Process Communication (IPC) techniques[1]. This is different from traditional IPC techniques because of the lack of head of line (HOL) blocking and focus on low-latency. Section 3.2 gives further details and comparisons of different IPCs.

The need for this unified framework was amplified when the Hubo was chosen to be the primary platform for the DRC-Hubo[1] Track-A team. Since its initial conception Hubo-Ach has become a fully functional system used in active research by multiple universities including MIT, WPI, Purdue, Ohio State, Swarthmore College, Georgia Tech, and Drexel University[2, 3]. This research also acts as a key source of verification and validation of the system.

2.2 Control System Structures

The traditional single loop control structure that is used in robot control software such as Orocos[22], Microsoft Robot Studio[23], RobotC[24], MATLAB[25] and LabVIEW[26] are not suited for high DOF robots. Due to the nature of these highly

[1]DRC-Hubo: http://www.drc-hubo.com/

redundant complex electrical mechanical system it is common to desire multiple different controllers running in tandem. Different controllers are needed when the system is in different states or doing different tasks or performing multiple tasks at the same time. Combining these controllers is a problem in complex system. This problem is hard when each controller has different loop rates that are not even multiples of on another, timing requirements (asynchronous vs. synchronous).

Multi-threaded approaches using shared memory allow for compatibility of multiple loop rates such as in Lee et. al.[27] with multi-threaded controllers on their humanoids, Rai et. al.[28] with multi-threaded controllers on their snake robots and Zheng et. al.[29] with multi-threaded controllers on their under water robots. The multi-threaded approach still has an inherent flaw. If the parent or one of the other controller threads crashes it is difficult or impossible to restart the controller and still have access the the shared memory.

By using a multi process approach allows controllers to fault and restart with minimal effect on the other controllers. Typical ways of communicating between different processes is via UDP or TCP/IP such as OpenHRP[30] with their server based control platform for the HRP humanoid robot; Aramaki et. al.[31] and Lofaro et. al.[7] with their multi-computer based control methods and the popular Robot Operating System (ROS)[32] by Willow Garage.

Communicating via TCP/IP sockets, such as in OpenHRP and ROS, guarantee that the data is received but it does not guarantee a arrival time. This means if the checksum fails the message will be sent again increasing the latency of the message. This does not work well if a real-time control loop is required. Using UDP does not resend if the checksum fails. This keeps the latency low and is better for real-time applications such as in the work of Lofaro. Both UDP and TCP/IP require that the buffer is read before new data is read. This means that you must read the older data

before newer data. This is called head of line blocking or HOL[20].

Newer state information is preferred by robots that work in the physical world over older data. Thus it is desired that HOL is eliminated. This can be done with some forms of inter-process communication (IPC).

OpenHRP and Webots[33] are two of the very short list of systems that have simulators that use the same controller as the hardware platforms. However at this time to the best of our knowledge there is no system that:

- uses the same controller with the software and hardware systems
- is inhreently robust by using a multi-process approach
- uses low-latency methods for controller communications

2.3 Multi-Process and Interprocess Comunication

This section gives a quick background to why inter-process communication (IPC) is used for the Hubo-Ach control system and a brief background of different IPC methods. Section 3 give this background in greater detail.

The idea for a Control Architecture for High DOF robots stems from a gap in physical implementation of control algorithms for robot hardware. The simplest approach to developing robot software is to integrate all functionality in one program. This functionality includes the following controllers:

- Hardware Control
- Kinematics
- Perception
- Planning
- etc.

If all of this functionality is in one process then it has the benefit of freedom of inter process communication latency. However being in one process also means that

if one of the controllers lags or faults it cause the entire controller to lag or fault. This is of great concern if a non-priority controller such as vision processing faults causing a priority controller such as a balance controller, to fail. This will cause the robot to fall. How is this fixed? One solution and my proposed solution is to use multiple processes and IPC methods. Inter-process communication is a method of exchanging data between multiple processes. Typical POSIX methods give you the **oldest** information first and have locks on the memory when processes are writing to it. An overview of these mechanisms are given in [34].

Robots work in the physical world. More recent information is more important to it then older. In most cases it is acceptable to know the most recent data and never read any of the older data. This would happen if your sensors update at a faster rate then that of the robot. Typically robot actuators have a bandwidth much much lower then that of a modern computer. If sensor information is shared using traditional shared memory over POSIX methods the controller would have to read the older information before it reaches the information it is most interested in, the newest data. This is known effect but new concern for robot controllers called head of line blocking[20].

It is desired to make a multi-process controller that can share data between multiple processes with low-latency and no head of line blocking. There are a few IPCs that offer no head of line blocking and low-latency. A description of each IPC type is in Section 3. Table 3.2 shows a full comparison of the different IPC types.

My thesis Hubo-Ach is a multi-process control system that uses IPC methods to communicate between processes. Section 3 describes Hubo-Ach in detail.

2.4 Platforms

This section describes the different platforms that are focused on in this document.

2.4.1 Hubo2 Plus

The Hubo2 Plus series robot is a 130 cm (4' 3") tall, 42 kg (93 lb) full-size humanoid commonly refereed to as Hubo. The Hubo series was designed and constructed by Prof Jun-Ho Oh at the Hubo Lab in the Korean Advanced Institute of Science and Technology (KAIST) in Daejeon, South Korea [35]. Hubo has 2 arms, 2 legs and a head making it anthropomorphic to a human. It contains 6 degrees of freedom (DOF) in each leg, 6 in each arm, 5 in each hand, 3 in the neck, and 1 in the waist; all totaling 38 DOF. All joints of the major joints are high gain PID position controlled with the exception of the fingers. The fingers are open-loop PWM controlled. The sensing capability consists of a three axis force-torque (FT) sensor on each leg between the end of the ankle and the foot as well as between the arm where it connects to the hand. Additionally it has an inertial measurement unit (IMU) at the center of mass and accelerometers on each foot. The reference commands for all of the joints are sent from the primary control computer (x86) to the individual motor controllers via two Controller Area Network (CAN) buses. There are currently eight Hubo's functioning in the United States as of December 2012. Jaemi Hubo is the oldest of the Hubos in America and has been at the Drexel Autonomous Systems Lab[2] (DASL) since 2008 [36]. Fig. 2.2 shows the major dimensions of Hubo. Table 2.1 shows the other attributes of the Hubo.

A full-scale safe testing environment designed for experiments with Jaemi Hubo was created using DASL's Systems Integrated Sensor Test Rig (SISTR) [37]. Additionally all algorithms are able to be tested on miniature and virtual versions of Jaemi Hubo prior to testing on the full-size humanoid through the creation of a surrogate testing platform for humanoids [38].

[2]Drexel Autonomous Systems Lab: http://dasl.mem.drexel.edu/

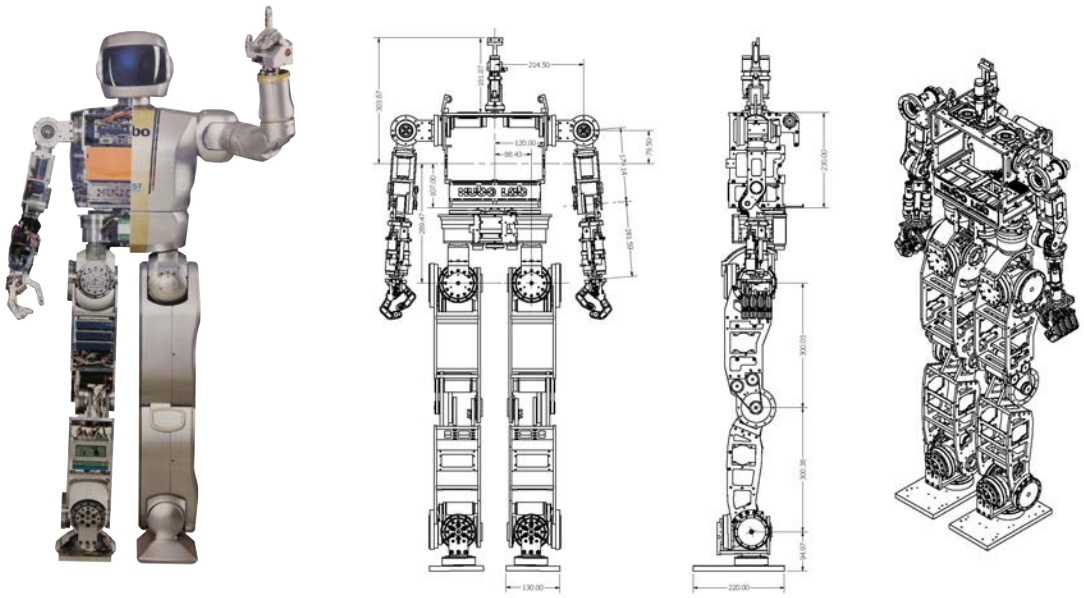


Figure 2.2: Hubo2 Plus platform: 40 DOF, 130 cm tall full-size humanoid robot weighing 37 kg.

Table 2.1: Hubo2 Plus (Hubo) Platform Specifications

Height	130 cm
Weight	37 kg
DOF	40
Joint Control Type	High-Gain PID Position
Computer	1.6 Ghz Atom 1 Gb DDR3 RAM
Operating System	Debian Linux Windows
Battery	$54V$ 7.5 Ah $40C$ LiPo
Sensors	1x 4 Axis IMU 4x 3 Axis Force Torque 2x 2 Axis Tilt
Vision	Stereo Monocular RGBD

Table 2.2: Mini-Hubo Platform Specifications

Height	46 cm
Weight	2.9 kg
DOF	22
Joint Control Type	PID Position
Computer	1.6 Ghz Atom 2 Gb DDR2 RAM
Operating System	Debian Linux
Battery	14.8V 3.2 Ah 30C LiPo
Sensors	2x 3 Axis Force Torque
Vision	Monocular RGBD

2.4.2 Mini-Hubo

Mini-Hubo[14] is a miniture version of the Hubo platform discribe in Section 2.4.1. It is used as the Test and Evaluation (T&E) stage of the three tier infrastructure discribed in Section 1.2. Mini-Hubo is kinematically scaled to the Hubo platform. The atrobutes of the Mini-Hubo system are in Table 2.2. The robot is shown in Fig. 2.3

2.4.3 OpenHubo

OpenHubo[39] is an open-source kinematic and dynamic simulator for the the Hubo2 and Hubo2+ series robots. It was developed by the Drexel Autonomous Systems Lab and runs using the open-source robot simulation environment OpenRAVE[16]. Fig. 2.4 shows the OpenHubo model. Table 2.3 shows the spesifications of OpenHubo.

Figure 2.3: Mini-Hubo platform: 22 DOF, 46 cm tall miniture-size humanoid robot weighing 2.9 kg.

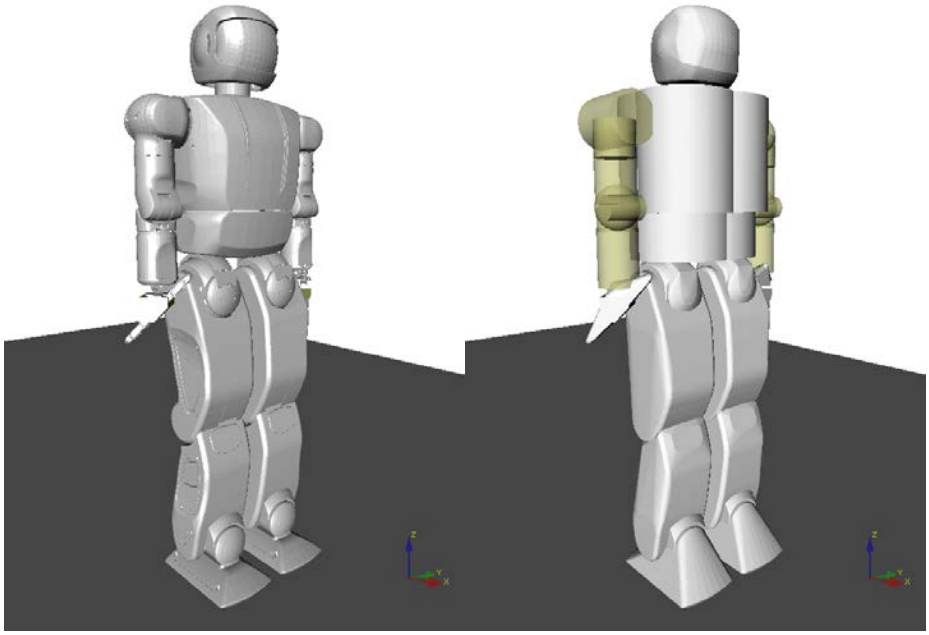


Figure 2.4: OpenHubo model of the Hubo2 humanoid robot developed by the Drexel Autonomous Systems Lab and runs using the open-source robot simulation environment OpenRAVE[16].

Table 2.3: OpenHubo Platform Specifications

Dynamics	Yes (ODE)
Kinematic	Yes
DOF	40
Joint Control Type	PID Position
Computer	1.6 Ghz Atom 2 Gb DDR2 RAM
Enviroment	OpenRAVE[16]
Sensors	1x 4 Axis IMU 4x 3 Axis Force Torque

3. Hubo-Ach: A Unified Algorithmic Framework for High DOF Robots

This section describes in detain the *unified algorithmic framework for high degree of freedom complex systems and humanoid robots*. An overview of the system is given in Section 3.1. Timing and system testing is given in Section 3.3 Validation examples are given in Section 3.5, 3.6 and 3.6.1. Verification of Hubo-Ach from independent parties is given in Section 3.8 and results from surveys about the system is given in Section M.

3.1 Overview

Hubo-Ach [1] is an multi-process unified algorithmic framework for high degree of freedom complex systems and humanoids. In this case spisificall Hubo. This provides a conventional GNU/Linux programming environment, with the variety of tools available therein, for developing applications on the Hubo. It also efficiently links the embedded electronics and real-time control to popular frameworks for robotics software: ROS [40], OpenRAVE,[2] and MATLAB[3].

Reliability is a critical issue for software on the Hubo. As a bipedal robot, Hubo must constantly maintain dynamic balance; if the software fails, it will fall and break. A multi-process software design improves Hubo's reliability by isolating the critical balance code from other non-critical functions, such as control of the neck or arms. For the high-speed, low-latency communications and priority access to latest sensor feedback, Ach provides the underlying IPC. Fig. 3.1 shows a block diagram with multiple controllers in multiple processes comunicating with Hubo-Ach. The diagram also shows that Hubo-Ach works with the RP, T&E and V&V stages seemelessly.

[1] Available under permissive license, http://github.com/hubo/hubo-ach
[2] OpenRAVE: http://openrave.org/
[3] MATLAB: http://www.mathworks.com/

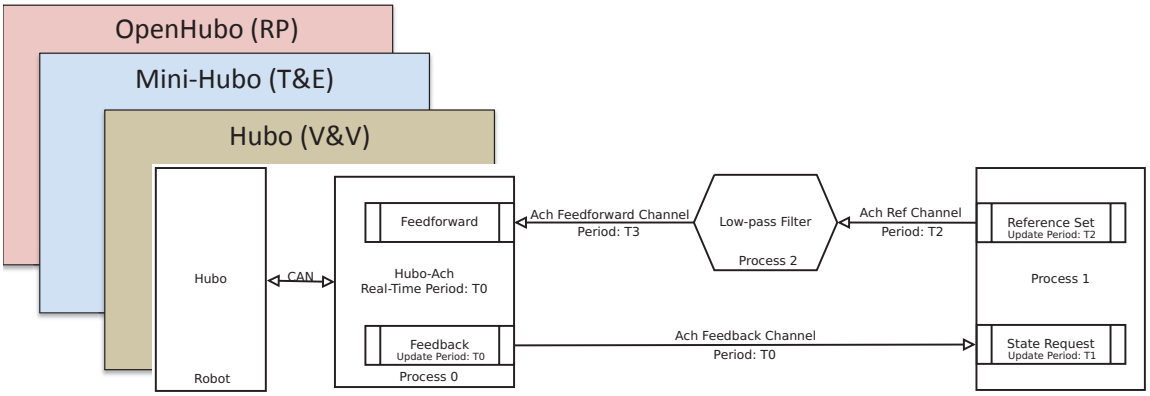


Figure 3.1: Hubo-Ach simple block diagram showing multiple controllers in multiple processes. Diagram also shows that Hubo-Ach works with the RP, T&E and V&V stages seemelessly.

Hubo-Ach handles CAN bus communication between the PC and embedded electronics. Because the motor controllers synchronize to the control period in a *phase lock loop* (PLL), the single `hubo-daemon` process runs at a fixed control rate and communicates on the bus. The embedded controllers lock to this rate and linearly interpolate between the commanded positions, providing smoother trajectories in the face of limited communication bandwidth. This communication process also avoids bus saturation; with CAN bandwidth of 1 Mbps and 200Hz control rate, `hubo-daemon` currently utilizes 78% of the bus. `Hubo-daemon` receives position targets from a `feedforward` channel and publishes sensor data to the `feedback` channel, providing the direct software interface to the embedded electronics.

Each Hubo-Ach controller is an independent processes. The controllers handle tasks such as balance, manipulation, and human-robot interaction. Each controller asynchronously reads state from the `feedback` Ach channel and sets reference positions in the `feedforward` channel. `Huobo-daemon` reads the most recent reference position from the `feedforward` channel on the the rising edge of its control cycle. This allows the controller processes to run at arbitrary rates without effecting the

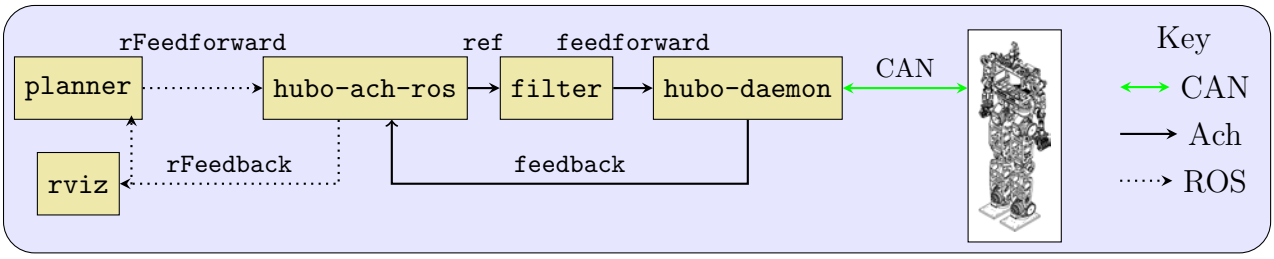


Figure 3.2: Feedback loop integrating Hubo-Ach with ROS

PLL of the embedded motor controllers or the CAN bus bandwidth utilization.

Figure 3.2 shows an example control loop integrating Hubo-Ach and ROS. The `hubo-daemon` communicates with the embedded controllers at 200Hz, publishing to the `feedback` channel. The `hubo-ach-ros` process bridges ROS topics and Ach channels. It translates messages on the `feedback` Ach channel to the `rFeedback` ROS topic and translates the `rFeedforward` ROS topic to the `ref` Ach channel. The `planner` process computes desired trajectories, which are relayed via `hubo-ach-ros` to `filter` for preprocessing to smooth the motion and reduce *jerk* before `hubo-daemon` communicates references to the embedded controllers. During operation, `rviz` displays a 3D model of the Hubo's current state. This is important because it allows for simple human feedback and diagonostics.

All of the above process runs asynchronously, communicating at different rates; however, `hubo-ach-daemon` maintains its 200Hz cycle, ensuring phase lock with the embedded controllers. This control loop effectively integrates real-time IPC and control under Hubo-Ach with the non-real-time ROS environment.

Hubo-Ach is being verified and validated through use in numerous projects at several research labs. In addition it is getting real-world validation by being the projects primarily revolve around the DARPA Robot Challenge (DRC)[4] team DRC-

[4]DARPA Robot Challenge: http://www.theroboticschallenge.org/

Hubo[5]. The DRC includes rough terrain walking, ladder climbing, valve turning, vehicle ingress/egress and more. Figure 3.26 shows the Hubo using the Hubo-Ach system to turn a valve.

Hubo-Ach provides an effective base for developing real-time applications on the Hubo. Separating software modules into different processes increases system reliability. A failed process can be independently restarted, minimizing chance of damage to the robot. In addition, the controllers can run at fast rates because Ach provides high-speed low-latency communication with `hubo-daemon`. Hubo-Ach provides a C API that is easily called from high-level programming languages and integrates with popular platforms for robot software such as ROS and MATLAB, providing additional development flexibility. Hubo-Ach is a validated and easy to use interface between the mechatronics and the software control algorithms of the Hubo full-size humanoid robot.

3.2 Inter Process Comunication Comparision

POSIX provides three main types of IPC: streams, datagrams and shared memory. A review of each is made before making a choice for desired message passing skeam.

Streams:

The IPC type *stream* includes pipes, FIFOs, stream sockets, and TCP sockets. All stream basted methods suffer from head of line (HOL) blocking which means older data **must** be read before newer data. For robotic applications we must be able to access the newest data imediately and read older data if needed. This is a different paradime then typical streaming application because robots are real-time sensitive meaning the newest information holds more value to the overall system than the older data.

[5]DRC-Hubo Homepage: http://drc-hubo.com/

Datagrams:

POSIX *datagrams* come in two major flavors, *datagram sockets* and *POSIX message queues*. Datagram sockets are less likely to block the sender then streams. The most important reason why datagrams are **not** a good solution for my application is that newer messages are lost if the buffer is filled. Newer data is more important than older data in my control system thus this is not a viable option.

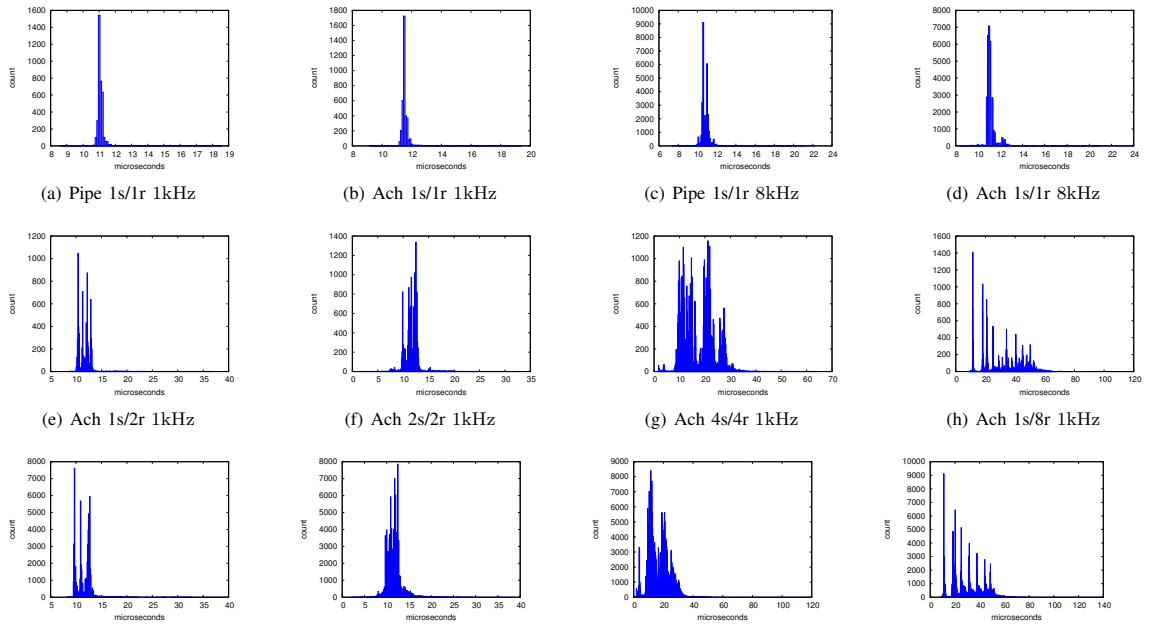
POSIX message queues are simular to datagrams sockets with the addition of message priorities. Unlike datagram sockets if the buffer fills the POSIX message queues will block. This will cause the application to stop processing until it is able to read/flush the old messages. Thus simular to other methods mentioned this also suffers from HOL.

Shared Memory:

POSIX shared memory is very fast and allows access to the latest data by simply writing over a variable. Though I have been advicating that the newest information is the most important, old information can not be discarded. If using POSIX shared memory there is no way of recovering older data that might have been missed by a controller.

What is needed is a method of sharing data that is *non-blocking* and as *low-latancy* like shared memory, but still holds older data and uses an asyncronous IO scheme. The asyncronous IO scheme is required so the controller is not locked to a set rate by the data transactionn method. N. Dantam et. al.[20] shows that Asynchronous IO (AIO) might be approperiate for this application however the implimentaiton under Linux is not as mature as I require. In addition N. Dantam shows that other IPC mechanism using select/poll/epoll/kqueue are widely used network server and help midigate but not totally removed the issue of HOL. The primary problem being that that thought the sender will not block the reader must stil read the oldest data first.

The question now is what IPC mechanism will be suitable for my control system.



(a) Pipe 1s/1r 1kHz (b) Ach 1s/1r 1kHz (c) Pipe 1s/1r 8kHz (d) Ach 1s/1r 8kHz

(e) Ach 1s/2r 1kHz (f) Ach 2s/2r 1kHz (g) Ach 4s/4r 1kHz (h) Ach 1s/8r 1kHz

(i) Ach 1s/2r 8kHz (j) Ach 2s/2r 8kHz (k) Ach 4s/4r 8kHz (l) Ach 1s/8r 8kHz

Figure 3.3: Histograms of Ach and Pipe messaging latencies. Benchmarking performed on a Core 2 Duo running Ubuntu Linux 10.04 with PREEMPT kernel. The labels αs/βr indicate a test run with α sending processes and β receiving processes[20].

Upon investigation three major mechanisms are avaliable; Robot Operating System (ROS)[32], Message Passing Interface (MPI)[41] and Ach[20]. Though ROS is ubicuitious in the robotics and automation field the inherent latency and the non real-time (RT) guarantee due to the use of TCP/IP it is not a good choice. MPI is ubiquitous in the high-proformance computing field. It has full non-blocking capiabilities and is geared towards maximizing message throughput for networked clusters[20]. Table 3.2 shows a comparision of a wide range of IPC methods. A focus on reducing latency is not given. Ach does focuses on latency. Fig. 3.3 shows histograms of Ach and Pipe messaging latencies. Benchmarking performed on a Core 2 Duo running

Table 3.1: Robot control system comparison

System	Open Source	POSIX Complaint	Non Blocking	Real Time	Low Latency	Light Weight
ROS	yes	yes	no	no	no	no
Orocos Real-Time Toolkit	yes	yes	no	yes	yes	no
Robotics Technology Middleware	yes	yes	no	yes	yes	no
Microsoft Robotics Studio	no	no	no	yes	yes	no
Aware2.0	no	yes	no	yes	yes	yes
Hubo-Ach	yes	yes	yes	yes	yes	yes

Ubuntu Linux 10.04 with PREEMPT kernel. The labels αs/βr indicate a test run with α sending processes and β receiving processes.

3.3 Timing

To ensure that the Hubo-Ach controller is able to run at the desired control rates, timing experiments of each part of the controller was taken. All tests were done with a sample step size of 0.005 *sec*. Each of the following figures have the same X and Y scale. This is to give a visual representation of how much each portion of the cycle each part of Hubo-Ach takes up.

Fig. 3.5 shows the amount of time it takes to request and get the reference for the actuators. This reads the most recent reference off of the feedforward channel and uses that as the reference used in this cycle of Hubo-Ach. This time is measured to be 0.0010 ms with micro-second accuracy. The standard deviation is 0.0028.

Fig. 3.6 shows the amount of time it takes to complete all unread commands given by the user via the console. User commands are manual actions such as homing individual or all joints, resetting actuator errors, and reading error states. This time

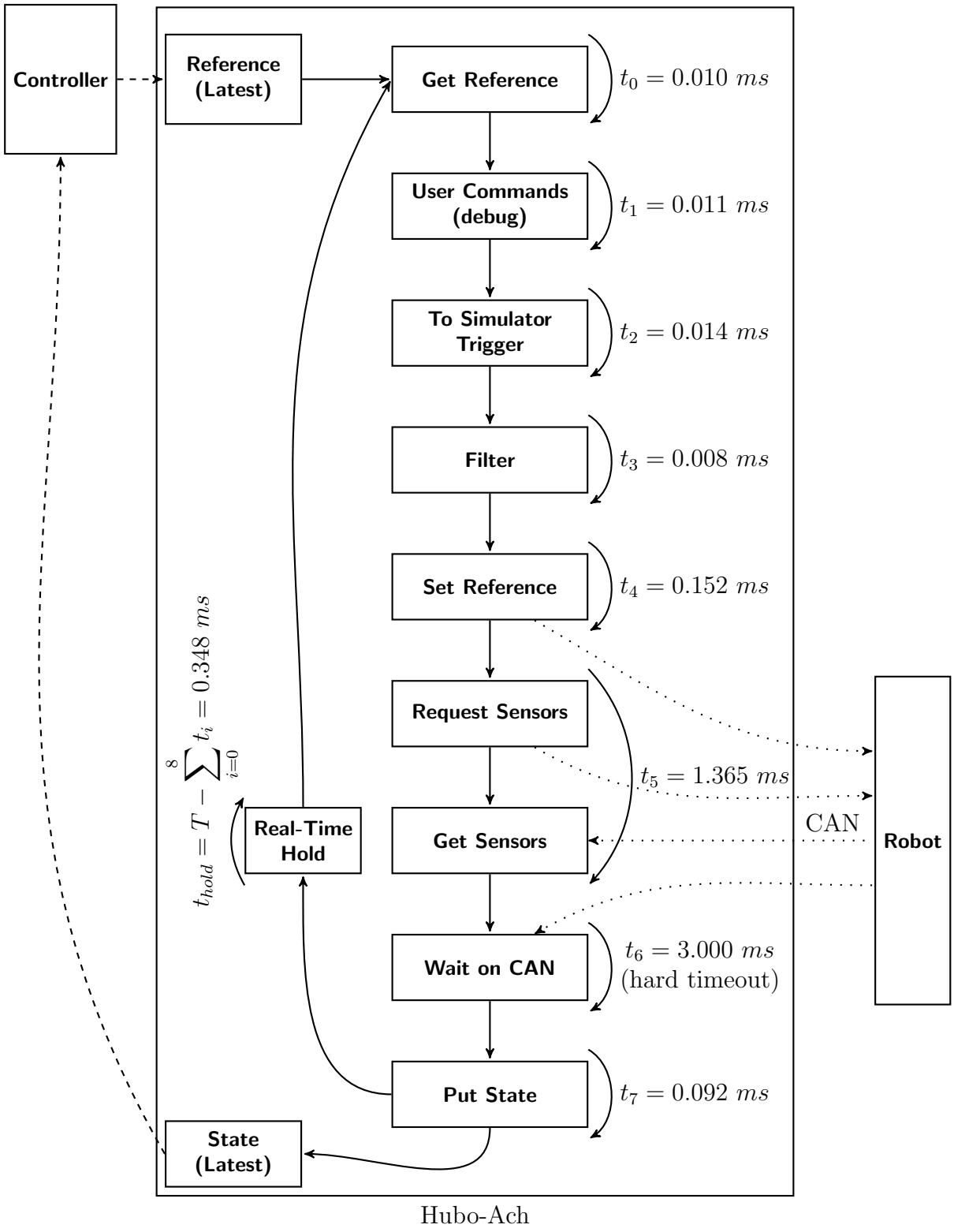


Figure 3.4: Timing diagram of Hubo-Ach. All times t_* denote measured times each block takes to complete. Tests were done on a 1.6Ghz Atom D525 Dual Core with 1GB DDR3 800Mhz memory running Ubuntu 12.04 LTS linux kernel 3.2.0-29 on a Hubo2+ utilizing a CAN bus running at 1Mbps baud. Average CPU usage is 7.6% using a total of 4Mb or memory.

Table 3.2: Inter Process Comunication Method Comparison

Inter-Process Comunication Method	Open Source	POSIX Complaint	Non Blocking	Multiple Senders and Receivers	Low Latency	Light Weight	Access Old Data
Streams	yes	yes	no	yes	no	yes	yes
Datagram Sockets	yes	yes	no	yes	no	yes	yes
POSIX Message Queues	yes	yes	no	yes	no	yes	yes
Shared Memory	yes	yes	yes	yes	yes	yes	no
AIO	yes	yes	yes	yes	yes	yes	yes
CORBA	yes	yes	yes	no	yes	yes	yes
ROS	yes	yes	no	yes	no	no	no
Data Distribution Service	yes	yes	yes	yes	yes	yes	yes
Ach	yes	yes	yes	yes	yes	yes	yes

is measured to be 0.011 ms with micro-second accuracy. The standard deviation is 0.0.0033.

Fig. 3.15 shows the amount of time it takes to send the external trigger. This external trigger tells a controller or simulator when the new reference's and commands have been read. In real-time mode the measured time delay is 0.0014 ms with micro-second accuracy and a standard deviation of 0.0035.

Fig. 3.8 shows the amount of time it takes to process the built in filter. This filter has multiple options:

- Direct reference mode where the filter acts as a reference pass through (Section 3.5.1).
- Low pass filter based on previous reference commands (Section 3.5.2).
- Low pass filter using feedback from the actual position of the joint (Section 3.5.4).

- Compliance amplification mode which artificially increases the compliance of the joint (Section 3.5.3) The measured time delay is 0.0080 ms with micro-second accuracy. The standard deviation is 0.0030 ms.

This gives the system the option of reducing the jerk on the high-gain position controlled actuators allowing for slower update rates on the reference channel. The *direct reference* mode allows a controller to have direct access to the commanded reference with no additional filtering.

Fig. 3.9 shows amount of time it takes to set the reference on the actuators via setting the data in the CAN bus buffer. The amount of time it takes for the for the references to be set to the actuators via the CAN is dependent on the baud rate of the CAN bus. Currently the baud rate is set to 1Mbps. CAN is the limiting factor in the loop rate. The table of the required bits to be sent via can is available in Table 3.3.

Fig. 3.10, 3.11, 3.12, 3.13 shows the amount of time it takes to request and get the state data from the actuator from over the CAN bus. This takes in total 1.365 ms plus an additional 3.0 ms for wait-on-CAN to ensure all queued messages in the CAN buffer are send and received.

Fig. 3.14 shows the amount of time it takes to set the state to the feedback channel. The measured delay is 0.092 ms with a standard deviation of 0.091.

Assuming no CAN delays Hubo-Ach can run at 1900 khz. With the current configuration of a 2 channel CAN bus it is restricted to below 237 hz. With a 4 channel CAN configuration it can be increased to 469 hz. With an 8 channel CAN configuration it can be increased to 1063 hz.

3.4 CPU Usage

The CPU usage was analized while the Hubo-Ach controller was being used in the following states:

Table 3.3: Hubo CAN packet data length and explanation

Field name	Length (bits)	Purpose	Pos CMD	Board Status (main)	Board Status (Neck and finger)	Encoder Pos (normal)	Encoder Pos (Neck)	Encoder Pos Finger (0)	Encoder Pos (Finger 1)	Current	FT	IMU
Start-of-frame	1	Denotes the start of frame transmission	1	1	1	1	1	1	1	1	1	1
Identifier	11	A (unique) identifier for the data which also represent the message priority	11	11	11	11	11	11	11	11	11	11
Remote transmission request (RTR)	1	Dominant (0) (see Remote Frame below)	1	1	1	1	1	1	1	1	1	1
Identifier extension bit (IDE)	1	Must be dominant (0)Optional	1	1	1	1	1	1	1	1	1	1
Reserved bit (r0)	1	Reserved bit (it must be set to dominant (0), but accepted as either dominant or recessive)	1	1	1	1	1	1	1	1	1	1
Data length code (DLC)*	4	Number of bytes of data (08 bytes)	4	4	4	4	4	4	4	4	4	4
Data field	064 (0-8 bytes)	Data to be transmitted (length in bytes dictated by DLC field)	48	64	40	64	48	48	32	64	64	64
CRC	15	Cyclic Redundancy Check	15	15	15	15	15	15	15	15	15	15
CRC delimiter	1	Must be recessive (1)	1	1	1	1	1	1	1	1	1	1
ACK slot	1	Transmitter sends recessive (1) and any receiver can assert a dominant (0)	1	1	1	1	1	1	1	1	1	1
ACK delimiter	1	Must be recessive (1)	1	1	1	1	1	1	1	1	1	1
End-of-frame (EOF)	7	Must be recessive (1)	7	7	7	7	7	7	7	7	7	7
		Total	92	108	84	108	92	92	76	108	108	108

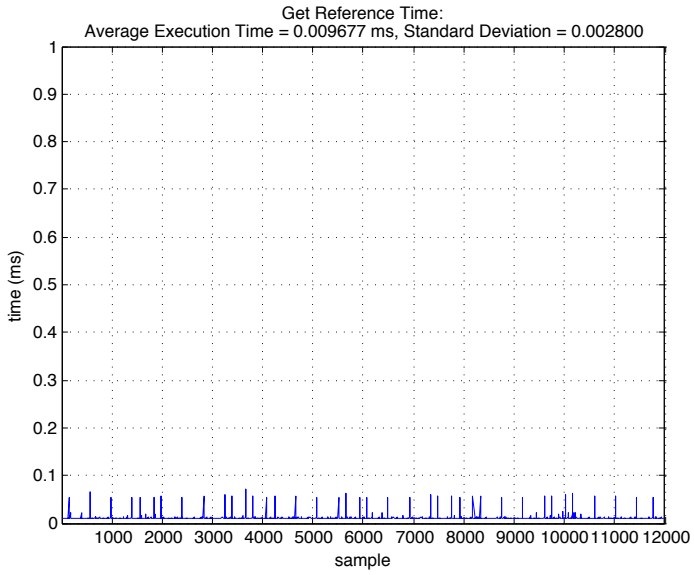


Figure 3.5: The amount of time it takes to request and get the reference for the actuators. In this case each sample has a time step of 0.005 *sec*

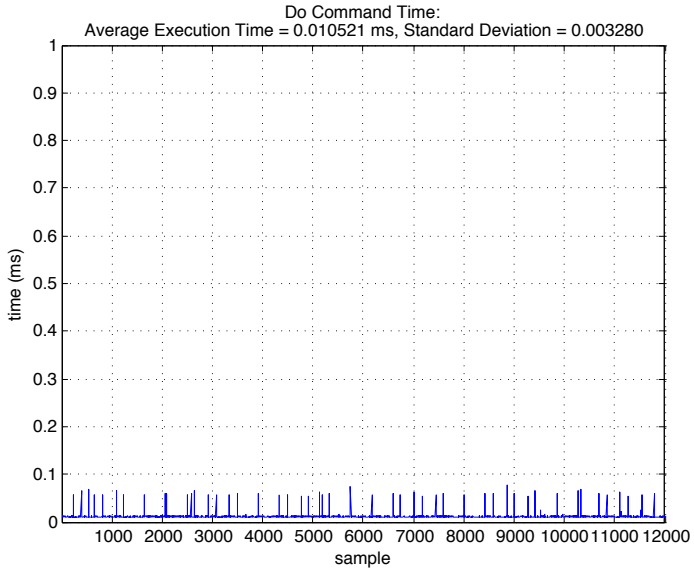


Figure 3.6: The amount of time it takes to complete all unread commands given by the user via the console. In this case each sample has a time step of 0.005 *sec*

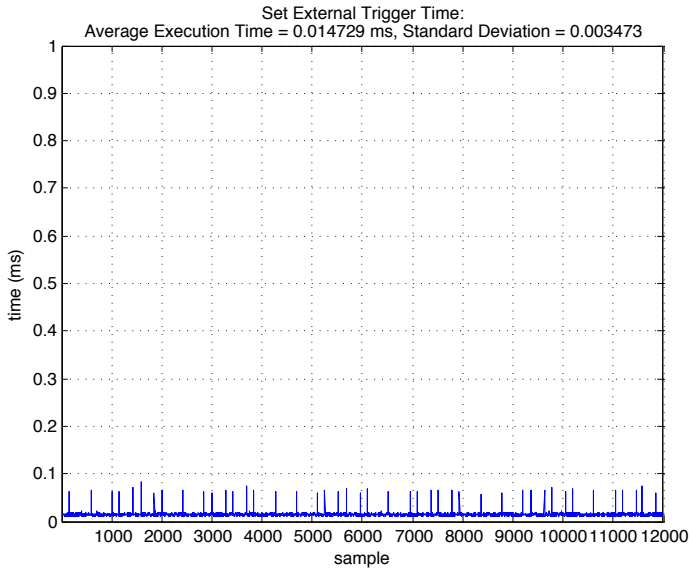


Figure 3.7: The amount of time it takes to send the external trigger. In this case each sample has a time step of 0.005 *sec*

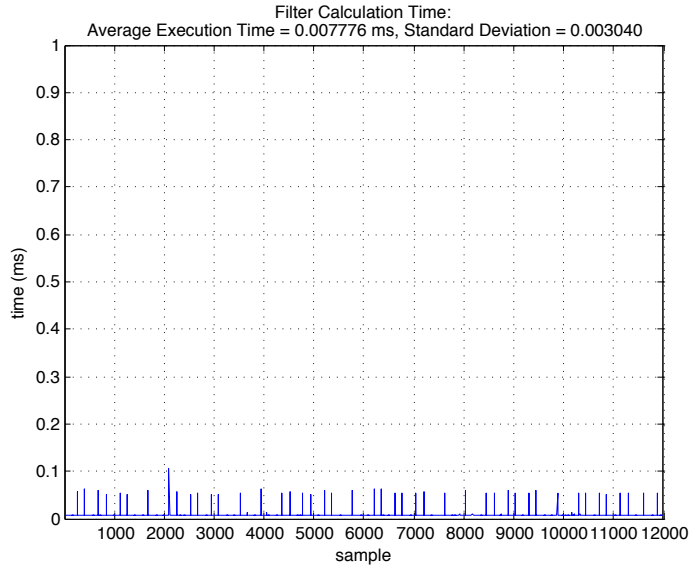


Figure 3.8: The amount of time it takes to process the built in filter. In this case each sample has a time step of 0.005 *sec*

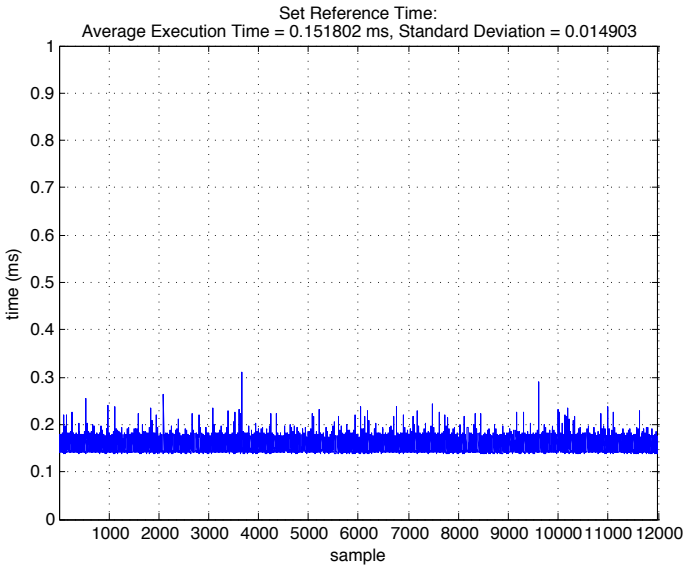


Figure 3.9: The amount of time it takes to set the reference on the actuators via setting the data in the CAN bus buffer. In this case each sample has a time step of 0.005 *sec*

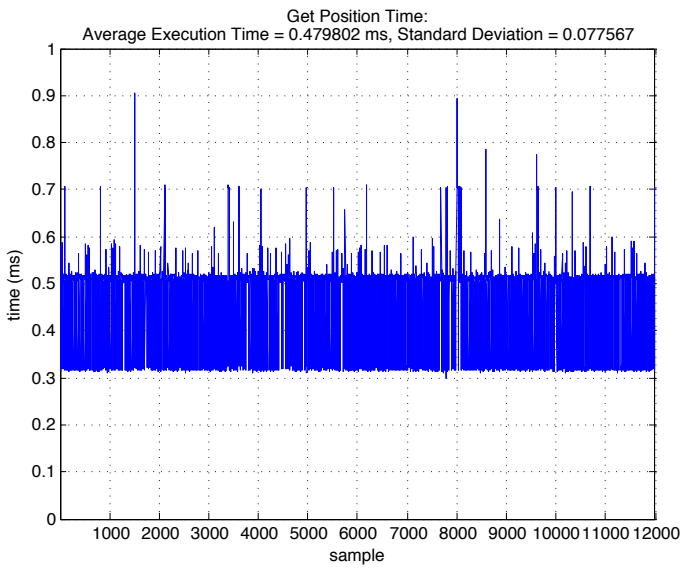


Figure 3.10: The amount of time it takes to request and get the actual position from the actuators. In this case each sample has a time step of 0.005 *sec*

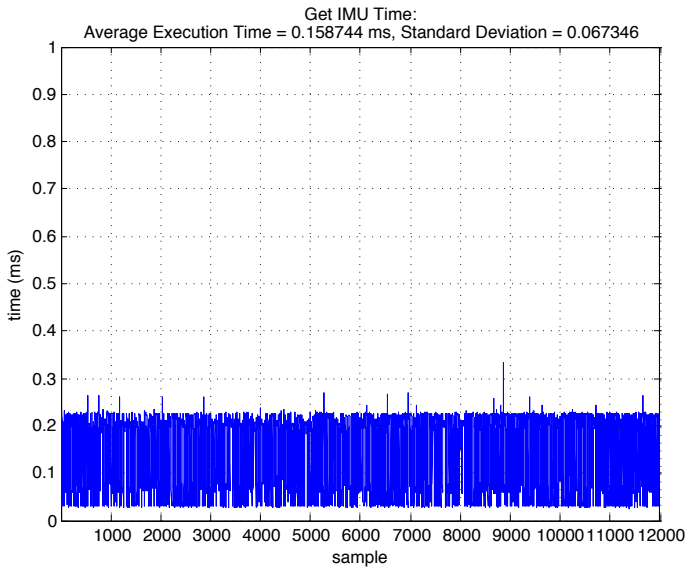


Figure 3.11: The amount of time it takes to request and get the IMU data. In this case each sample has a time step of 0.005 *sec*

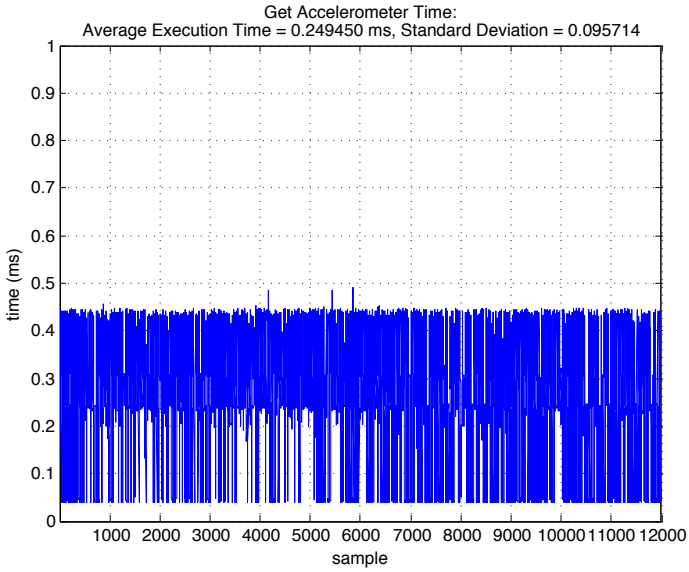


Figure 3.12: The amount of time it takes to request and get the accelerometers data. In this case each sample has a time step of 0.005 *sec*

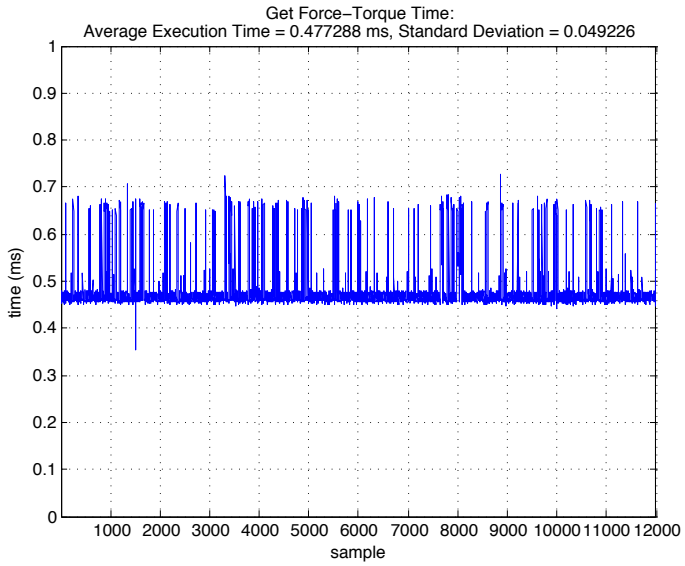


Figure 3.13: The amount of time it takes to request and get the force-torque sensors. In this case each sample has a time step of 0.005 *sec*

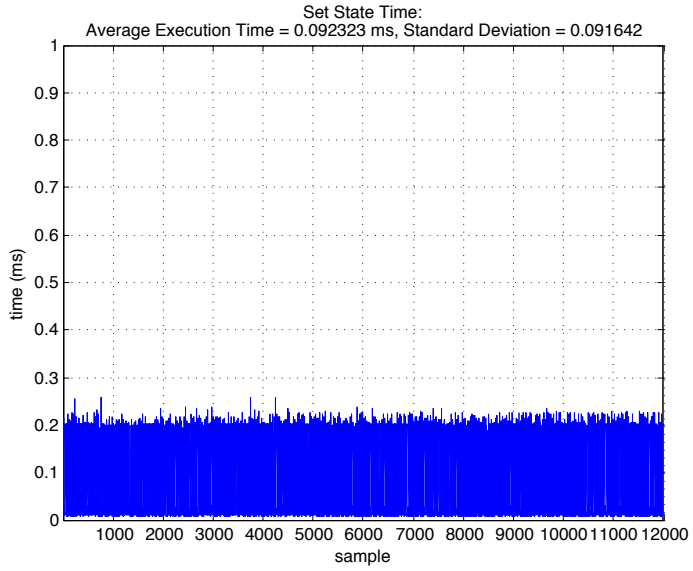


Figure 3.14: The amount of time it takes to set the state data on the feedback channel. In this case each sample has a time step of 0.005 *sec*

- Idle
- Reading the sensors
- Under open-loop control
- Under closed-loop control

Fig. 3.15 shows the result of this test. The results confirm that the CPU utilization stays within 0.3% when idle and under closed loop control. This means that the CPU utilization of Hubo-Ach is independent of the external control method. Thus it will not add more to the CPU load under complex control schemes then under simple ones. This makes it easy to model Hubo-Ach in when adding it to a CPU usage budget.

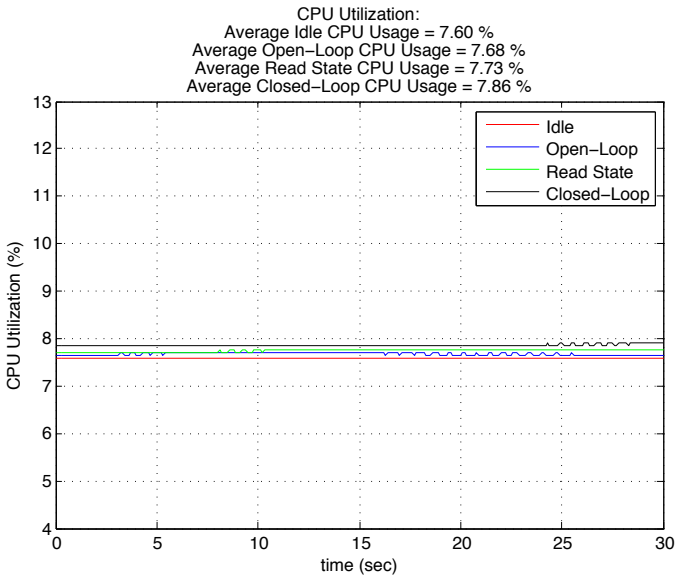


Figure 3.15: CPU utilization for the Hubo-Ach process when 1) idle, 2) under open-loop control, 3) reading the sensors, and 4) under closed-loop control. It is important to note that the cpu utilization stays within 0.3% when idle and under closed loop control. This means that the CPU utilization of Hubo-Ach is independent of the external control method. Thus it will not add more to the CPU load under complex control schemes then under simple ones.

Table 3.4: States being recorded for the single joint step response test

Signal	Symbol	Definition	Source	Units
FeedForward	θ_r	Desired reference on the Hubo-Ach FeedForward Channel	Hubo-Ach	*rad*
FeedForward	θ_c	Reference set to the actuator	Hubo-Ach	*rad*
Feedback	θ_a	Actual position of joint as measured from the encoders	JMC	*rad*

3.5 Verification Experiments

This section contains step by step verification examples showing the controller for high DOF complex system functions properly with the hubo system. All controllers are implemented using the multiple processes approach and includes all latencies found in Section 3.3.

3.5.1 Joint Space Step Response

This section shows the experimental and expected results of controlling a single joint via the Hubo-Ach system. In this example the right shoulder pitch (RSP) is given a step input from 0.0 *rad* to 0.4 *rad*. The reference position θ_r is begin recorded as well as the actuator setpoint θ_c and the actual position of the joint θ_a. These definitions are also available in Table 3.4

Fig. 3.16 shows the results when a step input is applied and Hubo-Ach is in *HUBO_REF_MODE_REF* also know as pass-through mode. This sets the what the desired reference on the **FeedForward** Hubo-Ach channel to the actuator's reference, i.e.:

$$\theta_c(N) = \theta_r(N) \tag{3.1}$$

From the results of Fig. 3.16 a 2^{nd} order model $G(s)$ of the joint can be made.

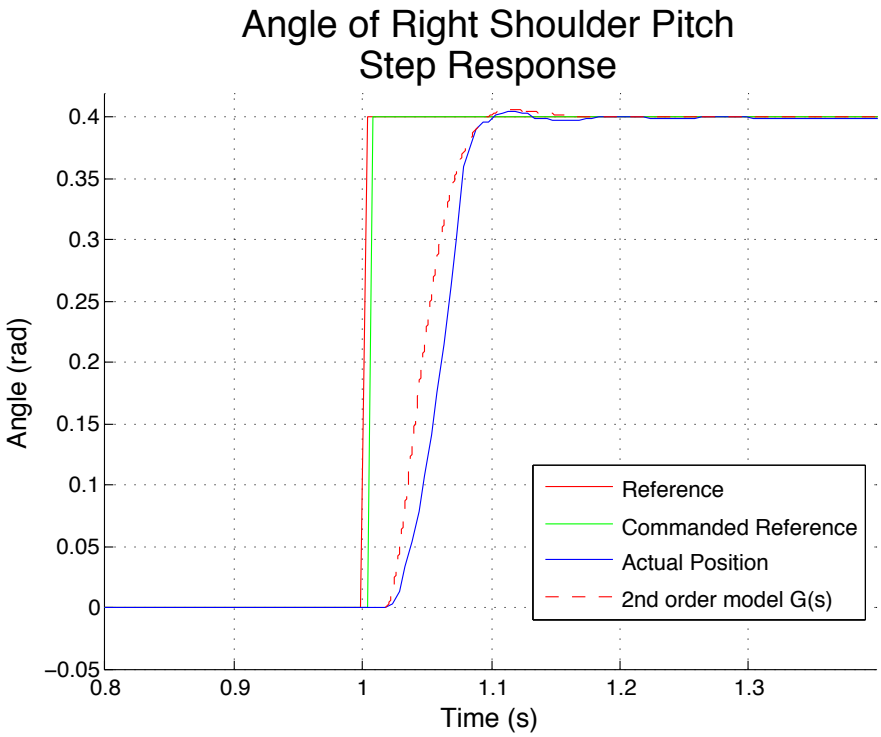


Figure 3.16: The commanded reference plotted against the actual reference recorded via Hubo-Ach and ground truth via CAN analyzing utilities. In this plot the commanded reference is not automatically filtered by Hubo-Ach. The commanded joint is the right shoulder pitch. The model of the joint $G(s)$ is also plotted. The resulting bandwidth is 45.79 $\frac{rad}{sec}$ or 7.29 hz.

$$G(s) = \frac{1120}{s^2 + 85s + 2800} \tag{3.2}$$

From the bandwidth can be determined to be 45.79 $\frac{rad}{sec}$ or 7.29 hz. The control loop is over an order of magnitude greater then the actuator's bandwidth thus the control rate is acceptable.

Have knowledge this specific system, it is known that the acceleration is artificially limited in the controller when starting from rest on this model of Hubo motor driver. Thus the system is non-linear. If the initial rate limiting is not taken into account and a focus is put to matching the slope of the middle The new model $G^*(s)$ is:

$$G^*(s) = \frac{2200}{s^2 + 115s + 5500} \tag{3.3}$$

It now has a bandwidth of 66.98 $\frac{rad}{sec}$ or 10.66 hz. This is still well within the Hubo-Ach bandwidth. The step response of $G^*(s)$ can be found in Fig. 3.17.

Fig. 3.18 shows the block diagram of the control setup.

As seen in Fig 3.16 θ_c tracks θ_r perfectly. As expected θ_a lags by a minimum of 1 time step T. This is the time it takes between sending θ_c to the actuator over the CAN bus plus the time it takes in receiving the feedback from the encoder of the motor over CAN. The remainder of the lag is due to the rise time of the actuator. This is different for each joint. Because all major joints are high-gain PID the rise-time and overshoot is very small which makes the robot very stiff. The total lag between commanding the joint on the **FeedForward** channel and the response of the actuator is:

$$t_{lag} = t_{filter} + t_{rise} \tag{3.4}$$

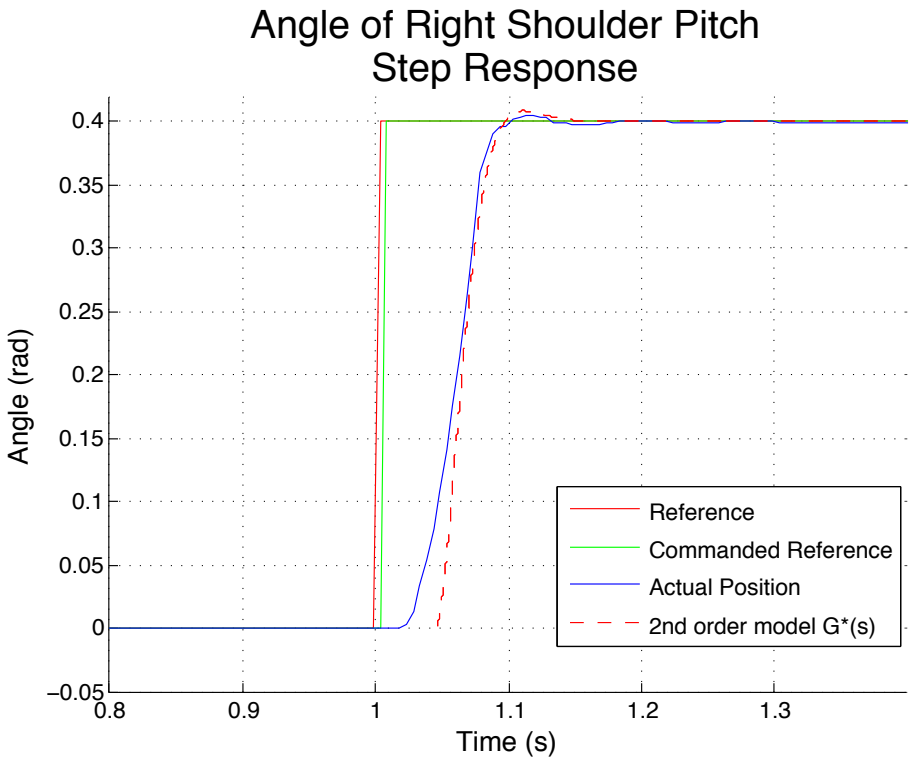


Figure 3.17: The commanded reference plotted against the actual reference recorded via Hubo-Ach and ground truth via CAN analyzing utilities. In this plot the commanded reference is not automatically filtered by Hubo-Ach. The commanded joint is the right shoulder pitch. The model of the joint $G^*(s)$ is also plotted. The resulting bandwidth is 66.98 $\frac{rad}{sec}$ or 10.66 hz.

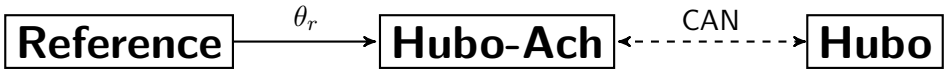


Figure 3.18: Reference θ_r being applied to Hubo via Hubo-Ach. θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle.

3.5.2 Joint Space Step Response with Position Filtering

Giving a step input to a high-gain PID position controlled actuator can cause an over current fault, burn out motor drivers, strip gears due to the *jerk* etc. To reduce this effect Hubo-Ach has multiple modes of on-board filtering. These modes are:

- Reference Input Filtering
- Compliance Amplification

This section talks about *reference input filtering* as a method to apply a step input each joint in joint space and limit the jerk. It is important to note that the obvious answer is to reduce the PID gains to make the robot *more complaint* however the goal of this work is to make a fully functional system that does not require modification of the robot. In this case the PID gains are set by the motor drivers and that is considered to be a part of the robot. In future firmware updates of the motor drivers we will have the ability to change PID gains on the fly.

reference input filtering uses the history of the previous θ_c sent to the given actuator. The current commanded actuator position $\theta_c(N)$ is given by:

$$\theta_c(N) = \frac{\theta_c(N-1) \cdot (L-1) + \theta_r(N)}{L} \tag{3.5}$$

Where L is an integer that represents the length of the filter and $L \geq 1$. If $L = 1$ then Equation 3.5 becomes Equation 3.1.

Fig. 3.20 shows the commanded reference plotted again the actual reference using the filtered mode defined in Equation 3.5. Fig. 3.21 shows the θ_r plotted against θ_c and θ_a for different values of L. It is easy to see that as L increases the t_{rise} also increases and the *jerk* is reduced.

This method is a feed-forward method that assumes that the position you set the actuator to is the actual position of the actuator.

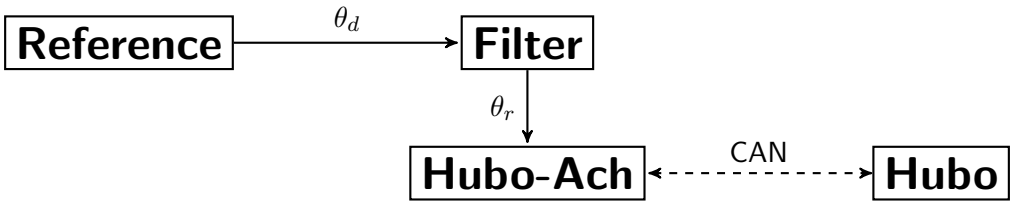


Figure 3.19: Desired reference θ_d being filtered before applied to Hubo via Hubo-Ach. θ_d is sent through a filter that reduces the *jerk* on the actuator then the new reference θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle.

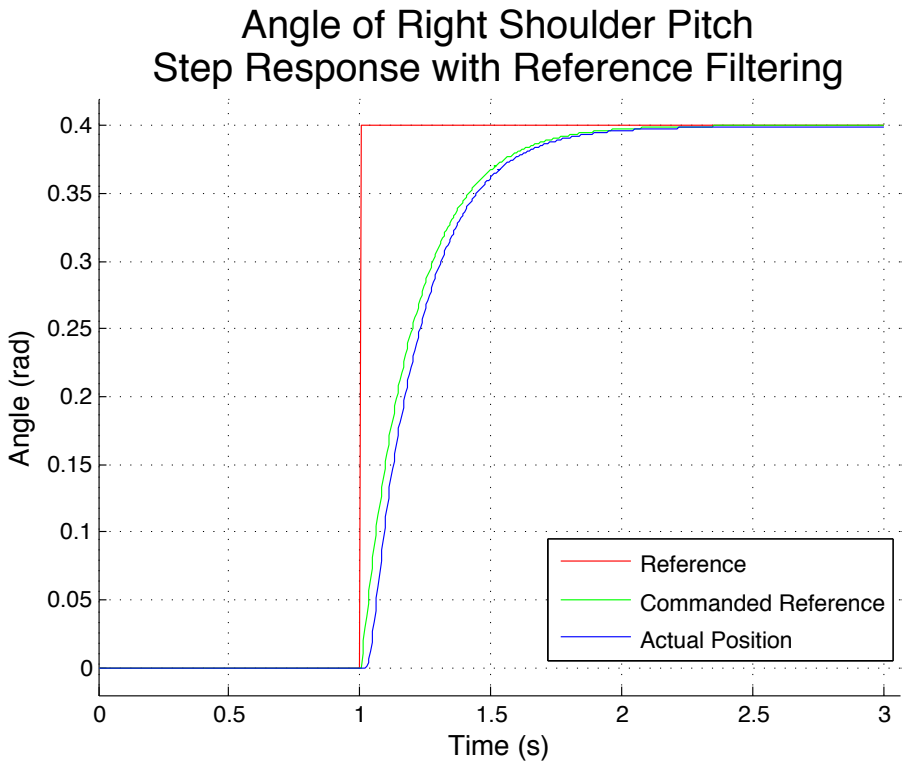


Figure 3.20: The commanded reference plotted against the actual reference recorded In this plot the commanded reference is automatically filtered by Hubo-Ach.

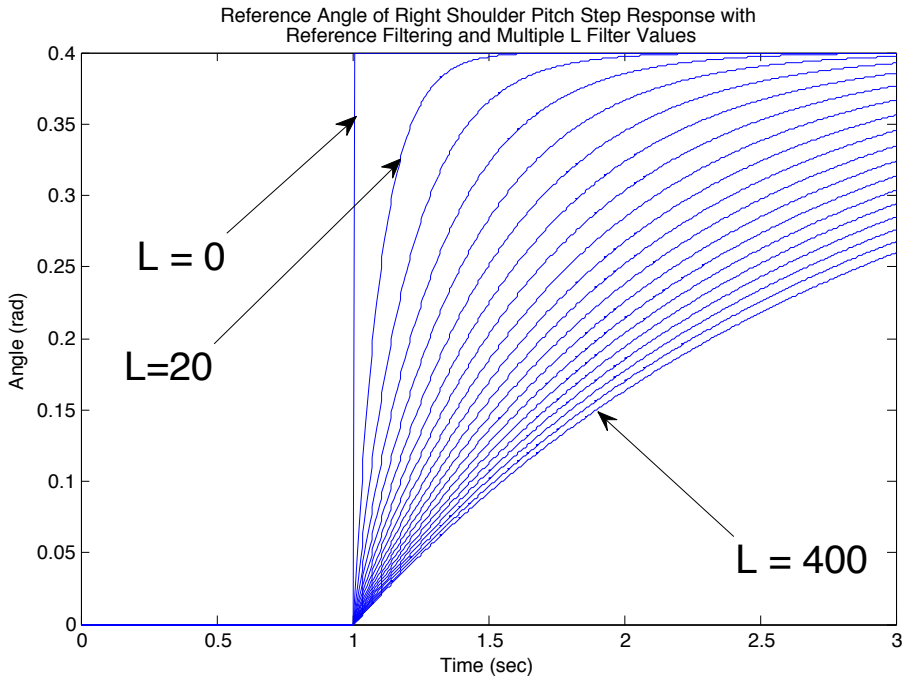


Figure 3.21: θ_r plotted against θ_c and θ_a recorded via Hubo-Ach with values for L ranging from 0 to 400 in increments of 20.

3.5.3 Compliance Amplification

Compliance amplification takes advandage of the internal compliance of the joints and amplifies that by feeding back the PID error θ_e. Like the Equation 3.1 we have no past information about the set reference and we have only the compiliance given by the joints. If we think about θ_e and what effects it we can use it to add compliance to our system. It is important to note that because the Hubo is a high-gain PID position controlled device with an intergral gain K_i set to zero the steady state error of the joint (the PID error θ_e) is proportional to the moment applied to the joint. If we combine the reference θ_r and θ_e multiplied by a compliance gain K_c we are able to add/amplify the compliance to the system.

$$\theta_c(N) = K_c\theta_e(N) + \theta_r(N) \tag{3.6}$$

It is important to note that $K_c \leq 1$ or the system will go unstable. If $K_c = 1$ then we have

$$\theta_c(N) = \theta_a(N) \tag{3.7}$$

3.5.4 Joint Space Step Response with Feedback Filtering

Feedback filtering allows us to removes the requirement that we know the joint's current position. Similar to Equation 3.5 this method sets θ_c based on a filter length L and the current desired value θ_r. However instead of assuming that we know all past θ_r we use the actual position θ_a. This method add compliance in a similar way to that of Section 3.5.3.

$$\theta_c(N) = \frac{\theta_a(N) \cdot (L-1) + \theta_r(N)}{L} \tag{3.8}$$

This causes three major effects:

Effect 1: The movement of the joint is guaranteed to be filtered even if the previous reference is unknown.

Effect 2: The steady state error of the feedback filtering method $\theta_e^{fbfilter}$ is greater than that of the PID error θ_e in the direction of the moment acting on the joint.

$$\theta_e^{fbfilter} > \theta_e \tag{3.9}$$

Effect 3: The joint's compliance has increased due to the effect of the moment applied to the joint has on the steady state error.

Fig. 3.23 shows θ_r plotted against θ_c and θ_a. θ_a not only lags behind θ_c but it also has a greater steady state error. Fig. 3.24 shows how the steady state error $\theta_e^{fbfilter}$ increases with an applied moment. This is where we get our compliance.

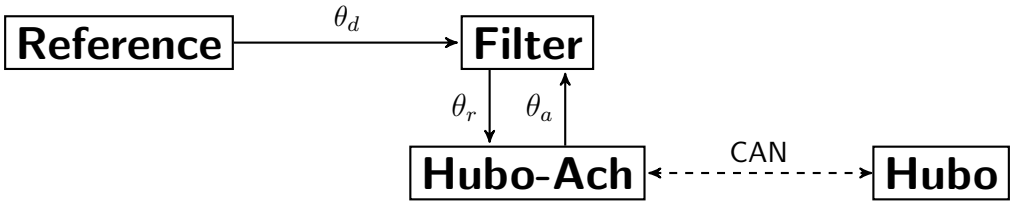


Figure 3.22: Desired reference θ_d being filtered before applied to Hubo via Hubo-Ach. θ_d is sent through a filter that reduces the *jerk* on the actuator by using Equation 3.8. The new reference θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle. This method adds compliance to the system

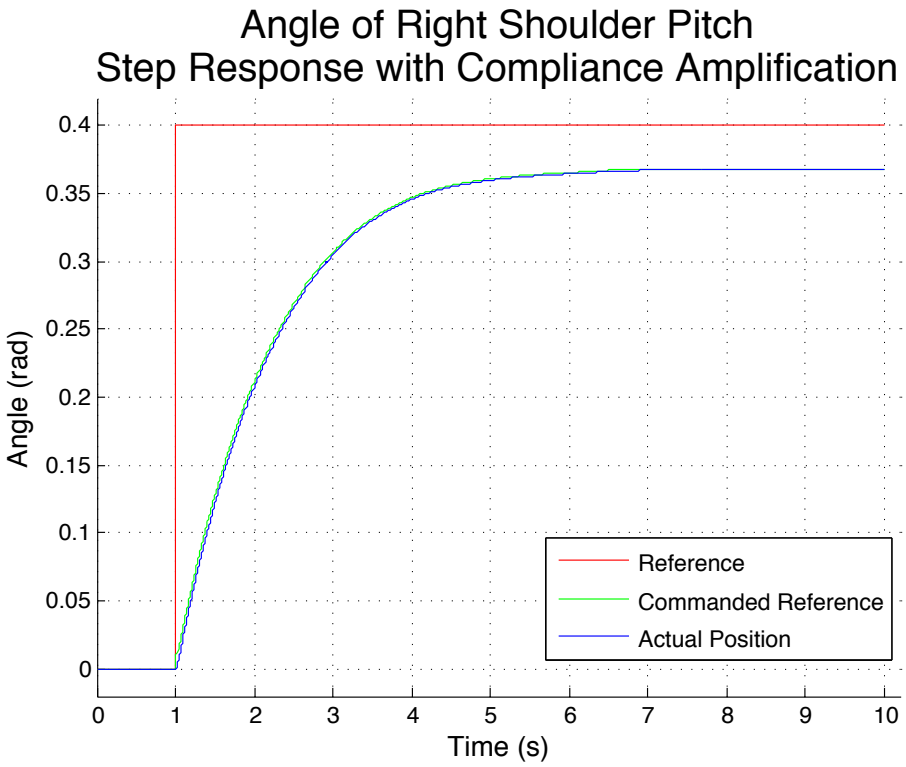


Figure 3.23: θ_r plotted against θ_c and θ_a recorded via Hubo-Ach using the feedback filtering method.

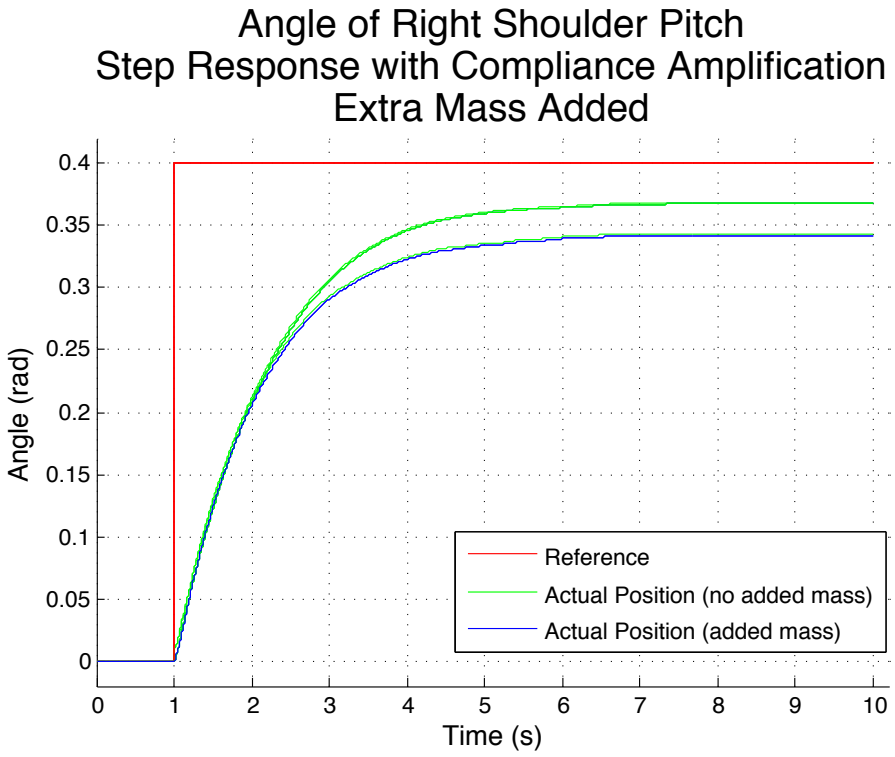


Figure 3.24: θ_r plotted against θ_c and θ_a recorded via Hubo-Ach using the feedback filtering method with different moments applied to the joint. You will note that as the moment increases so does $\theta_e^{fbfilter}$.

3.6 Kinematics

Kinematic planning is a key focus of the Hubo-Ach controller. This section provides two published examples of the Hubo-Ach controller being used for inverse kinematics and control. Section 3.6.1 shows the work of Lofaro et. al. [3] using Constrained Bi-Directional Rapidly-exploring Random Tree (CBiRRT) to provide a statically stable joint space trajectory allowing the robot to turn a valve. Section 3.7 shows the work of Lofaro et. al. [2] using traditional 6 DOF forward and inverse kinematic techniques to provide an analytical IK solution to each of the 6 DOF end effectors. The additional use of on-line trapezoidal velocity profiling methods in Section L.0.5 allow for the creation of a real-time IK controller based in Hubo-Ach.

3.6.1 Valve Turning

This section presents progress towards performing valve turning task set by the DARPA Robotics Challenge (DRC) Event #7[42]. This work is published verification of the Hubo-Ach system with details in Lofaro et. al. [3]. The task requires that a robot locate, approach, grasp, and turn an industrial valve with two hands. A core constraint for the DRC is that communications with the robot are limited, making conventional tele-operation is infeasible. Thus, the valve-turning task requires a straightforward way for a user to command the robot to perform complex actions.

Fig. 3.25 shows the block diagram of Hubo-Ach being used for the DRC event #7, valve turning. The process to get the Hubo to turn a valve consists of loading a model of the Hubo and the valve into the simulator. OpenRAVE is used as the simulator using the OpenHubo model of Hubo. The trajectory planner uses CBiRRT to plan a collision free statically stable joint space path. Once the planning is completed the resulting joint space trajectory it is sent through a low-pass filter then sent to the Hubo. Fig. 3.26 shows the Hubo turning a valve using this method.

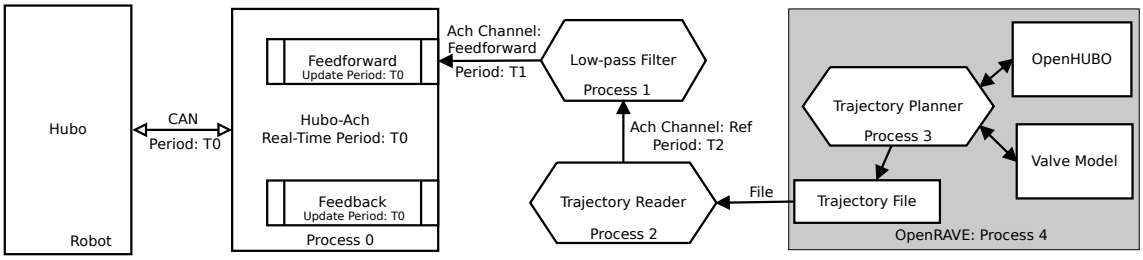


Figure 3.25: Block diagram of Hubo-Ach being used for the DRC event #7, valve turning. The process to get the Hubo to turn a valve consists of loading a model of the Hubo and the valve into the simulator. OpenRAVE is used as the simulator using the OpenHubo model of Hubo. The trajectory planner uses CBiRRT to plan a collision free statically stable joint space path. Once the planning is completed the resulting joint space trajectory it is sent through a low-pass filter then sent to the Hubo.

Planning

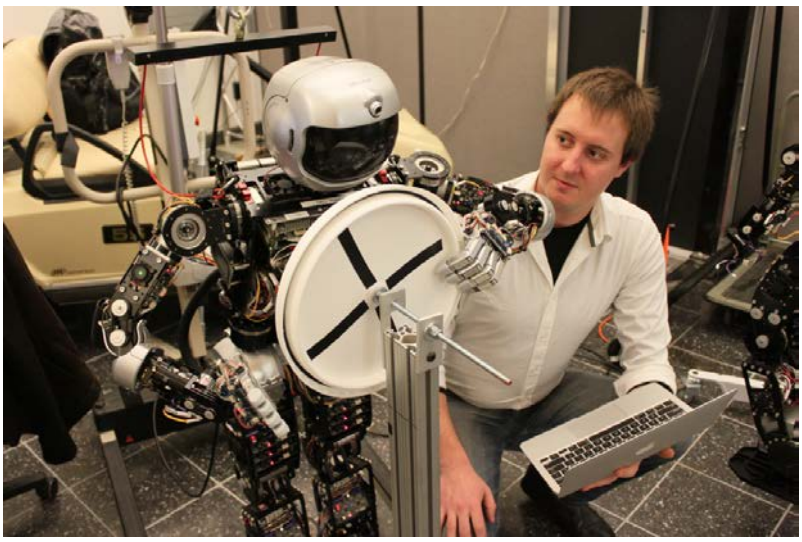
The planning package plans trajectories for high degree of freedom robots so that they can perform object manipulation. The initial configuration of the robot is critical to manipulation because the robot must be able to:

- reach and manipulate the object for the entirety of the desired trajectory,
- maintain balance during execution,
- avoid self-collisions and collisions with the environment

Motion planning is provided by the Constrained Bi-Directional Rapidly-exploring Random Tree (CBiRRT), an efficient and probabilistically complete manipulation planning suite. CBiRRT consists of three main components: constraint representation, constraint-satisfaction, and a general planning algorithm. For full details of CBiRRT and its implementation, see Berenson et. al.[43].

Experiment

Our preliminary experiments with the Hubo were centered around validating our method of motion planning for the robot and evaluating the robots capabilities in relation to the requirements of our DRC task (turning the valve). These tests were performed on the Hubo2+ at MIT and Drexel University, housed in the lab of Professor Russ Tedrake and Paul Oh respectively. Our experiments conrmed that the planning system enabled control of the Hubo and that the Hubo was physically capable of turning the valve. A full description of our methods and experiment can be found in [3].



Video: http://danlofaro.com/phd/valve/

Figure 3.26: Hubo (left) turning a valve via Hubo-Ach alongside Daniel M. Lofaro (right). Valve turning developed in conjunction with Dmitry Berenson at WPI for the DARPA Robotics Challenge.

Table 3.5: DenavitHartenberg for Hubo2+ upper body (arms) in standard format

Link	Length (m)
l_{A1}	0.215
l_{A2}	0.179
l_{A3}	0.182
l_{A4}	0.121
l_E	0.100

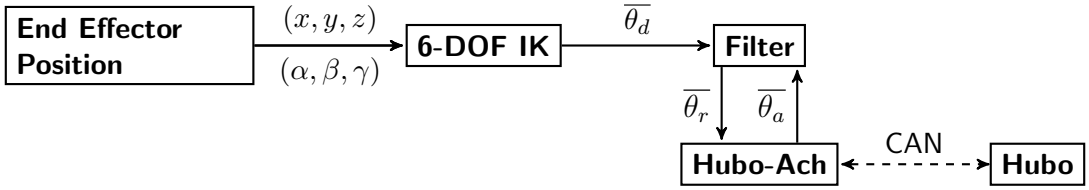


Figure 3.27: Desired reference θ_d being filtered before applied to Hubo via Hubo-Ach. θ_d is sent through a filter that reduces the *jerk* on the actuator by using Equation 3.8. The new reference θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle. This method adds compliance to the system

3.7 Six Degree of Freedom Inverse Kinematic Implementation Example

This section shows how we calculate the inverse kinematics (IK) for the Hubo's right arm and how we use that calculation in conjunction with Section 3.5. The result is the ability to command the end effector (EEF)

In order to control the Hubo's upper body manipulators in work space as opposed to joint space both forward and inverse kinematics are required, (FK) and (IK) respectively. In order to find a proper solution the joint limits, singularities and feasible workspace (no-self collisions) must be accounted for.

The kinematic structure of the right and left arm of the Hubo are identical with the caveat that the work space offset is mirrored over the z-axis. This means that they have the same DenavitHartenberg (DH) parameters.

Table 3.6: DenavitHartenberg Parameters (continued) for Hubo2+ upper body (arms) in standard format

i (frame)	θ_i (rad)	α_i (rad)	a_i (m)	d_i (m)
1	$\theta_1 + \frac{\pi}{2}$	$\frac{\pi}{2}$	0	0
2	$\theta_2 - \frac{\pi}{2}$	$\frac{\pi}{2}$	0	0
3	$\theta_3 + \frac{\pi}{2}$	$-\frac{\pi}{2}$	0	$-l_{A2}$
4	θ_4	$\frac{\pi}{2}$	0	0
5	θ_5	$\frac{\pi}{2}$	0	$-l_{A3}$
6	$\theta_6 + \frac{\pi}{2}$	0	l_{A4}	0

3.7.1 Froward Kinematics

The transform between joint adjacent joints is represented by the transform:

$$T_{i-1}^{i} = \begin{bmatrix} cos(\theta_i) & -sin(\theta_i)cos(\alpha_i) & sin(\theta_i)sin(\alpha_i) & a_i cos(\theta_i) \\ sin(\theta_i) & cos(\theta_i)cos(\alpha_i) & -cos(\theta_i)sin(\alpha_i) & a_i sin(\theta_i) \\ 0 & sin(\alpha_i) & cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.10}$$

Where θ_i, α_i and d_i are shown in Fig. 3.28. The coordinate frame of the Hubo2+ used for both the forward and inverse kinematics are defined in Fig. 3.29 and Fig. 3.30. The DH parameters for the arm for the transform T_i^{i-1} is found in Table 3.6.

In order to calculate for full FK transform T_N^E, where E represents the end-effector and N is the neck (robot origin), we must first calculate:

$$T_0^6 = \prod_{i=1}^{6} T_{i-1}^{i} = T_0^1 T_1^2 T_2^3 T_3^4 T_4^5 T_5^6 \tag{3.11}$$

In order to procure the transform T_N^E we must pre-multiply T_0^6 by the transform T_N^0 and post-multiply it by the transform T_6^E. This results in transform T_N^E:

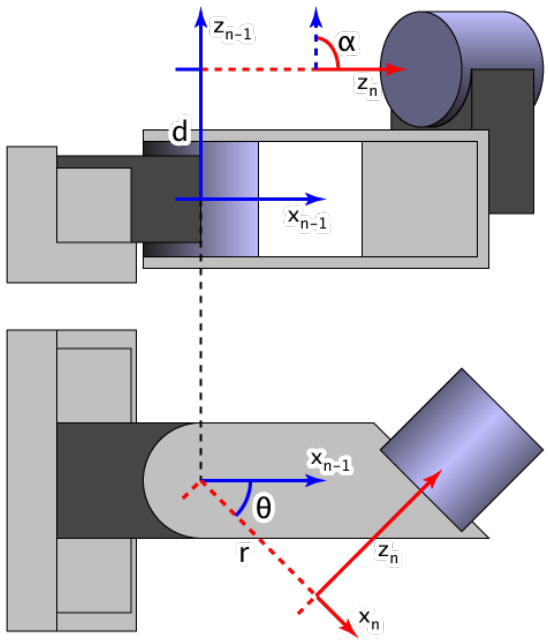


Figure 3.28: Denavit-Hartenberg diagram showing that axis of rotations and displacements to create the transform in Equation 3.10. α is the angle between the axis of rotation of joint n and $n-1$ about the of n. θ is the angle between the axis of rotation of joint n and $n-1$ about the axis perpendicular to the axis about n.

Image Credit:

http://en.wikipedia.org/wiki/File:Sample_Denavit-Hartenberg_Diagram.png

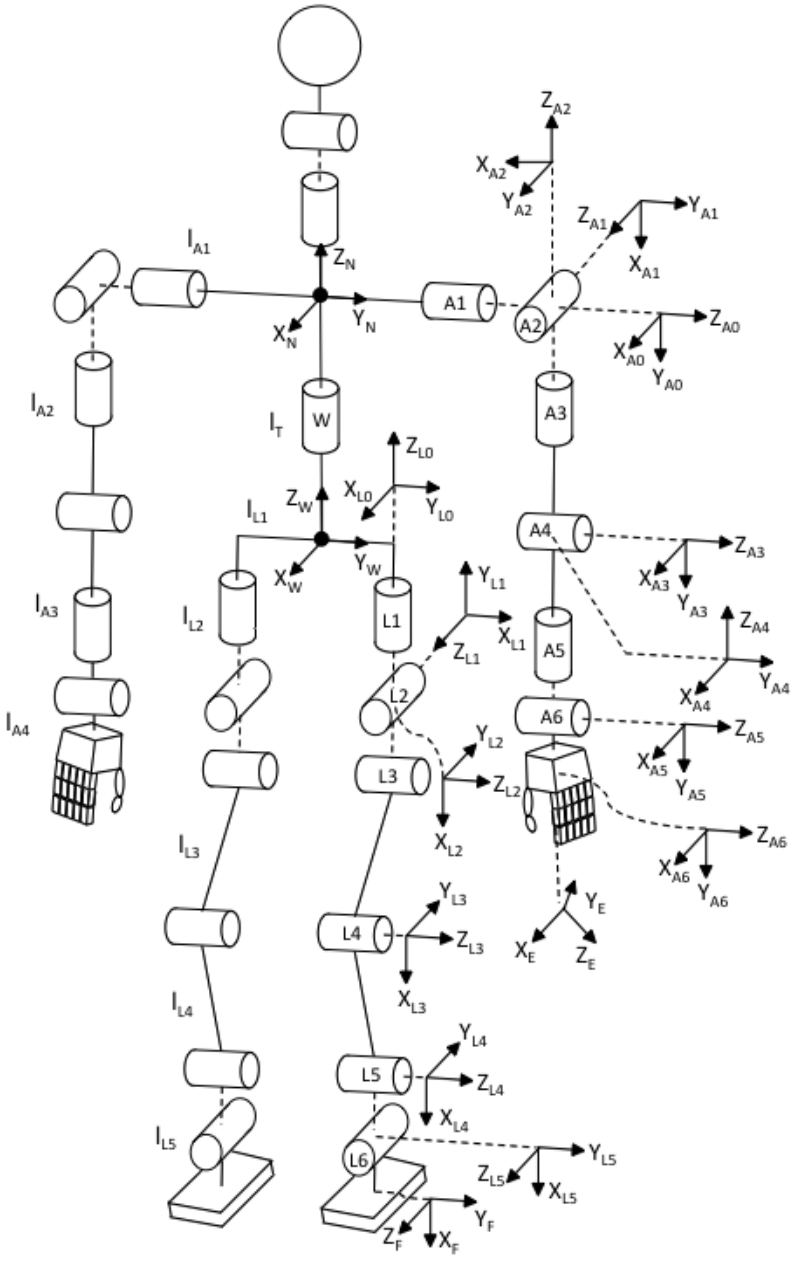


Figure 3.29: Hubo2+ coordinate frame for use with the forward and inverse kinematic example. These coordinate frames are defined specifically for the IK and FK examples and are the same frame as in[44]

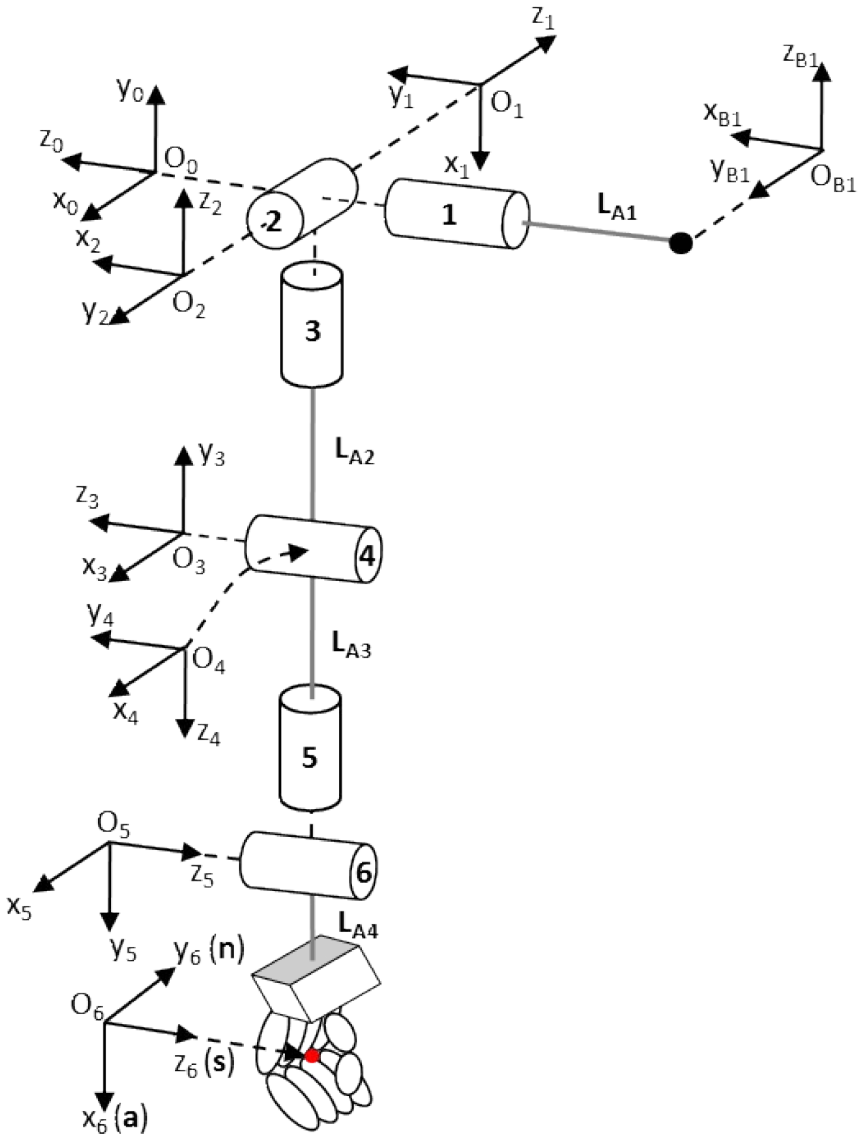


Figure 3.30: Hubo2+ coordinate frame for right arm. Uses with the forward and inverse kinematic example. These coordinate frames are defined specifically for the IK and FK examples and are the same frame as in[44]

$$T_N^E = T_N^0 T_0^6 T_6^E \tag{3.12}$$

Where T_N^0 is

$$T_N^0 = \begin{bmatrix} 0 & 0 & 1 & l_{A1} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.13}$$

Now with a given set of joint space angles we can find the end-effectors position in reference to the robot origin, the neck.

3.7.2 Inverse Kinematics

The next step is to find the inverse kinematic (IK) solution for the right arm. Inherently this problem has multiple solutions. When solving the IK Pieper[45] states that a closed-form solution does exist if:

- Three consecutive joints axes of the manipulator are parallel to one another

OR

- Three consecutive joints intercect at a single point

The kinematic structure in Fig 2.2 and Fig 3.29 shows that the Hubo2+ platform does have a three joints that intersect the same point in the shoulders and in the hips. Thus a closed-form solution exists for both arms and both legs.

The transform T_0^6 in Equation 3.11 is needed to solve the IK problem for the shoulder. It is important to note that T_0^6 is in the form of

$$T_0^6 = \begin{bmatrix} \overline{x_6} & \overline{y_6} & \overline{z_6} & \overline{p_6} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

Where $\overline{x_6}$, $\overline{y_6}$ and $\overline{z_6}$ are $[3x1]$ unit vectors along the principle axes of the end-effectors coordinate frame i, see Fig. 3.29. Position vector $\overline{p_6}$ describes the hand about joint $A1$ (shoulder). The arm can be vied in different frames. If we look at the arm in reference to the end-effector's frame. The reverse transform is defined as (T_0^6)‘

$$(T_0^6)' = T_6^0 = (T_0^6)^{-1} = \begin{bmatrix} \overline{x_6} & \overline{y_6} & \overline{z_6} & \overline{p_6} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} \tag{3.15}$$

The following method is based on the work done by our partner Park et. al.[44]. The general link translation matrix T_{i-1}^i relates the i^{th} coordinate frame to the $(i-1)^{th}$ coordinate frame. In addition we can extend Equation 3.14 to

$$T_0^6 = \begin{bmatrix} \overline{x_6} & \overline{y_6} & \overline{z_6} & \overline{p_6} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \overline{n} & \overline{s} & \overline{a} & \overline{p} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.16}$$

Where $[\overline{n}, \overline{s}, \overline{a}, \overline{p}]$ represents the normal vector, the sliding vector, the approach vector and the position vector of the end effector respectively[46]. We can now state that

$$(T_0^6)' = T_6^0 = (T_0^6)^{-1} = \begin{bmatrix} \overline{x_6} & \overline{y_6} & \overline{z_6} & \overline{p_6} \\ 0 & 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} \overline{n}' & \overline{s}' & \overline{a}' & \overline{p}' \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.17}$$

We can now use the reverse method to solve for the joint angles as in [46] and derived in the tech report[47]. The first three lower joint angles of A_4, A_5 and A_6 are solved for. Subsequently the upper joint angles of A_1, A_2 and A_3 are solved.

Using inverse transform methods[48] we can modify Equation 3.11 to

$$T_6^0 = (T_0^6)^{-1} = \prod_{i=6}^{1} T_i^{i-1} = T_6^5 T_5^4 T_4^3 T_3^2 T_2^1 T_1^0 \tag{3.18}$$

Then we equate Equation 3.16 to Equation 3.18

$$T_6^5 T_5^4 T_4^3 T_3^2 T_2^1 T_1^0 = \begin{bmatrix} \overline{n}' & \overline{s}' & \overline{a}' & \overline{p}' \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.19}$$

Then move T_6^5 to the other side of the equation

$$T_5^4 T_4^3 T_3^2 T_2^1 T_1^0 = T_5^6 \begin{bmatrix} \overline{n}' & \overline{s}' & \overline{a}' & \overline{p}' \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.20}$$

For simplicity we will represent Equation 3.20 as G_L and G_R standing for *right* and *left* side.

$$G_L = T_5^6 \begin{bmatrix} \overline{n}' & \overline{s}' & \overline{a}' & \overline{p}' \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.21}$$

$$G_R = T_5^4 T_4^3 T_3^2 T_2^1 T_1^0 \tag{3.22}$$

Expanding gives us

$$G_L = \begin{bmatrix} g_{11} & g_{12} & g_{13} & cos(\theta_6)(p'_x + l_{A_4}) - sin(\theta_6)p'_y \\ g_{21} & g_{22} & g_{23} & sin(\theta_6)(p'_x + l_{A_4}) - cos(\theta_6)p'_y \\ g_{31} & g_{32} & g_{33} & p'_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.23}$$

and

$$G_R = \begin{bmatrix} g_{11} & g_{12} & g_{13} & sin(\theta_4)cos(\theta_5)l_{A_2} \\ g_{21} & g_{22} & g_{23} & -cos(\theta_6)l_{A_2} - l_{A3} \\ g_{31} & g_{32} & g_{33} & sin(\theta_4)sin(\theta_5)l_{A_2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.24}$$

We can then equate elements $(1,4)$, $(2,4)$ and $(3,4)$ of G_L and G_R. This gives us

$$cos(\theta_6)(p'_x + l_{A_4}) - sin(\theta_6)p'_y = sin(\theta_4)cos(\theta_5)l_{A_2} \tag{3.25}$$

$$sin(\theta_6)(p'_x + l_{A_4}) - cos(\theta_6)p'_y = -cos(\theta_6)l_{A_2} - l_{A3} \tag{3.26}$$

$$p'_z = sin(\theta_4)sin(\theta_5)l_{A_2} \tag{3.27}$$

Based on the desired task space location we let

$$p'_x + l_{A_4} = r \cdot cos(\phi) \tag{3.28}$$

and

$$p'_y = r \cdot sin(\phi) \tag{3.29}$$

where

$$r = sqrt(p'_x + l_{A_4})^2 + (p'_y)^2 \tag{3.30}$$

and

$$\phi = atan2(p'_y, p'_x + l_{A_4}) \tag{3.31}$$

Note: *atan*2() represents the the *atan* method that gathers the information of the signs of the inputs in order to put the returned value in the appropriate quadrant.

Combining Equation (3.25), (3.26) and (3.27) with Equation (3.28) and (3.29) we get

$$r \cdot cos(\theta_6 + \phi) = sin(\theta_4)cos(\theta_5)l_{A_2} \tag{3.32}$$

$$r \cdot sin(\theta_6 + \phi) = -cos(\theta_4)l_{A_2} - l_{A_3} \tag{3.33}$$

$$p'_z = sin(\theta_4)sin(\theta_5)l_{A_2} \tag{3.34}$$

When we combine above with Equation (3.30) and (3.31) and obtain

$$\theta_4 = atan2\left(\pm\sqrt{1 - cos(\theta_4)^2}, cos(\theta_4)\right) \tag{3.35}$$

where

$$cos(\theta_4) = \frac{(p'_x + l_{A_4})^2 + p'^2_y + p'^2_z - l^2_{A_2} - l^2_{A_3}}{2l_{A_2}l_{A3}} \tag{3.36}$$

Using Equation 3.34 we can get θ_5

$$\theta_5 = atan2(sin(\theta_5), \pm\sqrt{1 - sin(\theta_5)^2}) \tag{3.37}$$

where

$$sin(\theta_5) = \frac{p'_z}{sin(\theta_4)l_{A_2}} \tag{3.38}$$

We can then solve for θ_6 by dividing Equation 3.33 by Equation 3.32.

$$\frac{r \cdot sin(\theta_6 + \phi)}{r \cdot cos(\theta_6 + \phi)} = tan(\theta_6 + \phi) = \frac{-cos(\theta_4) l_{A_2} - l_{A_3}}{sin(\theta_4) cos(\theta_5) l_{A_2}} \tag{3.39}$$

$$\theta_6 = atan2(-(cos(\theta_4) l_{A_2} + l_{A_3}), sin(\theta_4) cos(\theta_5) l_{A_2}) - \phi \tag{3.40}$$

Now that we have θ_4, θ_5 and θ_6 we reconstruct G in reference to joint A_1, A_2 and A_3. We will call this G^*. Like G we will have a right (G^*_R) and left (G^*_R) of G:

$$G^*_L = \begin{bmatrix} g^*_{11} & g^*_{12} & g^*_{13} & g^*_{14} \\ g^*_{21} & g^*_{22} & g^*_{23} & g^*_{24} \\ g^*_{31} & g^*_{32} & g^*_{33} & g^*_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.41}$$

$$G^*_R = \begin{bmatrix} cos(\theta_1)cos(\theta_2)cos(\theta_3) - sin(\theta_1)sin(\theta_3) & cos(\theta_1)sin(\theta_3) + sin(\theta_1)cos(\theta_2)cos(\theta_3) & sin(\theta_2)cos(theta_3) & 0 \\ -cos(\theta_1)sin(\theta_2) & -sin(\theta_1)sin(\theta_1) & cos(\theta_2) & l_{A_2} \\ sin(\theta_1)cos(\theta_3) + cos(\theta_1)cos(\theta_2)sin(\theta_3) & sin(\theta_1)cos(\theta_2)sin(\theta_3) - cos(\theta_1)cos(\theta_3) & sin(\theta_2)sin(\theta_3) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.42}$$

as before we can copare the elements of G^*_L and G^*_R. Specifically compare element $(2, 3)$. We then get

$$cos(\theta_2) = \begin{array}{l} a'_z sin(\theta_4) sin(\theta_5) - a'_y (cos(\theta_4) cos(\theta_6) + sin(\theta_4) cos(\theta_5) sin(\theta_6)) \\ -a'_x (cos(\theta_4) sin(\theta_6) - sin(\theta_4) cos(\theta_5) cos(\theta_6)) \end{array} \tag{3.43}$$

$$\theta_2 = atan2(\pm\sqrt{1 - cos(\theta_2)^2}, cos(\theta_2)) \tag{3.44}$$

If we take elements $(1, 3)$ and $(3, 3)$ of G^*_L with those of G^*_R we get

$$g_{13}^{*} = \begin{array}{l} a_x'(cos(\theta_4)cos(\theta_5)cos(\theta_6) + sin(\theta_4)sin(\theta_6) + a_z'cos(\theta_4)sin(\theta_5) \\ +a_y'(sin(\theta_4)cos(\theta_6) - cos(\theta_4)cos(\theta_5)sin(\theta_6)) \end{array} \tag{3.45}$$

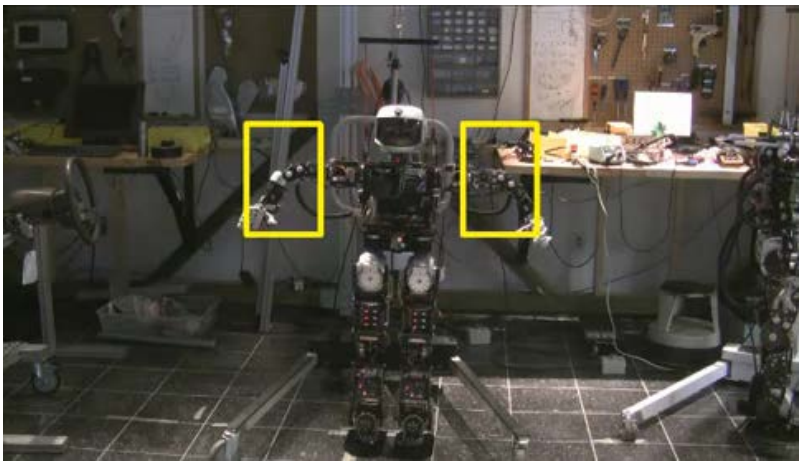
By dividing these two equations we can get θ_1

$$\theta_2 = atan2(g_{33}^{*}, g_{13}^{*}) \tag{3.46}$$

and thus

$$IF: \quad sin(\theta_2) < 0 \quad THEN: \quad \theta_1 = \theta_1 + \pi \tag{3.47}$$

Fig. 3.31 shows the example of using Hubo-Ach to move the right end-effector to the desired work space coordinates using the IK method described in this section.

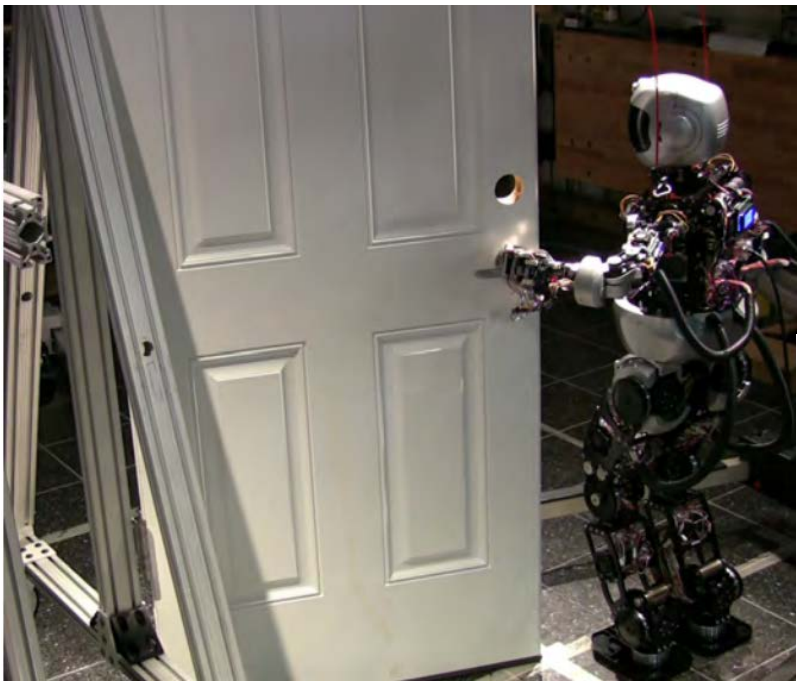


Video: http://danlofaro.com/phd/ik/#HuboTwoArmIk

Figure 3.31: Hubo preforming 6-DOF IK in real-time using method discussed in Section L.0.4

3.8 Verification: Door Opening

Section 3.3, 3.6.1, and 3.6 verrified the functionality of Hubo-Ach under different circumstances.



http://danlofaro.com/phd/door/

Figure 3.32: Indipendent validation of Hubo-Ach via Zucker et. al.[12] work in*Continuous Trajectory Optimization for Autonomous Humanoid Door Opening.*

Zucker et. al.[12] independently validates Hubo-Ach through their work in*Continuous Trajectory Optimization for Autonomous Humanoid Door Opening.* Fig. 3.32 shows Zucker's work

In conclusion Hubo-Ach is validated as being a useful *unified algorithmic framework for complex systems and humanoid robots* by peers. Shown to have consistent system performance in Section 3.3. It is verified via implementation of full body

kinematic examples in Section 3.6.1 and 3.7.

4. Hubo-Ach Manual

This section gives the prerequisites for the Hubo-Ach system, shows the user how to install/un-install the system, and how to use the built in tools. Programming examples in C/C++ and Python are also given.

4.1 Prerequisites

The following items are needed to run Hubo-Ach on a Hubo:

- Hubo2+ or OpenHubo (Virtual Hubo)
- SocketCAN compatible CAN card
- Debian based linux install - tested with Ubuntu 12.04 LTS
- Ach IPC installed

4.2 Installation

4.2.1 From Hubo-Ach Dep (Recommended)

Updates to latest release:

```
$ hubo-ach update
```

Updates to latest develop:

```
$ hubo-ach update develop
```

From Repo:

(1) Add one of the two lines to */etc/apt/sources.list*

```
deb http://www.repo.danlofaro.com/release precise main \# Development
deb http://www.drc-hubo.com/release precise main \# Stable Release
```

(2) Install via apt-get:

```
$ sudo apt-get update
$ sudo apt-get install hubo-ach hubo-dev
```

or

```
$ hubo-ach update apt-get
```

4.2.2 From Source

Install from source

Download the source and install

```
$ git clone https://github.com/hubo/hubo-ach.git
$ cd hubo-ach
$ autoreconf -i
$ ./hubo-ach-install.sh
```

Uninstall/Clean Hubo-Ach

Removed all Hubo-Ach version from deb, apt-get and source

```
$ hubo-ach clean
```

4.3 Usage

Starting Hubo-Ach will automatically start the interface between hubo and the user called hubo-console

4.3.1 Hubo-Ach Main Interface

Available commands

```
$ hubo-ach
```

Start Hubo-Ach on Hubo

```
$ hubo-ach start
```

Start Hubo-Ach on OpenHubo (Virtual Hubo)

```
$ hubo-ach virtual
```

4.3.2 Update Hubo-Ach

Updates to latest release:

```
$ hubo-ach update
```

Updates to latest develop:

```
$ hubo-ach update develop
```

Updates via apt-get

Dependent on your apt-get entry in */etc/apt/source.list*:

```
deb http://www.repo.danlofaro.com/release precise main # Develop
deb http://www.drc-hubo.com/release precise main # Stable
```

```
$ hubo-ach update apt-get
```

Remove Hubo-Ach

Removed all installed versions of Hubo-Ach including from source.

```
$ hubo-ach clean
```

Start Hubo-Ach Console

```
$ hubo-ach console
```

Start Hubo-Ach Read Tool

```
$ hubo-ach read
```

Start Remote Connection

Connection is made from the client to the server where the robot is the server. The robot's has an IP address of xxx.xxx.xxx.xxx and is the hubo-ach computer. Note: you have to have to enable the network daemon via the process described here: http://golems.github.com/ach/manual/#AEN399

```
$ hubo-ach remote xxx-xxx-xxx-xxx
```

Kill Remote Connection

```
$ hubo-ach remote kill
```

Make Ach Channes

Creates Ach channels for Hubo-Ach without starting the Hubo-Ach Daemon

```
$ hubo-ach make
```

4.3.3 Hubo-Console

Hubo-Console is a basic user interface between the Hubo and the user. It allows you to do the following:

- Home a single joint
- Home all joints at once
- Reset joint errors
- Initialize sensors

Note: In all examples below XXX stands for the standard joint naming i.e. RHP, RHR, LSP, LSR, etc. Hubo-Console will start automatically when you type:

```
$ hubo-ach start
```

If Hubo-Ach is already started you can start hubo-console by:

```
$ hubo-ach console
```

Once Hubo-Console has started you will be able to:

Home Joint XXX

This will make the joint move to find the limit switch then goto its predefined offset. The reference will be set to zero.

```
>> hubo-ach: home XXX
```

Home All Joints

This will make all of the joints move to find their respective limit switches and goto their predefined offsets at the same. The reference to all joints are set to zero. Note: All joints will move at the same time. The rotbot should not be on the ground when this is done.

```
>> hubo-ach: homeAll
```

Initialize All Sensors

This will initialize all sensors including the IMU and FT sensors. The robot should be off the ground and not moving.

```
>> hubo-ach: iniSensors
```

Initialize All joints

This will initialize all joints. Note: They will maintain the current control mode. If they were inactive they will be active and able to read back encoder values at this point.

```
>> hubo-ach: initializeAll
```

Clear Errors on Joint XXX

This will clear the following errors on joint XXX.

- Big Error
- Encoder Error
- Homing Error

```
>> hubo-ach: reset XXX
```

Clear Errors on All Joints

This is the same as reset but will clear errors on all active joints

```
>> hubo-ach: resetAll
```

Joint XXX goto position

Commands joint XXX to goto position YYY (in radians)

```
>> hubo-ach: goto XXX YYY
```

Turn on/off Joint XXX Controller

This will turn on or off the controller for joint XXX. Note: Y represents the desired state

- 1 = on
- 0 = off

```
>> hubo-ach: ctrl XXX Y
```

Turn on/off All Joint Controllers

This is the same as Turn on/off Joint XXX Controller but it applies to all joints:

```
>> hubo-ach: ctrlAll Y
```

Check Status of Joint XXX

Check the status of joint XXX

```
>> hubo-ach: status XXX
```

4.3.4 Hubo-Read

Hubo-Read is a simple tool that prints out the reference and state channels to the console. You can start Hubo-Read in one of two ways:

Method 1

```
$ hubo-ach read
```

Method 2

Note: the sudo is needed because it uses RT permissions for the loop.

```
$ sudo hubo-read
```

What you will see

```
t = 1363724430.086347873
WST : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
NKY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
NK1 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
NKP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LSP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LSR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LSY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LEB : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LWY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LWR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LWP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RSP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RSR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RSY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
REB : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RWY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RWR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RWP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LHY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LHR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LHP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LKN : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LAP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LAR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RHY : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RHR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RHP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RKN : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RAP : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RAR : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RF1 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RF2 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RF3 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RF4 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
RF5 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LF1 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LF2 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LF3 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
LF4 : Cmd = 0.000000     Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
```

```
LF5 : Cmd = 0.000000      Ref = 0.000000     Enc = 0.000000     Cur = 0.000000     Tmp = 0.000000
    : Mx = 0.000000     My = 0.000000     Fz = 0.000000
    : Mx = 0.000000     My = 0.000000     Fz = 0.000000
    : Mx = 0.000000     My = 0.000000     Fz = 0.000000
    : Mx = 0.000000     My = 0.000000     Fz = 0.000000
    : Ax = 0.000000     Ay = 0.000000     Az = 0.000000
    : Ax = 0.000000     Ay = 0.000000     Az = 0.000000
    : Ax = 0.000000     Ay = 0.000000     Wx = 0.000000     Wy = 0.000000
```

4.4 Simulator

This section shows how to run a Hubo simulator in conjunction with Hubo-Ach. Note: The simulator is a full 3D simulator and is recomended to run on a computer other the Hubo body computer. It will work on it but it will be slow.

4.4.1 Prerequisites

OpenHubo

To install OpenHubo follow the directions here:

- http://dasl.mem.drexel.edu/drcwiki/index.php/OpenHubo_Introduction

Assuming you have all of the prerequisites you can simply do the following to install OpenHubo:

```
$ git clone --recursive https://github.com/hubo/openHubo.git
$ cd openHubbo
$ ./setup
```

4.4.2 Using the Simulator

Once OpenHubo is installed you can run run the simulator. The simulator at the moment is restricted to kinimatic output. Dynamics do not run in real-time.

This simulator creates two models overlaid on each-other. The green model is the commanded reference sent to the actuator. The solid model is the actual position as read from the encoders. Note: if the robot is not on this will stay zero but the green reference will move. To run the simulator:

With No Physics (fast):

Starts OpenHubo running with no physics. This is good for watching what the robot is doing live or to preview trajectories.

```
$ hubo-ach sim openhubo nophysics
```

With Physics:

Starts OpenHubo running with physics. This runs at about 35% real time (on an i7 processor). The simulator and hubo-ach are synced via triggering from newly received messaged on the following Ach channels:

- HUBO_CHAN_VIRTUAL_TO_SIM_NAME
- HUBO_CHAN_VIRTUAL_FROM_SIM_NAME

Please note that the state channel will have simulation time NOT real time.

```
$ hubo-ach sim openhubo physics
```

4.4.3 Run Visualizer

You can use OpenHubo as a real-time (live) visualizer of our state data on your computer in which you login to the Hubo from. This will show the OpenHubo model using no physics. The green shows what the joints are commanded to and the grey

show where the joints are. This will run with little to no lag/latency on an i5 or i7 processor. In order to do this start hubo-ach normally on the hubo:

```
(hubo@xxx.xxx.xxx.xxx) $ hubo-ach start
```

On your control computer (not the hubo) start the simulator with remote

```
(i5 or i7) $ hubo-ach sim openhubo nophysics remote xxx.xxx.xxx.xxx
```

4.5 Programming

This section will show quick examples of how to program using Hubo-Ach in C/C++ and Python

4.5.1 C/C++

The C/C++ Example is available below. This is bare bones for you to:

- Get the latest feed-back (state) channel information.
- Set the feed-forward (reference) information.

C/C++ Example (hubo-simple-demo.c)

```
/* Standard Stuff */
#include <string.h>
#include <stdio.h>

/* Required Hubo Headers */
#include <hubo.h>

/* For Ach IPC */
#include <errno.h>
#include <fcntl.h>
```

```
#include <assert.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include <stdbool.h>
#include <math.h>
#include <inttypes.h>
#include "ach.h"



/* Ach Channel IDs */
ach_channel_t chan_hubo_ref;      // Feed-Forward (Reference)
ach_channel_t chan_hubo_state;    // Feed-Back (State)

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

    /* Open Ach Channel */
    int r = ach_open(&chan_hubo_ref, HUBO_CHAN_REF_NAME , NULL);
    assert( ACH_OK == r );

    r = ach_open(&chan_hubo_state, HUBO_CHAN_STATE_NAME , NULL);
    assert( ACH_OK == r );




    /* Create initial structures to read and write from */
    struct hubo_ref H_ref;
    struct hubo_state H_state;
    memset( &H_ref,   0, sizeof(H_ref));
    memset( &H_state, 0, sizeof(H_state));
```

```
    /* for size check */
    size_t fs;

    /* Get the current feed-forward (state) */
    r=ach_get(&chan_hubo_state, &H_state, sizeof(H_state), &fs, NULL, ACH_O_LAST);
    if(ACH_OK != r) {
        assert( sizeof(H_state) == fs );
    }

    /* Set Left Elbow Bend (LEB) and Right Shoulder Pitch (RSP) */
    /* to  -0.2 rad and 0.1 rad respectively */
    H_ref.ref[LEB] = -0.2;
    H_ref.ref[RSP] = 0.1;

    /* Print out the actual position of the LEB */
    double posLEB = H_state.joint[LEB].pos;
    printf("Joint = %f\r\n",posLEB);

    /* Print out the Left foot torque in X */
    double mxLeftFT = H_state.ft[HUBO_FT_L_FOOT].m_x;
    printf("Mx = %f\r\n", mxLeftFT);

    /* Write to the feed-forward channel */
    ach_put( &chan_hubo_ref, &H_ref, sizeof(H_ref));

}
```

Where the default **MakeFile** is:

C/C++ Default MakeFile

```
default: all


CFLAGS := -I./include -g --std=gnu99
CC := gcc


BINARIES := hubo-simple-demo
all : $(BINARIES)


LIBS := -lach


hubo-simple-demo: src/hubo-simple-demo.o
gcc -o $@ $< $(LIBS)


%.o: %.c
$(CC) $(CFLAGS) -o $@ -c $<


clean:
rm -f $(BINARIES) src/*.o
```

Run the example

The C/C++ example uses the MakeFile seen in Section 4.5.1. You just need to make and run

```
$ make clean
$ make
$ ./hubo-simple-demo
```

4.5.2 Python

The Python Example is available below. This is bare bones for you to:

- Get the latest feed-back (state) channel information
- Set the feed-forward (reference) information

Please note that:

- Must use Hubo-Ach ¿= 0.0.20130319
- The Ach python bindings must be installed. You can install via PIP (see below)

```
$ sudo apt-get install python-pip
$ sudo pip install http://code.golems.org/src/ach/py_ach-latest.tar.gz
```

Python Example (hubo-simple-demo.py)

```
#!/usr/bin/env python
# /* -*-  indent-tabs-mode:t; tab-width: 8; c-basic-offset: 8  -*- */

import hubo_ach as ha
import ach
import sys
import time
from ctypes import *

# Open Hubo-Ach feed-forward and feed-back (reference and state) channels
s = ach.Channel(ha.HUBO_CHAN_STATE_NAME)
r = ach.Channel(ha.HUBO_CHAN_REF_NAME)
s.flush()
r.flush()

# feed-forward will now be refered to as "state"
state = ha.HUBO_STATE()

# feed-back will now be refered to as "ref"
```

```
ref = ha.HUBO_REF()


# Get the current feed-forward (state)
[statuss, framesizes] = s.get(state, wait=False, last=False)


# Set Left Elbow Bend (LEB) and Right Shoulder Pitch (RSP)
# to  -0.2 rad and 0.1 rad respectively
ref.ref[ha.LEB] = -0.2
ref.ref[ha.RSP] = 0.1


# Print out the actual position of the LEB
print "Joint = ", state.joint[ha.LEB].pos


# Print out the Left foot torque in X
print "Mx = ", state.ft[ha.HUBO_FT_L_FOOT].m_x


# Write to the feed-forward channel
r.put(ref)


# Close the connection to the channels
r.close()
s.close()
```

Run the example

The python example can be run via:

```
$ python ./hubo-simple-demo.py
```

4.6 Connecting a Simulator to Hubo-Ach

This is a brief example of how to connect a simulator to Hubo-Ach. In this example we will be running Hubo-Ach in simtime mode. This makes Hubo-Ach wait for a trigger from the simulator. This trigger tells Hubo-Ach that it has updated the state data with the simulated state data. Hubo-Ach will then run and send a trigger to the simulator telling it that it can do its next calculation.

Hubo-Ach is simulator agnostic. The examples is given in this section uses the OpenHubo simulator described in Section 4.6.1

4.6.1 Simulator

The simulator used for Hubo-Ach is the OpenHubo. OpenHubo is an open-source kinematic and dynamic simulator for the the Hubo2 and Hubo2+ series robots. It was developed by the Drexel Autonomous Systems Lab and runs using the open-source robot simulation environment OpenRAVE[16]. Fig. 4.1 shows the OpenHubo shell model and collision model.

The masses and lengths are of the OpenHubo model are all based off of the CAD model. The shell model includes an external skin based off of the CAD model of the Hubo's shell. This model is high polygon count and thus tends to require more processing time to detect collisions. The collision model is constructed out of primitives in order to decrease the complexity of the model and decrease required processing time. The collision model is a representation of the shell model. It does not precisely fit the contours but through experimentation and use has been calibrated to be a good representation of the Hubo's outer shell.

Fig 4.2 shows the diagram of how the OpenHubo simulator is connected to Hubo-Ach. No changes to previous controllers are required for them to work with the simulator. Just as before the desired reference θ_d being filtered before applied to

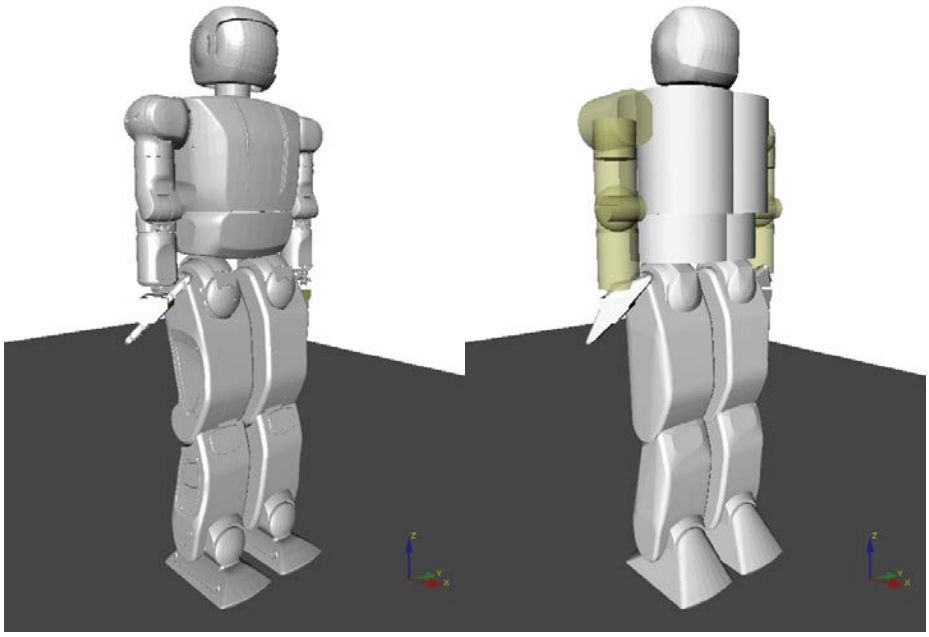


Figure 4.1: OpenHubo model of the Hubo2 humanoid robot developed by the Drexel Autonomous Systems Lab and runs using the open-source robot simulation environment OpenRAVE[16]. (Left) Shell Model - High polygon count. (Right) Collision model - Made with primitives.

Hubo via Hubo-Ach. θ_d is sent through a filter that reduces the *jerk* on the actuator then the new reference θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle. At this point the *to simulator* trigger, Γ_{ts}, is set high and the OpenHubo simulator reads θ_c. The simulator waits until Hubo-Ach is ready until it starts its next set of cycles. The reference is set within OpenHubo and solved with a simulation period of T_{sim}. The simulation period T_{sim} must be an integer deviser of the robot real-time period T_r. In this case

$$T_r = 0.005\ s \tag{4.1}$$

$$T_{sim} = \frac{T_r}{n} \tag{4.2}$$

Once the simulator has gone through n cycles the current state, H_{state} is placed on the Hubo-Ach **FeedForward** channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of the *from simulator* trigger, Γ_{fs}, to continue on to the next cycle.

The external controllers do not know weather Hubo-Ach is running in *simulation* or *real-time* mode. In order to ensure a Hubo-Ach controller stays what ever timing method is being used the controller can do any of the following:

- Wait for the Γ_{fs} trigger
- Wait for a new H_Pstate to be updated
- Watch the time listed within H_{state}

If the given task does not require physics or feedback from H_{state} then you can run in *no physics* mode. *No physics* mode only gives collisions, joint angles and ideal

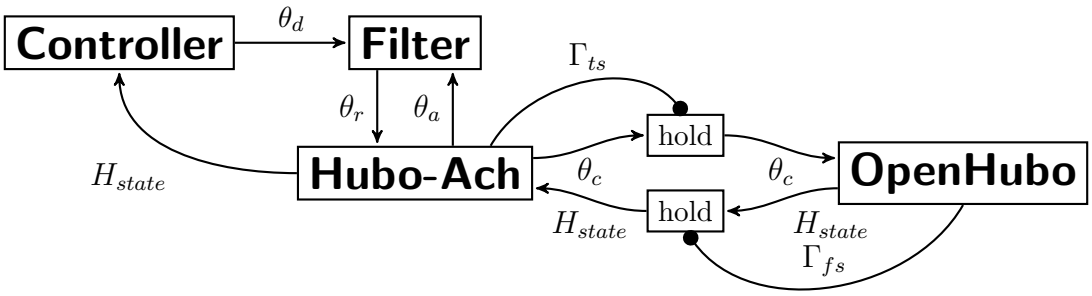


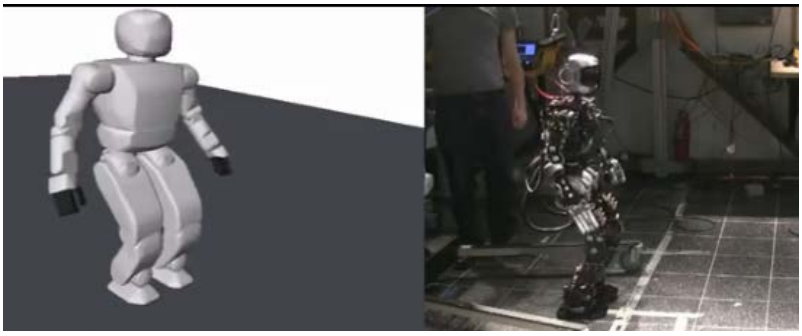
Figure 4.2: Diagram of how the OpenHubo simulator is connected to Hubo-Ach. No changes to previous controllers are required for them to work with the simulator. Just as before the desired reference θ_d being filtered before applied to Hubo via Hubo-Ach. θ_d is sent through a filter that reduces the *jerk* on the actuator then the new reference θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the OpenHubo simulator reads θ_c. The reference is set within OpenHubo and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward** channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle.

Table 4.1: OpenHubo simulator sim-time and real-time comparison chart. Shows the maximum percent real-time the OpenHubo simulator is capable of preforming at where 100% is real-time. All tests were preformed on an Intel i7 running at 2.8Ghz with 18Gb of RAM.

Mode	Timing	Maximum Percent Real-Time (%)
Physics	Sim-Time	37%
No Physics	Real-Time or Sim-Time	362%

feedback from the sensors. In addition *no physics* is capable of running much faster then real-time if needed.

Fig. 4.3 shows the capability of Hubo-Ach to run in both sim-time and real-time modes. This is the same statically stable trajectory as seen in Section 5.1.1



Video: http://danlofaro.com/phd/walking/#WalkingHuboAndOpenHubo

Figure 4.3: Hubo and OpenHubo walking using Hubo-Ach in Real-Time and Sim-Time Respectively

4.6.2 Setup

Step 1: Start Hubo-Ach in SimTime mode

This mode makes hubo-ach wait for a trigger from the simulator. This trigger is when the HUBO_CHAN_VIRTUAL_FROM_SIM_NAME channel has been updated.

```
$ hubo-ach sim
```

Step 2: Run your simulator

Run your simulator. The simulator must do the following in this order

(Sudo Code)

```
Open all needed Channels
Write to HUBO_CHAN_VIRTUAL_FROM_SIM_NAME channel
Loop:
    Wait for HUBO_CHAN_VIRTUAL_TO_SIM_NAME channel update
    Set feed forward data from H_state.joint[jnt].ref to your simulator
    Do simulation
    Populate state data to H_state struct
        H_state.joint[jnt].pos
```

```
        H_state.imu[i].*
        H_state.ft[i].*
    Write H_state to HUBO_CHAN_STATE_NAME
    Write to HUBO_CHAN_VIRTUAL_FROM_SIM_NAME channel
    Goto Loop
```

Step 3: Run a Hubo-Ach controller of your choosing

You can now run any hubo-ach controller with no modification required. Note: If you want to take into account the simtime then it must either:

- Watch the state time H_state.time
- Wait for the following triggers:
  - HUBO_CHAN_VIRTUAL_FROM_SIM_NAME
  - HUBO_CHAN_VIRTUAL_TO_SIM_NAME

4.6.3 C/C++ Simulation Example

This example code shows the basics of connecting to Hubo-Ach using simtime mode with triggering. The example can be found on the hubo group on github in the example/simtime branch:

```
$ git clone https://github.com/hubo/hubo-simple-demo.git
$ cd hubo-simple-demo
$ git checkout example/simtime
```

C/C++ Simulation Example (hubo-simple-demo.c) - simtime branch

```
/* Standard Stuff */
#include <string.h>
#include <stdio.h>
```

```
/* Required Hubo Headers */
#include <hubo.h>

/* For Ach IPC */
#include <errno.h>
#include <fcntl.h>
#include <assert.h>
#include <unistd.h>
#include <pthread.h>
#include <ctype.h>
#include <stdbool.h>
#include <math.h>
#include <inttypes.h>
#include "ach.h"

/* Ach Channel IDs */
ach_channel_t chan_hubo_ref;      // Feed-Forward (Reference)
ach_channel_t chan_hubo_state;    // Feed-Back (State)
ach_channel_t chan_hubo_to_sim;    // To Sim
ach_channel_t chan_hubo_from_sim;    // From Sim

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

    int i = 0;

    /* Open Ach Channel */
    int r = ach_open(&chan_hubo_ref, HUBO_CHAN_REF_NAME , NULL);
    assert( ACH_OK == r );

    r = ach_open(&chan_hubo_state, HUBO_CHAN_STATE_NAME , NULL);
    assert( ACH_OK == r );
```

```
/* Create initial structures to read and write from */
struct hubo_ref H_ref;
/* this is a place holder for what ever */
/* way your store your sim ref data     */
struct hubo_ref H_ref_your_sim;
/* this is a place holder for what ever */
/* way your store your sim state data   */
struct hubo_state H_state_your_sim;
struct hubo_state H_state;
struct hubo_virtual H_sim;
memset( &H_ref,   0, sizeof(H_ref));
memset( &H_ref,   0, sizeof(H_ref_your_sim));
memset( &H_state, 0, sizeof(H_state));
memset( &H_state, 0, sizeof(H_state_your_sim));
memset( &H_sim, 0, sizeof(H_sim));

/* for size check */
size_t fs;

/* Flush old messages */
ach_flush(&chan_hubo_to_sim);
ach_flush(&chan_hubo_from_sim);

/* send the from sim trigger */
ach_put( &chan_hubo_from_sim, &H_sim, sizeof(H_sim));

/* Start the sim time loop */
while(1) {
    /* Waits for hubo-ach trigger */
    r = ach_get( &chan_hubo_to_sim, &H_sim, sizeof(H_sim), &fs, NULL, ACH_O_WAIT );
    if(ACH_OK != r) {
```

```
        assert( sizeof(H_sim) == fs );
    }

    /* Get the current feed-forward (state) */
    r = ach_get( &chan_hubo_state, &H_state, sizeof(H_state), &fs, NULL, ACH_O_LAST );
    if(ACH_OK != r) {
        assert( sizeof(H_state) == fs );
    }

    /* Sets the commanded joint value to your simulators feed forward */
    for( i = 0; i < HUBO_JOINT_COUNT; i++){
H_ref_your_sim.ref[i] = H_state.joint[i].ref;
    }

   /* ---------------------------- */
   /* run your simulator stuff here */
   /* ---------------------------- */
   /* Note1: you can run multiple itterations of your      */
   /*       sim here before you post the data from the sim */
   /*       i.e. if you want to run your sim at 1khz then  */
   /*       step your sim N times                          */
   /*       where N = ceil(1000/HUBO_LOOP_PERIOD)   */
   /*       HUBO_LOOP_PERIOD is defigned in hubo.h  */
   /* Note2: Hubo-Ach updates the time via the H_sim.time. */

   /* Set time to simtime */
   /* H_sim.time = Time from your simulation in seconds*/

    /* set your state data to the hubo-ach state data */
    /* Joint pos*/
    for( i = 0; i < HUBO_JOINT_COUNT; i++){
```

```
        // actual joint position from simulator
        H_state.joint[i].pos = H_state_your_sim.joint[i].pos;
    }

    /* FT */
    // force torque from sim  note: 4 will soon be changed to HUBO_FT_COUNT
    for( i = 0; i < 4; i++){
        H_state.ft[i].m_x = H_state_your_sim.ft[i].m_x;
        H_state.ft[i].m_y = H_state_your_sim.ft[i].m_y;
        H_state.ft[i].f_z = H_state_your_sim.ft[i].f_z;
    }

    for( i = 0; i < HUBO_IMU_COUNT; i++){        // IMU Data from your sim
        H_state.imu[i].a_x = H_state_your_sim.imu[i].a_x;
        H_state.imu[i].a_y = H_state_your_sim.imu[i].a_y;
        H_state.imu[i].a_z = H_state_your_sim.imu[i].a_z;
        H_state.imu[i].w_x = H_state_your_sim.imu[i].w_x;
        H_state.imu[i].w_y = H_state_your_sim.imu[i].w_y;
        H_state.imu[i].w_z = H_state_your_sim.imu[i].w_z;
    }

    /* at this point hubo-ach has been waiting for the sim to be done */
    /* now that all of the sim data has been set to the state you can */
    /* tell hubo-ach that it can resume.  This is done by giving it the */
    /* from_sim trigger */

    /* Push new state data */
    ach_put( &chan_hubo_state, &H_state, sizeof(H_state));

    /* send the from sim trigger */
    ach_put( &chan_hubo_from_sim, &H_sim, sizeof(H_sim));
```

```
    }
}
```

Run the example

The simulation example uses the same make file from the hubo-simple-demo in Section 4.5.1. You just need to make and run

```
$ make clean
$ make
$ ./hubo-simple-demo
```

5. Experiment

This section gives examples of the Hubo-Ach system being used on the physical and simulated systems. Examples include: a walking controller implementation on the physical robot and in simulation; visual servoing while walking using the physical robot; and active damping implementation using the physical robot.

5.1 Walking

This section shows examples of how Hubo-Ach was used for stable walking. Examples are given using:

- Hubo2+ (Physical Robot)
- OpenHubo (Simulator)
- RobotSim Hubo (Simulator)

section 5.1.1 shows how the open-loop walking trajectory is created.

Section 5.1.2 shows how the open loop walking trajectory is run in sim-time on OpenHubo using Hubo-Ach. Section 5.1.3 shows how the open loop walking trajectory is run in sim-time on RobotSim using Hubo-Ach. Section 5.1.4 shows the same walking trajectory running on the real Hubo hardware in real-time using Hubo-Ach. It also shows the difference between running in sim-time and real-time. Section I shows the result of a five day *hack-a-thon* using Hubo-Ach to add dynamic walking capability.

5.1.1 Walking Pattern Generation

The walking pattern demonstrated in this section is generated based on the work of Park et. al.[49] A walking pattern is the way in which a legged robot, in this case two legged, moves its joints to create a walking gate while maintaining stability. The walking pattern consists of two major phases:

- Single Support Phase (SSP)
- Double Support Phase (DSP)

Single support phase is when one foot is on the ground. This phase is when one leg moves from one stepping position to the other. The ZMP must remain above the planted foot to guarantee stability.

Double support phase is when both feet are planted on the ground. When in this phase the ZMP moves from above one foot to the other along the stable area as seen in Fig. G.1.

Fig 5.1 and 5.2 shows the walking pattern phases on a Hubo model in the x and y direction respectively. In these figures A_R and A_L defines the left and right ankles respectively. t_1 is the time of the starting of the step, t_2 defines the landing of the stepping foot. t_0 defines time when the stepping foot is at peak step height. P defines the hip location. $\widetilde{x}$ defines the walking velocity. $\widetilde{y}$ defines the body sway velocity. The middle diagram depicts the SSP and the left and right diagrams show the DSP.

The walking patterns are generated creating a joint space trajectory with a period T of 0.005 *sec*. The patterns keep the ZMP criteria described in Section G. These walking patterns are used to test the simulated robots and the physical robots. Fig. 5.3 shows the joint space walking pattern verses time.

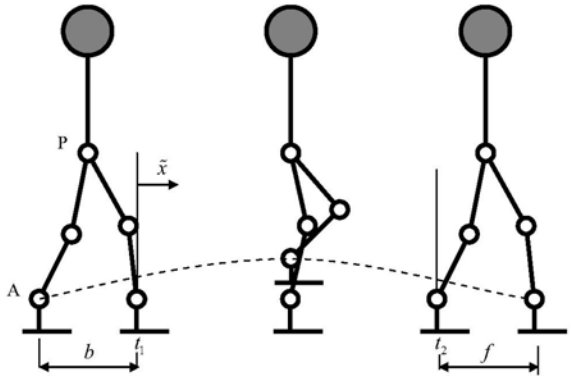


Figure 5.1: Hubo model diagram for ZMP walking in the x direction (side view). b and f are the step lengths for the left and the right foot. A defines the ankle. t_1 is the time of the starting of the step, t_2 defines the landing of the stepping foot. P defines the hip location. $\widetilde{x}$ defines the walking velocity. The middle diagram depicts the SSP and the left and right diagrams show the DSP.

5.1.2 Walking Using OpenHubo Simulator and Hubo-Ach

The walking pattern that was generated in Section 5.1.1 was then applied to the OpenHubo system described in Section 4.6.1 via the Hubo-Ach controller. The walking pattern was applied in sim-time with a period T_{sim} of 0.005 *sec.* The block diagram of the system using OpenHubo in sim-time for a walking trajectory is shown in Fig. 5.7.

In Fig. 5.7 the OpenHubo simulator is connected to Hubo-Ach and is used to run the walking trajectory. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. θ_r is set directly on the **FeedForward** channel thus each joint will have the response as seen in Fig. 3.16 for each commanded reference command at each time step. Hubo-Ach reads the **FeedForward** channel and commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the OpenHubo simulator reads θ_c. The reference is set within OpenHubo and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward**

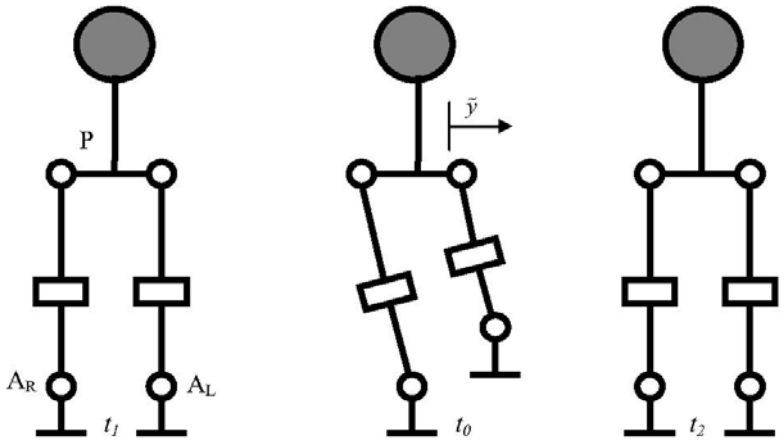


Figure 5.2: Hubo model diagram for ZMP walking in the y direction (front view). A_R and A_L defines the left and right ankles respectively. t_1 is the time of the starting of the step, t_2 defines the landing of the stepping foot. t_0 defines time when the stepping foot is at peak step height. P defines the hip location. $\widetilde{y}$ defines the body sway velocity. The middle diagram depicts the SSP and the left and right diagrams show the DSP.

channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle. In order to keep with the sim-time the *Walking Pattern* also waits for the rising edge of Γ_{fs} to put the next desired reference on the **FeedForward** channel. Fig. 5.5 shows the Virtual Hubo successfully ZMP walking using OpenHubo and Hubo-Ach.

5.1.3 Walking Using RobotSim and Hubo-Ach

The walking pattern that was generated in Section 5.1.1 was then applied to the RobotSim dynamic simulator via the Hubo-Ach controller. RobotSim was developed by Professor Kris Hauser from Indiana University. The simulator was integrated into Hubo-Ach on April 24^{th}, 2013 during a 12 hour *Hack-A-Thon* at Worchester Polytechnic Institute by Daniel M. Lofaro, Jingru Luo and Professor Kris Hauser[50]. The walking pattern was applied in sim-time with a period T_{sim} of 0.005 *sec*. The

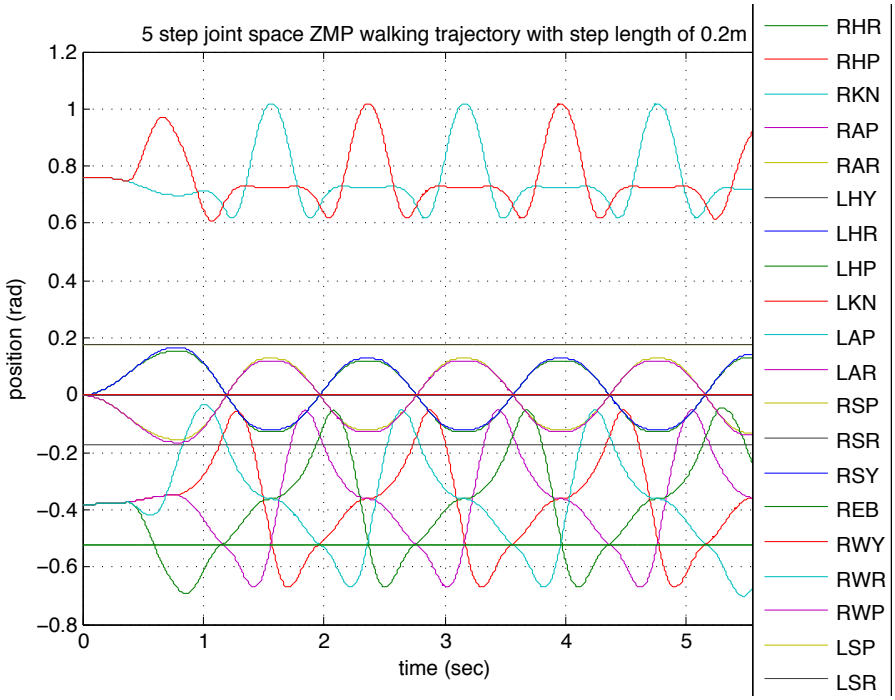


Figure 5.3: Joint space walking pattern. The trajectory sampling period T is 0.005 *sec*. Forward step length is 0.2 m, sway velocity $\widetilde{y}$ is 0.062 $\frac{m}{sec}$, and step period is 0.8 *sec*.

block diagram of the system using RobotSim in sim-time for a walking trajectory is shown in Fig. 5.8.

In Fig. 5.8 the RobotSim simulator is connected to Hubo-Ach and is used to run the walking trajectory. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. θ_r is set directly on the **FeedForward** channel thus each joint will have the response as seen in Fig. 3.16 for each commanded reference command at each time step. Hubo-Ach reads the **FeedForward** channel and commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the RobotSim simulator reads θ_c. The reference is set within RobotSim and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward**

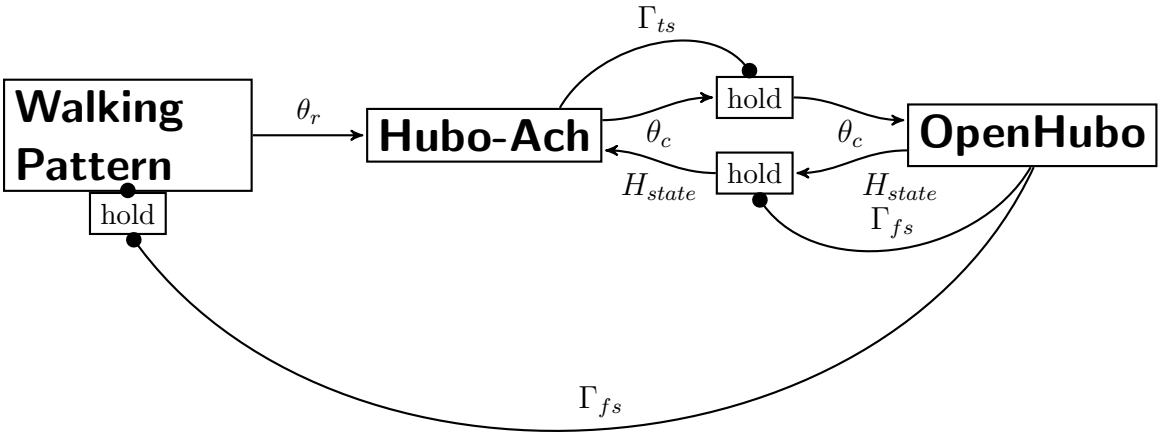
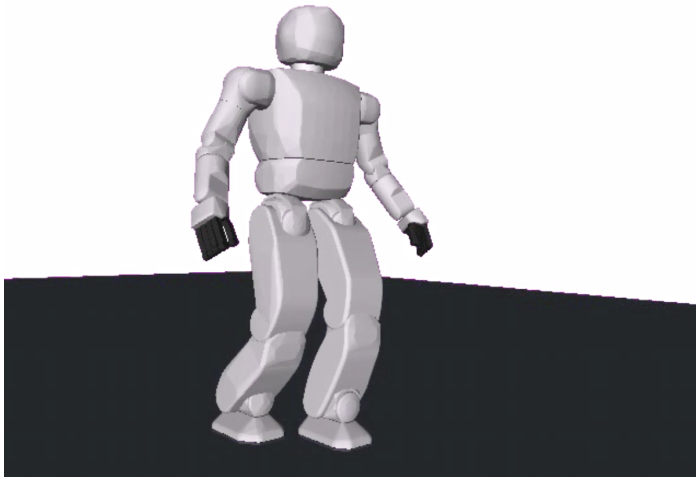


Figure 5.4: Diagram of how the OpenHubo simulator is connected to Hubo-Ach and is used to run a walking trajectory. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. θ_r is set directly on the **FeedForward** channel thus each joint will have the response as seen in Fig. 3.16 for each commanded reference command at each time step. Hubo-Ach reads the **FeedForward** channel and commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the OpenHubo simulator reads θ_c. The reference is set within OpenHubo and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward** channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle. In order to keep with the sim-time the *Walking Pattern* also waits for the rising edge of Γ_{fs} to put the next desired reference on the **FeedForward** channel.



Video: http://danlofaro.com/phd/walking/#WalkingOpenHubo

Figure 5.5: Virtual Hubo in OpenHubo preforming ZMP walking using Hubo-Ach in sim-time based on the walking pattern generated in Section 5.1.1

Video: http://danlofaro.com/phd/walking/#WalkingRobotSim

Figure 5.6: Virtual Hubo in RobotSim preforming ZMP walking using Hubo-Ach in sim-time based on the walking pattern generated in Section 5.1.1

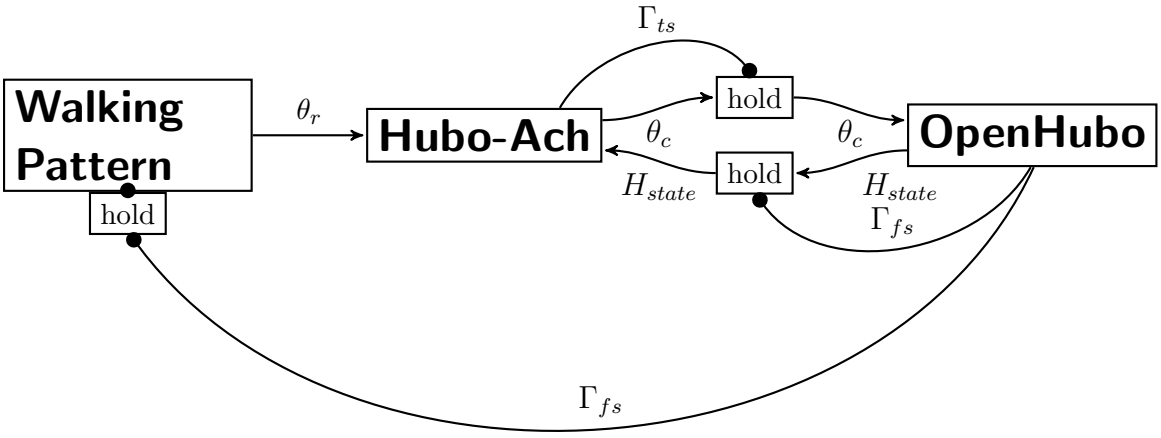


Figure 5.7: Diagram of how the OpenHubo simulator is connected to Hubo-Ach and is used to run a walking trajectory. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. θ_r is set directly on the **FeedForward** channel thus each joint will have the response as seen in Fig. 3.16 for each commanded reference command at each time step. Hubo-Ach reads the **FeedForward** channel and commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the OpenHubo simulator reads θ_c. The reference is set within OpenHubo and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward** channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle. In order to keep with the sim-time the *Walking Pattern* also waits for the rising edge of Γ_{fs} to put the next desired reference on the **FeedForward** channel.

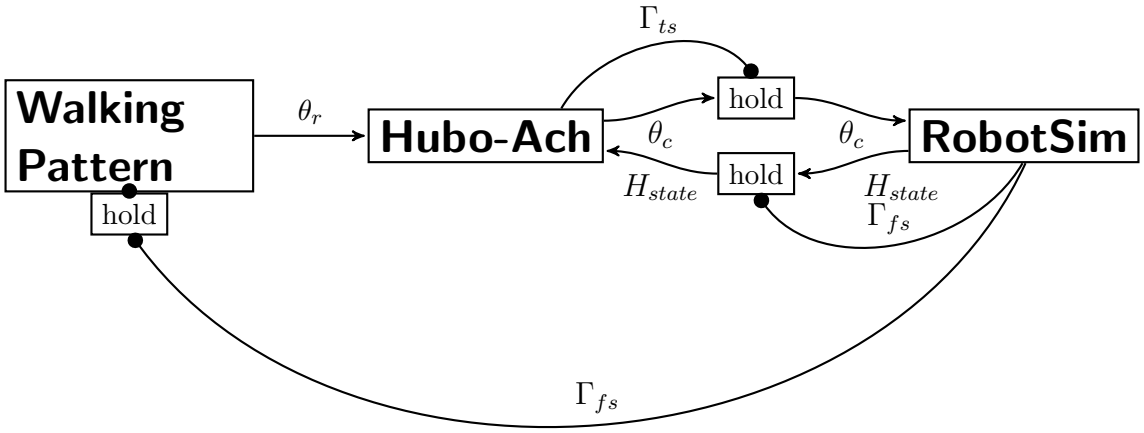


Figure 5.8: Diagram of how the RobotSim simulator is connected to Hubo-Ach and is used to run the walking trajectory. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. θ_r is set directly on the **FeedForward** channel thus each joint will have the response as seen in Fig. 3.16 for each commanded reference command at each time step. Hubo-Ach reads the **FeedForward** channel and commands Hubo at the rising edge of the next cycle. At this point Γ_{ts} is set high and the RobotSim simulator reads θ_c. The reference is set within RobotSim and solved with a simulation period of T_{sim}. Once The state, H_{state} has been determined it is placed on the Hubo-Ach **FeedForward** channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle. In order to keep with the sim-time the *Walking Pattern* also waits for the rising edge of Γ_{fs} to put the next desired reference on the **FeedForward** channel.

channel and the ready trigger Γ_{fs} is raised. Hubo-Ach is waiting for the rising edge of Γ_{fs} to continue on to the next cycle. In order to keep with the sim-time the *Walking Pattern* also waits for the rising edge of Γ_{fs} to put the next desired reference on the **FeedForward** channel. Fig. 5.6 shows the Virtual Hubo successfully ZMP walking using RobotSim and Hubo-Ach.

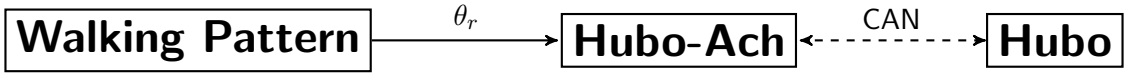


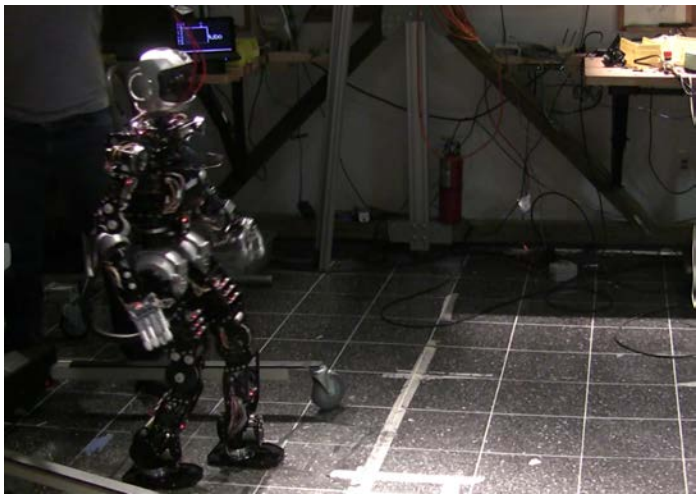
Figure 5.9: Reference θ_r being applied to Hubo via Hubo-Ach. θ_r is set on the **FeedForward** channel, Hubo-Ach reads it then commands Hubo at the rising edge of the next cycle.

5.1.4 Hubo Walking using Hubo-Ach

The walking pattern that was generated in Section 5.1.1 was then applied to the physical Hubo platform using the Hubo-Ach controller. The walking pattern was applied in real-time with a period T_r of 0.005 *sec*. Unlike the simulated versions which run in sim-time the system is now running in real-time; thus it no longer needs to wait for an external trigger. The walking pattern trajectory is now posted to the **FeedForward** channel at an RT period of T_r. The walking pattern generator ensures proper constraints on the velocity, acceleration and jerk and thus the filter seen in Fig. 4.2 is not desired. Fig. 5.9 shows the block diagram of the walking pattern from Section 5.1.1 being run in real-time on the physical Hubo2+ platform. Fig. 5.10 shows the Hubo successfully ZMP walking using OpenHubo and Hubo-Ach. Fig. 5.11 shows the Hubo successfully ZMP walking in place using OpenHubo and Hubo-Ach.

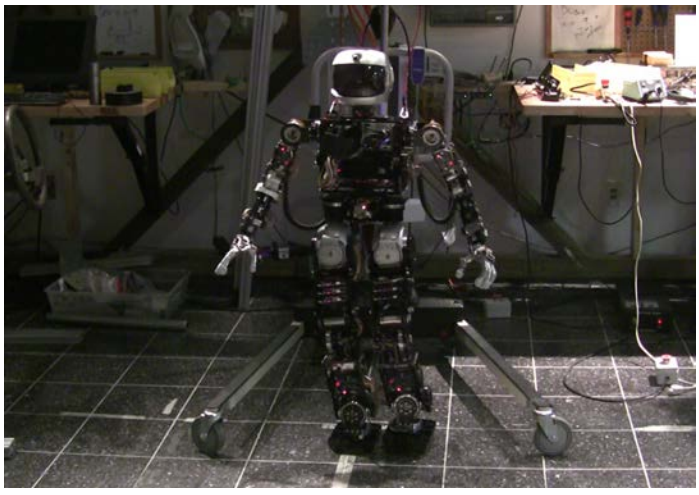
5.2 Visual Serving Example

This section uses visual feedback using an RGB-D (Red Green Blue - Depth) camera and the walking trajectories given in Section 5.1. The goal of this experiment is to have the robot visually track an object, walk towards the object, and stop when it is within 0.2 m of it.



Video: http://danlofaro.com/phd/walking/#WalkingHubo

Figure 5.10: Hubo2+ preforming ZMP walking using Hubo-Ach in real-time based on the walking pattern generated in Section 5.1.1.



Video: http://danlofaro.com/phd/walking/#WalkingInPlaceHubo

Figure 5.11: Hubo2+ preforming ZMP walking in place using Hubo-Ach in real-time based on the walking pattern generated in Section 5.1.1 with a forward velocity of 0.0 $\frac{m}{sec}$

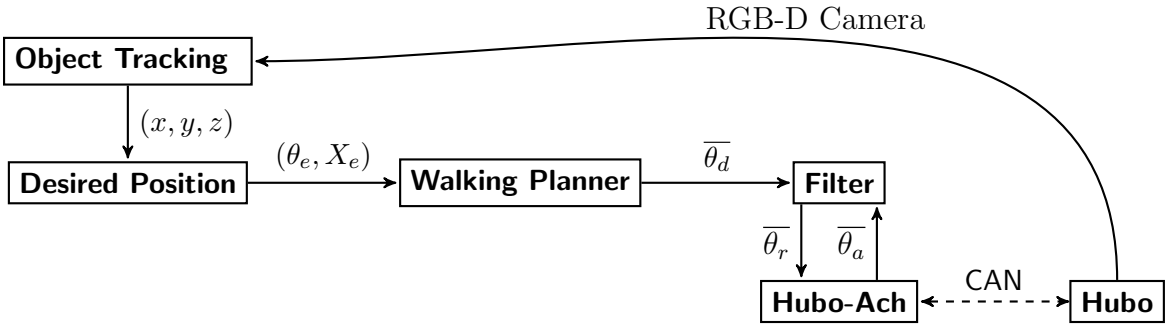


Figure 5.12: The (x, y, z) work space position of the objet is found via HSV tracking. The rotation error θ_e and distance of the object from the projection of the robot onto the ground X_e is sent to the *walking planner*. The walking planner decides if it has to turn or walk forwards. The robot will stop when it is within 0.2 m of the object and facing it within an error of ± 0.02 rad.

5.2.1 Tracking Using Vision

Using a the well know HSV (Hue Saturation Value) tracking algorithm. HSV tracking was implemented via the use of built in libraries in OpenCV[51]. The depth is found by averaging the distance data from the tracked location of the HSV color on the RGB-D device.

Fig. 5.12 shows the block diagram of the full system. The (x, y, z) work space position of the objet is found via HSV tracking. The rotation error θ_e and distance of the object from the projection of the robot onto the ground X_e is sent to the *walking planner*. The walking planner decides if it has to turn or walk forwards. The robot will stop when it is within 0.2 m of the object and facing it within an error of ± 0.02 rad. The *walking planner* is a state machine with the following characteristics:

$$f(\theta_e, X_e) = \begin{cases} \theta_e > 0.3\ rad & THEN \quad turn\ 0.3\ rad \\ \hline \theta_e < -0.3 & THEN \quad turn\ -0.3\ rad \\ \hline 0.3\ rad > \theta_e > 0.02\ rad & \\ OR & THEN \quad turn\ \theta_e\ rad \\ -0.02 < \theta_e < -0.3\ rad & \\ \hline -0.02 < \theta_e < 0.02\ rad & \\ AND & THEN \quad step\ forward\ 0.1\ m \\ X_e > 0.2\ m & \\ \hline else & THEN \quad stop \end{cases} \tag{5.1}$$

Video: http://danlofaro.com/phd/tracking/#HsvTracking

Figure 5.13: 3D Object tracking using HSV color matching and an RGB-D camera to gain depth information.

5.2.2 Visual servoing during full-body locomotion task

Hubo using Hubo-Ach to walk and track a blue box. The robot will walk towards the blue box until it is within 0.2 m at which point it will stop. If the box moves, the robot will turn to track the box. It is tracking the box in work-space via an RGBD camera and the HSV tracking method discribed in Section 5.2.1. Section 5.1 discribes the method used for the walking task. Fig. 5.14 shows the robot completing this task.

http://danlofaro.com/phd/tracking/#TrackingAndWalking

Figure 5.14: Hubo using Hubo-Ach to walk and track a blue box. The robot will walk towards the blue box until it is within 0.2 m at which point it will stop. If the box moves, the robot will turn to track the box.

5.3 Active Damping

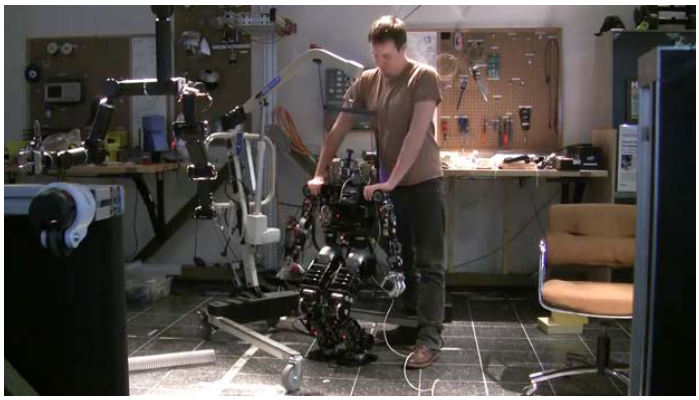
Using feedback from the force-torque sensors the Hubo-Ach controller adds compliance to the legs via active damping. Fig. 5.15 shows as the user pushes down on the robot the force is detected by the force-torque (FT) sensors. This then modifies the joint commands such that the center of mass (CoM) acts like there is an over-damped spring-damper system between it and mechanical ground.

Python Source Code for Active Damping using force-torque sensors

```
#!/usr/bin/env python
# /* -*-  indent-tabs-mode:t; tab-width: 8; c-basic-offset: 8  -*- */
import hubo_ach as ha
import ach
import sys
import time
from ctypes import *

# Open Hubo-Ach feed-forward and feed-back (reference and state) channels
s = ach.Channel(ha.HUBO_CHAN_STATE_NAME)
r = ach.Channel(ha.HUBO_CHAN_REF_NAME)
s.flush()
r.flush()

# feed-forward will now be refered to as "state"
state = ha.HUBO_STATE()

# feed-back will now be refered to as "ref"
ref = ha.HUBO_REF()

g = 0.4
kz = 0.28

imax = 10
delta = g/imax
for i in range(1, imax):
  ref.ref[ha.RAP] = -delta*i
  ref.ref[ha.LAP] = ref.ref[ha.RAP]
  ref.ref[ha.RKN] = 2*delta*i
  ref.ref[ha.LKN] = ref.ref[ha.RKN]
  ref.ref[ha.RHP] = -delta*i
```

```
  ref.ref[ha.LHP] = ref.ref[ha.RHP]


  r.put(ref)
  time.sleep(0.1)


d = 0.0
L = 5.0
while True:
  # Get the current feed-forward (state)
  [statuss, framesizes] = s.get(state, wait=False, last=False)
  ft = (state.ft[ha.HUBO_FT_R_FOOT].f_z + state.ft[ha.HUBO_FT_L_FOOT].f_z)/2.0/200.0
  dt = kz*ft
  d = (d*(L-1.0)+dt)/L


  if d > 0.4:
    d = 0.4
  if d < -0.4:
    d = -0.4


  ref.ref[ha.RAP] = -(g+d)
  ref.ref[ha.LAP] = ref.ref[ha.RAP]
  ref.ref[ha.RKN] = 2*(g+d)
  ref.ref[ha.LKN] = ref.ref[ha.RKN]
  ref.ref[ha.RHP] = ref.ref[ha.RAP]
  ref.ref[ha.LHP] = ref.ref[ha.RHP]


#  ref.ref[ha.REB] = g + ky*state.ft[ha.HUBO_FT_R_HAND].m_y
  print 'New Ref: ', ref.ref[ha.RKN], ' d = ', d, ' dt = ', dt


  # Write to the feed-forward channel
  r.put(ref)
```

```
  time.sleep(0.05)


# Close the connection to the channels
r.close()
s.close()
```



http://danlofaro.com/phd/activedamping/

Figure 5.15: Using feedback from the force-torque sensors the Hubo-Ach controller adds compliance to the legs via active damping.

6. Conclusion

This work shows the successful creation of a *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots*. It was shown that the Hubo-Ach system works as the unifying algorithmic framework for the three tier infrastructure described in Section 1.2. This means Hubo-Ach works with all three tiers including:

- Rapid Prototype (RP) phase with zero cost to entry (OpenHubo Platform Section 2.4.3)
- Test and Evaluation (T&E) phase with low cost to entry (Mini-Hubo Platform Section 2.4.2)
- Verify and Validate (V&V) phase with lease-time cost to entry (Hubo Platform Section 2.4.1)

The three tier infrastructure was used to enable the robot to throw a ball. This resulted in a unique algorithm for end-effector velocity control called Sparse Reachable Maps (SRM)(Section K.1). One end-effector velocity control method was used in a live throwing experiment at a baseball game (Section L.1). The challenges from this successful experiment was answered by the creation of the following controllers.

- *Challenge: Aiming the throw:* Answer: Visual seroving while performing full body locomotive task (Section 5.2)
- *Challenge: Safely landing when throwing:* Answer: Active damping via force-torque feedback (Section 5.3)

The Hubo-Ach system was verified under many circumstances including:

- Real-time closed form inverse kinematic controller (Section 3.6)
- Full body locomotive task of turning a valve (Section 3.6.1)
- Full body locomotive task of walking (Section 5.1.2)

Hubo-Ach was independently validated by other researcher through the examples of:

- Door opening (Section 3.8)
- Dynamic walking (Appendix I)

A study/survey done on how well the Hubo-Ach system performs as a *unifying algorithmic framework* returned positive results (Section M). The result is the creation of a truly *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots*.

6.1 Future Work

Future work includes applying the framework to the DRC-Hubo for the DARPA Robot Challenge. In addition an over arching goal is to implement Hubo-Ach on other high DOF robots such as HRP-2, Baxter etc. thus creating a truly *Unified Algorithmic Framework for High Degree of Freedom Complex Systems and Humanoid Robots*. This will allow for a greater ease of controller sharing and increase positive research output.

Bibliography

[1] N. Dantam, D. Lofaro, A. Hereid, P. Oh, A. Ames, and M. Stilman, "Reliable software for humanoid robots," in *IEEE Robotics and Automation Magazine*, 2013.

[2] M. Grey, N. Dantam, M. Stilman, and D. Lofaro, "Multi-process architecture for robust control the hubo2+ robot," in *IEEE International Conference on Technologies for Practical Robot Applications*, 2013.

[3] N. Alunni, C. Phillips-Graffin, H. Suay, D. Lofaro, and D. Berenson, "Toward a user-guided manipulation framework for high-dof robots with limited communication," in *IEEE International Conference on Technologies for Practical Robot Applications*, 2013.

[4] R. O'Flaherty, P. Vieira, G. M.X., 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*, 2013.

[5] Y. Kim, D. Lofaro, B. A., and D. Grunberg, "Towards a musically-aware humanoid for interactive music performance," in *EURASIP Journal on Audio, Speech, and Music Processing, 2011*, 2011.

[6] D. K. Grunberg, D. M. Lofaro, P. Y. Oh, and Y. E. Kim, "Robot audition and beat identification in noisy environments," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, sept. 2011.

[7] D. Lofaro, D. Grunberg, P. Oh, Y. Kim, and J. Oh, "Design of humanoids as interactive musical participants," in *International Association of Science and Technology (IASTED), 2011 International Conference on Robotics*, 2011.

[8] D. Lofaro, P. Oh, J. Oh, and Y. Kim, "Interactive musical participation with humanoid robots through the use of novel musical tempo and beat tracking techniques in the absence of auditory cues," in *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, dec. 2010.

[9] D. Lofaro, C. Sun, and P. Oh, "Humanoid pitching at a major league baseball game: Challenges, approach, implementation and lessons learned," in *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*, 2012.

[10] D. Lofaro, R. Ellenberg, and P. Oh, "Interactive games with humanoids: Playing with jaemi hubo," in *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, 2010.

[11] Y. Zhang, J. Luo, K. Hauser, R. Ellenberg, P. Oh, H. Park, M. Paldhe, and G. Lee, "Motion planning of ladder climbing for humanoid robots," in *IEEE International Conference on Technologies for Practical Robot Applications*, 2013.

[12] M. Zucker, Y. Jun, B. Killen, T. Kim, and P. Oh, "Continuous trajectory optimization for autonomous humanoid door opening," in *IEEE International Conference on Technologies for Practical Robot Applications*, 2013.

[13] R. OFlasherty, 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*, 2013.

[14] J. Youngbum and P. Oh, "A 3-tier infrastructure: Virtual-, mini-, online-hubo stair climbing as a case study," in *International Association of Science and Technology for Development-Robo2011, Pittsburgh, PA, USA*, 2011.

[15] D. Lofaro, R. Ellenberg, P. Oh, and J. Oh, "Humanoid throwing: Design of collision-free trajectories with sparse reachable maps," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, 2012, pp. 1519–1524.

[16] R. Diankov, "Automated construction of robotic manipulation programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010.

[17] J.-H. Oh, D. Hanson, W.-S. Kim, I. Y. Han, J.-Y. Kim, and I.-W. Park, "Design of android type humanoid robot albert hubo," in *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, 2006, pp. 1428–1433.

[18] E. Ackerman, "Video friday: Hubo and valves, uavs and lasers, and one very lucky parrot," in *IEEE Spectrum Blogs*, 2012, pp. http://spectrum.ieee.org/automaton/robotics/robotics–hardware/video–friday–8 562 746.

[19] R. M. Sherbert and P. Oh, "Conductor: A controller development framework for high degree of freedom systems," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, 2011, pp. 1022–1029.

[20] N. Dantam and M. Stilman, "Robust and efficient communication for real-time multi-process robot software," in *International Conference on Humanoid Robots (Humanoids)*, 2012.

[21] D. Lofaro and P. Oh, "Humanoid throws inaugural pitch at major league baseball game: Challenges, approach, implementation and lessons learned," in *Ubiquitous Robots and Ambient Intelligence (URAI), 2012 9th International Conference on*, 2012, pp. 153–157.

[22] K. Gadeyne, T. Lefebvre, and H. Bruyninckx, "Bayesian hybrid model-state estimation applied to simultaneous contact formation recognition and geometrical parameter estimation," *The International Journal of Robotics Research*, vol. 24, no. 8, pp. 615–630, 2005.

[23] J. Jackson, "Microsoft robotics studio: A technical introduction," *Robotics Automation Magazine, IEEE*, vol. 14, no. 4, pp. 82–87, 2007.

[24] *ROBOTC LEGO MINDSTORMS NXT*. Betascript Publishing, 2009.

[25] MATLAB, *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[26] G. W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, 2nd ed. McGraw-Hill School Education Group, 1997.

[27] W.-T. Lee, T.-Y. Wu, M.-Y. Chen, Y.-B. Wang, H.-Y. Lin, and K.-H. Liao, "Research of multi-thread applications for real-time control systems on humanoid robot embedded platforms," in *SICE Annual Conference 2010, Proceedings of*, 2010, pp. 2279–2286.

[28] L. Rai and S.-J. Kang, "Multi-thread based synchronization of locomotion control in snake robots," in *Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on*, 2005, pp. 559–562.

[29] Z. Qin and J. Gu, "Multi-thread technology based autonomous underwater vehicle," in *Control and Automation (ICCA), 2010 8th IEEE International Conference on*, 2010, pp. 898–903.

[30] F. Kanehiro, H. Hirukawa, and S. Kajita, "Openhrp: Open architecture humanoid robotics platform." *I. J. Robotic Res.*, vol. 23, no. 2, pp. 155–165, 2004. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijrr/ijrr23.html#KanehiroHK04

[31] S. Aramaki, H. Shirouzu, and K. Kurashige, "Control program structure of humanoid robot," in *IECON 02 [Industrial Electronics Society, IEEE 2002 28th Annual Conference of the]*, vol. 3, 2002, pp. 1796–1800 vol.3.

[32] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, *ROS: an open-source Robot Operating System*, 2009.

[33] Webots, "http://www.cyberbotics.com," commercial Mobile Robot Simulation Software. [Online]. Available: http://www.cyberbotics.com

[34] W. Stevens and S. Rago, *Advanced programming in the Unix environment*, ser. Addison-Wesley professional computing series. Addison-Wesley, 2005. [Online]. Available: http://books.google.com/books?id=D_VQAAAAMAAJ

[35] 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*, 2005, pp. 321–326.

[36] D. M. Lofaro, R. Ellenberg, P. Oh, and J.-H. Oh, "Humanoid throwing: Design of collision-free trajectories with sparse reachable maps," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, oct. 2012.

[37] R. Ellenberg, R. Sherbert, P. Oh, A. Alspach, R. Gross, and J. Oh, "A common interface for humanoid simulation and hardware," in *Humanoid Robots, 10th IEEE-RAS International Conference on*, 2010.

[38] R. Ellenberg, D. Grunberg, P. Oh, and Y. Kim, "Using miniature humanoids as surrogate research platforms," in *Humanoid Robots, 9th IEEE-RAS International Conference on*, dec. 2009.

[39] D. Lofaro, R. Ellenberg, P. Oh, and J. Oh, "Humanoid throwing: Design of collision-free trajectories with sparse reachable maps," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, oct. 2012.

[40] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[41] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1999.

[42] G. Pratt. (2012, Oct.) Darpa robotics challenge. [Online]. Available: http://www.theroboticschallenge.org/

[43] D. Berenson, S. Srinivasa, and J. Kuffner, "Task space regions: A framework for pose-constrained manipulation planning," *International Journal of Robotics Research (IJRR)*, vol. 30, no. 12, pp. 1435 – 1460, October 2011.

[44] M. Ali, H. Park, and C. S. G. Lee, "Closed-form inverse kinematic joint solution for humanoid robots," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 2010, pp. 704–709.

[45] D. Peiper, *The Kinematics of Manipulators Under Computer Control*. Stanford University California Department of Computer Science Defense Technical Information Center, 1968. [Online]. Available: http://books.google.com/books?id=g4tqNwAACAAJ

[46] K. Fu, R. González, and C. Lee, *Robotics: control, sensing, vision, and intelligence*, ser. McGraw-Hill series in CAD/CAM robotics and computer vision. McGraw-Hill, 1987. [Online]. Available: http://books.google.com/books?id=VkdSAAAAMAAJ

[47] R. O'Flaherty, P. Vieira, G. M.X., P. Oh, A. Bobick, M. Egerstedt, and M. Stilman, "Computing the analytical inverse kinematics for the arms and legs of hubo2+," in *Tech. Rep. GT-GOLEM-2013-001, Georgia Institute of Technology, Atlanta, GA*, 2013.

[48] R. Paul and B. Shimano, "Kinematic control equations for simple manipulators," in *Decision and Control including the 17th Symposium on Adaptive Processes, 1978 IEEE Conference on*, vol. 17, 1978, pp. 1398–1406.

[49] I.-W. Park, J.-Y. Kim, and J.-H. Oh, "Online biped walking pattern generation for humanoid robot khr-3(kaist humanoid robot - 3: Hubo)," in *Humanoid Robots, 2006 6th IEEE-RAS International Conference on*, 2006, pp. 398–403.

[50] D. Lofaro, J. Luo, K. Hauser, and D. Berenson, "Darpa robotics challenge drc-hubo team hack-a-thon," Worchester Polytechnic Institute, Interactive Session, April 2013.

[51] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.

[52] R. King, J. Rowland, W. Aubrey, M. Liakata, M. Markham, L. Soldatova, K. Whelan, A. Clare, M. Young, A. Sparkes, S. Oliver, and P. Pir, "The robot scientist adam," *Computer*, vol. 42, no. 8, pp. 46–54, 2009.

[53] M. Hirose and T. Takenaka, "Development of humanoid robot asimo," Vol. 13, No. 1., pp. 1-6 2001.

[54] D. Wooden, M. Malchano, K. Blankespoor, A. Howardy, A. Rizzi, and M. Raibert, "Autonomous navigation for bigdog," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, 2010, pp. 4736–4741.

[55] È. Coste-Manière, L. Adhami, R. Severac-Bastide, A. Lobontiu, J. K. S. Jr., J.-D. Boissonnat, N. Swarup, G. Guthart, É. Mousseaux, and A. Carpentier, "Optimized port placement for the totally endoscopic coronary artery bypass grafting using the da vinci robotic system," in *ISER*, 2000, pp. 199–208.

[56] W. Walter, "An electromechanical animal, dialectica," Vol. 4: 42 to 49 1950.

[57] B. Scassellati, "Eye finding via face detection for a foveated, active vision system," in *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98*, 1998, pp. 969–976.

[58] 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, 2002, pp. 2478–2483 vol.3.

[59] J. Mailisto, J. Sorvari, and H. Koivo, "Identification of the first joint of the puma robot," in *Industrial Electronics, Control and Instrumentation, 1991. Proceedings. IECON '91., 1991 International Conference on*, 1991, pp. 1095–1099 vol.2.

[60] D. Grunberg, R. Ellenberg, Y. Kim, and P. Oh, "From robonova to hubo: Platforms for robot dance," in *Progress in Robotics*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2009, vol. 44, pp. 19–24.

[61] P. Springer, "Contemporary world issues: Military robots and drones," ABC-CLIO LLC, 2013.

[62] B. Siciliano and O. Khatib, "Springer handbook of robotics," Springer-Verlag Berlin Heidelberg, 2008.

[63] P. Oh, "Team drc-hubo," Defense Advanced Research Projects Administration, Robotics Challenge Kickoff Meeting Presentation, October 2012.

[64] M. Vukobratovic and B. Borovac, "Zero-moment point - thirty five years of its life," *I. J. Humanoid Robotics*, vol. 1, no. 1, pp. 157–173, 2004.

[65] M. Vukobratovic and J. Stepanenko, "On the stability of anthropomorphic systems," *Mathematical Biosciences*, 1972.

[66] B.-K. Cho, S.-S. Park, and J. ho Oh, "Controllers for running in the humanoid robot, hubo," in *Humanoid Robots, 2009. Humanoids 2009. 9th IEEE-RAS International Conference on*, dec. 2009.

[67] Y. Jun, R. Ellenberg, and P. Oh, "Realization of miniature humanoid for obstacle avoidance with real-time zmp preview control used for full-sized humanoid," in *Humanoid Robots, 10th IEEE-RAS International Conference on*, dec. 2010.

[68] W. Mori, J. Ueda, and T. Ogasawara, "1-dof dynamic pitching robot that independently controls velocity, angular velocity, and direction of a ball: Contact models and motion planning," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, may 2009.

[69] T. Senoo, A. Namiki, and M. Ishikawa, "High-speed throwing motion based on kinetic chain approach," in *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, sept. 2008.

[70] N. Kato, K. Matsuda, and T. Nakamura, "Adaptive control for a throwing motion of a 2 dof robot," in *Advanced Motion Control, 1996. AMC '96-MIE. Proceedings., 1996 4th International Workshop on*, mar 1996.

[71] K. M. Lynch and M. T. Mason, "Dynamic nonprehensile manipulation: Controllability, planning, and experiments," *International Journal of Robotics Research*, 1997.

[72] T. Nakamura, "Search guided by skill in motion planning using dynamic programming," in *Advanced Motion Control, 1996. AMC '96-MIE. Proceedings., 1996 4th International Workshop on*, mar 1996.

[73] A. Sato, O. Sato, N. Takahashi, and M. Kono, "Trajectory for saving energy of a direct-drive manipulator in throwing motion," *Artificial Life and Robotics*, 2007.

[74] H. Frank, A. Mittnacht, T. Moschinsky, and F. Kupzog, "1-dof-robot for fast and accurate throwing of objects," in *Emerging Technologies Factory Automation, 2009. ETFA 2009. IEEE Conference on*, 2009, pp. 1–7.

[75] R. Thandiackal, C. Brandle, D. Leach, A. Jafari, and F. Iida, "Exploiting passive dynamics for robot throwing task," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, 2012, pp. 2443–2448.

[76] H. Miyashita, T. Yamawaki, and M. Yashima, "Control for throwing manipulation by one joint robot," in *Robotics and Automation, 2009. ICRA '09. IEEE International Conference on*, 2009, pp. 1273–1278.

[77] O. Yuuki, K. Yamada, and N. Kubota, "Trajectories tracing for a pitching robot based on human recognition," in *Computational Intelligence in Robotics and Automation (CIRA), 2009 IEEE International Symposium on*, 2009, pp. 252–257.

[78] T. Frank, U. Janoske, A. Mittnacht, and C. Schroedter, "Automated throwing and capturing of cylinder-shaped objects," in *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, 2012, pp. 5264–5270.

[79] E. Yedeg and E. Wadbro, "Optimal control of a ball pitching robot," in *Methods and Models in Automation and Robotics (MMAR), 2012 17th International Conference on*, 2012, pp. 456–456.

[80] H. Frank, T. Frank, A. Mittnacht, and C. Sichau, "A bioinspired 2-dof throwing robot," in *AFRICON, 2011*, 2011, pp. 1–6.

[81] S. Haddadin, K. Krieger, M. Kunze, and A. Albu-Schaffer, "Exploiting potential energy storage for cyclic manipulation: An analysis for elastic dribbling with an anthropomorphic robot," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, sept. 2011.

[82] Z. Wang, C. H. Lampert, K. Mulling, B. Scholkopf, and J. Peters, "Learning anticipation policies for robot table tennis," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, sept. 2011.

[83] S. Schaal, S. Vijayakumar, S. D'Souza, A. Ijspeert, and J. Nakanishi, "Real-time statistical learning for robotics and human augmentation," in *International Symposium of Robotics Research (ISRR01)*. Springer, 2001.

[84] J. Hu, M. Chien, Y. Chang, S. Su, and C. Kai, "A ball-throwing robot with visual feedback," in *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, 2010.

[85] J. Kim, "Motion planning of optimal throw for whole-body humanoid," in *Humanoid Robots (Humanoids), 2010 10th IEEE-RAS International Conference on*, dec. 2010.

[86] ——, "Optimization of throwing motion planning for whole-body humanoid mechanism: Sidearm and maximum distance," *Mechanism and Machine Theory*, 2011.

[87] M. Vukobratovic, "How to control artificial anthropomorphic systems," *Systems, Man and Cybernetics, IEEE Transactions on*, 1973.

[88] J. Zannatha and R. Limon, "Forward and inverse kinematics for a small-sized humanoid robot," in *Electrical, Communications, and Computers, 2009. CONIELECOMP 2009. International Conference on*, 2009, pp. 111–118.

[89] H. Zhang and R. Paul, "A parallel inverse kinematics solution for robot manipulators based on multiprocessing and linear extrapolation," *Robotics and Automation, IEEE Transactions on*, vol. 7, no. 5, pp. 660–669, 1991.

[90] D. Manocha and J. Canny, "Efficient inverse kinematics for general 6r manipulators," *Robotics and Automation, IEEE Transactions on*, vol. 10, no. 5, pp. 648–657, 1994.

[91] P. Chang, "A closed-form solution for inverse kinematics of robot manipulators with redundancy," *Robotics and Automation, IEEE Journal of*, vol. 3, no. 5, pp. 393–403, 1987.

[92] D. Berenson, S. Srinivasa, D. Ferguson, and J. Kuffner, "Manipulation planning on constraint manifolds," in *IEEE International Conference on Robotics and Automation (ICRA '09)*, May 2009.

[93] A. Guez and Z. Ahmad, "Solution to the inverse kinematics problem in robotics by neural networks," in *Neural Networks, 1988., IEEE International Conference on*, 1988, pp. 617–624 vol.2.

[94] E. Oyama and S. Tachi, "Modular neural net system for inverse kinematics learning," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 4, 2000, pp. 3239–3246 vol.4.

[95] S. Kieffer, V. Morellas, and M. Donath, "Neural network learning of the inverse kinematic relationships for a robot arm," in *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, 1991, pp. 2418–2425 vol.3.

[96] E. Oyama and S. Tachi, "Inverse kinematics learning by modular architecture neural networks," in *Neural Networks, 1999. IJCNN '99. International Joint Conference on*, vol. 3, 1999, pp. 2065–2070 vol.3.

[97] Z. Bingul, H. M. Ertunc, and C. Oysu, "Comparison of inverse kinematics solutions using neural network for 6r robot manipulator with offset," in *Computational Intelligence Methods and Applications, 2005 ICSC Congress on*, 2005, pp. 5 pp.–.

[98] E. Oyama, N. Y. Chong, A. Agah, and T. Maeda, "Inverse kinematics learning by modular architecture neural networks with performance prediction networks," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 1, 2001, pp. 1006–1012 vol.1.

[99] C. Qin and M. Carreira-Perpinan, "Trajectory inverse kinematics by conditional density modes," in *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, 2008, pp. 1979–1986.

[100] J. Burdick, "On the inverse kinematics of redundant manipulators: characterization of the self-motion manifolds," in *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, 1989, pp. 264–270 vol.1.

[101] G. Tevatia and S. Schaal, "Inverse kinematics for humanoid robots," in *Robotics and Automation, 2000. Proceedings. ICRA '00. IEEE International Conference on*, vol. 1, 2000, pp. 294–299 vol.1.

[102] A. D'Souza, S. Vijayakumar, and S. Schaal, "Learning inverse kinematics," in *Intelligent Robots and Systems, 2001. Proceedings. 2001 IEEE/RSJ International Conference on*, vol. 1, 2001, pp. 298–303 vol.1.

[103] K. Tchon, "Optimal extended jacobian inverse kinematics algorithms for robotic manipulators," *Robotics, IEEE Transactions on*, vol. 24, no. 6, pp. 1440–1445, 2008.

[104] W. A. Wolovich and H. Elliott, "A computational technique for inverse kinematics," in *Decision and Control, 1984. The 23rd IEEE Conference on*, dec. 1984.

[105] S. Fleisig, R. F. Escamilla, J. R. Andrews, T. Matsuo, Y. Satterwhite, and S. W. Barrentine, "Kinematic and kinetic comparison between baseball pitching and football passing," *Journal of Applied Biomechanics*, 1996.

[106] W. Barrentine, T. Matsuo, R. F. Escamilla, G. S. Fleisig, and J. R. Andrews, "Kinematic analysis of the wrist and forearm during baseball pitching," *Journal of Applied Biomechanics*, jan 1998.

[107] Y. Mochizuki, T. Matsumoto, S. Inokuchi, and K. Omura, "Computer simulation of the effect of ball mass and shape to upper limb in baseball pitching," *Theoretical and Applied Mechanics*, 1998.

[108] A. Uesaki, Y. Mochizuki, T. Matsuo, K. Hashizume, K. Omura, and S. Inokuchi, "Computer simulation for dynamics analysis of pedaling motion on lower limbs in a racing cycle," *Theoretical and Applied Mechanics*, 1999.

[109] Q. Huang, Z. Peng, W. Zhang, L. Zhang, and K. Li, "Design of humanoid complicated dynamic motion based on human motion capture," in *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, aug. 2005.

[110] S.Pollard, J. Hondgins, M.J.Riley, and C. Atkeson, "Adapting human motion for the control of a humanoid robot," in *In Proc. of IEEE International Conference on Robotics and Automation*, 2002.

[111] S. Gaertner, M. Do, T. Asfour, R. Dillmann, C. Simonidis, and W. Seemann, "Generation of human-like motion for humanoid robots based on marker-based motion capture data," *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, june 2010.

[112] D. Lofaro and P. Oh, in *Humanoid Throws Inaugural Pitch at Major League Baseball Game: Challenges, Approach, Implementation and Lessons Learned*, nov. 2012.

[113] Z.-Y. Ying, Y.-G. Xi, and Z.-H. Zhang, "Test of the reachability of a robot to an object," in *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, 1989.

[114] Z. Xue and R. Dillmann, "Efficient grasp planning with reachability analysis," in *Intelligent Robotics and Applications*, ser. Lecture Notes in Computer Science, H. Liu, H. Ding, Z. Xiong, and X. Zhu, Eds. Springer Berlin / Heidelberg, 2010.

[115] R. Geraerts and M. Overmars, "Reachability analysis of sampling based planners," in *Robotics and Automation, 2005. ICRA 2005. Proceedings of the 2005 IEEE International Conference on*, april 2005.

[116] M. Vande Weghe, D. Ferguson, and S. Srinivasa, "Randomized path planning for redundant manipulators without inverse kinematics," in *Humanoid Robots, 2007 7th IEEE-RAS International Conference on*, dec. 2007.

Appendix A. Acronyms

Acronyms

AI	Artificial Intelligence
AIO	Asynchronous Input Output
CAD	Computer Aided Design
CAN	Controller Area Network
CBiRRT	Constrained Bi-directional Rapidly-exploring Random Tree
DARPA	Defense Advanced Research Projects Agency
DH	DenavitHartenberg
DOF	Degree of Freedom
DRC	DARPA Robotics Challenge
DSP	Double Support Phase
EEF	End Effector
FIFO	First In, First Out
FK	Forward Kinematics
HRI	Human Robot Interaction
HOL	Head of Line
IK	Inverse Kinematics
IO	Input Output
IPC	Inter Process Communication
JMC	Joint Motor Controller
KAIST	Korea Advanced Institute of Science and Technology
MPI	Message Passing Interface
MIRR	Major Research Infrastructure Recovery and Reinvestment
MLB	Major League Baseball
NSF	National Science Foundation
ODE	Open Dynamics Engine
PID	Proportional Integral Derivative
POSIX	Portable Operating System Interface
RP	Rapid Prototype
RT	Real-Time
ROS	Robot Operating System
SI Units	International System of Units
SSP	Single Support Phase
SRM	Sparse Reachable Map
T&E	Test and Evaluation
V&V	Verify and Validate
ZMP	Zero Moment Point

Appendix B. Hubo Joint Acronyms

Hubo Joint Acronyms

RHY	Right Hip Yaw	RHR	Right Hip Roll
RHP	Right Hip Pitch	RKN	Right Knee Pitch
RAP	Right Ankle Pitch	RAR	Right Ankle Roll
LHY	Left Hip Yaw	LHR	Left Hip Roll
LHP	Left Hip Pitch	LKN	Left Knee Pitch
LAP	Left Ankle Pitch	LAR	Left Ankle Roll
RSP	Right Shoulder Pitch	RSR	Right Shoulder Roll
RSY	Right Shoulder Yaw	REB	Right Elbow Pitch
RWY	Right Wrist Yaw	RWR	Right Wrist Roll
RWP	Right Wrist Pitch		
LSP	Left Shoulder Pitch	LSR	Left Shoulder Roll
LSY	Left Shoulder Yaw	LEB	Left Elbow Pitch
LWY	Left Wrist Yaw	LWR	Left Wrist Roll
LWP	Left Wrist Pitch		
NK1	Neck 1	NKY	Neck Yaw
NK2	Neck 2	WST	Trunk Yaw
RF1	Right Finger 1	RF2	Right Finger 2
RF3	Right Finger 3	RF4	Right Finger 4
RF5	Right Finger 5		
LF1	Left Finger 1	LF2	Left Finger 2
LF3	Left Finger 3	LF4	Left Finger 4
LF5	Left Finger 5		

Appendix C. Symbols

Symbols

Symbol	Definition	Units
L	Filter buffer length	N/A
N	Discrete time step	$sample$
T	Period	sec
T_R	Robot Real-Time Loop Period	sec
θ_a	Actual position of joint as measured from the encoders	rad
θ_c	Reference set to the actuator	rad
θ_d	Desired Reference before being set to Hubo-Ach FeedForward Cahnnel	rad
θ_e	Actuiator PID error	rad
θ_r	Desired reference on the Hubo-Ach FeedForward Channel	rad

Appendix D. Robots with the year they were created and their DOF

Robots with the year they were created and their DOF. A study of 180 robots from 1929 to the present.

Robot	DOF	year
Adam [52]	40	2009
Adelbrecht	2	1985
AIBO (Sony)	20	1999
AIBO MUTANT (Sony)	16	1999
AIBO Prototype (Sony)	17	1999
AISoy1	5	2010
Albert HUBO (KHR-3)	66	2005
Alice	3	1998
Allen	18	1986
Arachno-Bot	40	2011
ASIMO (Honda) [53]	26	2000
ASIMO R2 [53] (Honda) [53]	34	2005
ATHLETE (NASA)	36	2008
Beast (John Hopkins)	3	1960
Beetle (Mobile Land Mine)	2	1940
Big Trak	2	1979
BigDog [54]	16	2005
Biloid (ROBOTIS)	20	2007
BioHazard	4	1996
Blendo	2	1995
Boe-Bot	2	2001
Borgward	2	1942
Canadarm	6	1981
Chandrayaan-2 (NASA) (proposed)	15	2015
Chaos 2	3	1999
CHEETAH (BD)	15	2012
Choromet	30	2004
Cosmobot	3	1999
Cyberknife	8	1990
Da Vinci Surgical System [55]	28	1998
DARwin-OP (ROBOTIS)	20	2010
Dirt Dog (iRobot)	2	2010
Don Cuco El Guapo	28	1992

Dragon Runner	4	2002
DRDO Daksh	12	2008
E0 (Honda)	6	1986
E1 (Honda)	12	1987
E2 (Honda)	12	1989
E3 (Honda)	12	1991
E4 (Honda)	12	1991
E5 (Honda)	12	1992
E6 (Honda)	12	1993
Electrolux Trilobite	3	1996
Elise [56]	2	1949
Elmer [56]	2	1948
Entomopter	5	2000
ERS-110 AIBO (Sony)	19	1999
ERS-210 (Sony)	20	1999
ERS-220 (Sony)	16	2000
ERS-300 (Sony Latte and Macaron)	15	1999
ERS-311 (Sony Latte)	15	2001
ERS-312 (Sony Macaron)	15	2001
ERS-7 AIBO (Sony)	20	2003
ERS-7M2 AIBO (Sony)	20	2004
ERS-7M3 AIBO (Sony)	20	2005
FAMULUS (KUKA)	7	1973
Flame	20	2003
Freddy	3	1969
Freddy II	5	1973
FRIEND	9	2003
Gakutensoku	5	1929
Geminoid (Hiroshi Ishiguro)	30	2005
Geoff Peterson	20	2010
George	5	1949
Goliath	2	1944
Great Moments with Mr. Lincoln (Disney)	26	1965
HAL (Hybrid Assistive Limb)	35	2011
Hardiman (GE)	24	1965
HERO (Heathkit Educational Robot)	3	1982
HRP-1	28	1997
HRP-2 Promet	30	2002
HRP-2P	30	1998
HRP-3 Promet MK-II	42	2007
HRP-3P	36	2005

HRP-4C	42	2009
HRP-4C	34	2010
Hubo 2 (KHR-4)	40	2008
HUBO 2 Plus (KHR-4 Plus)	38	2011
Hypno-Disc	3	2006
INSECT	24	1992
IR 6/60 (KUKA)	6	1979
iRobot Create	2	2007
Kanguera	20	2007
Khepera	2	1991
KHR-0	12	2001
KHR-1 (KAIST)	21	2002
KHR-1 (Kondo Kagaku)	17	2004
KHR-2	41	2004
KHR-3 (HUBO)	41	2005
Kismet [57]	15	1998
Kobian	35	2009
Koolvac	3	2005
KR AGILUS (KUKA)	6	2012
KR QUANTEC (KUKA)	6	2010
KUKA titan (KUKA)	6	2007
LAURON I	25	1994
LAURON II	26	1995
LAURON III	26	1999
LAURON IV	27	2004
Legged Squad Support System	16	2009
Lewis	5	2002
LittleDog (BD)	12	2005
Looj	3	2008
Luna 2	0	1959
Lunokhod 1	11	1970
Lunokhod 2	11	1973
MANOI PF01	17	2007
Milton Bradley Playmate	2	1968
Mini-Hubo	22	2009
Modulus	16	1984
Nao (Aldebaran)	25	2004
Navlab (series)	5	1986
Neato XV	3	2010
Nomad 200 (N200)	2	1994
Nomad Rover (CMU/NASA)	8	1997

Omnibot	6	1985
Open PINO Platform	28	2006
Orazio	3	2004
P1 (Honda) [58]	30	1993
P2 (Honda) [58]	30	1996
P3 (Honda) [58]	28	1997
P4 (Honda) [58]	34	2000
PackBot (iRobot)	11	1998
Panic Attack	3	2003
PETMAN (BD)	30	2010
Plen	18	2007
Polly (MIT)	5	1993
PR2 (Willow Garage)	18	2010
PUMA (Westinghouse) [59]	6	1975
Push the Talking Trash Can (Disney Land)	2	1995
R.O.B. (Nintendo Robot)	4	1985
Ranger (iRobot)	3	2009
Razer	3	1998
RB5X	5	1983
RiSE (BD)	35	2006
Roadblock	3	1997
RoboBee	2	2013
RoboMop	2	2011
Robonaut (NASA)	40	2009
Robonaut 2 (NASA)	40	2010
Robonova [60]	16	2007
Roboreptile	11	2006
Robosapien	11	2005
Robosaurus	13	1989
RoboTuna (MIT)	6	1993
RoboTurb	5	1988
Roomba (iRobot)	3	2002
Ropid	30	2012
RuBot II	24	2011
Sarcoman	41	1995
Scarab Rover	11	2008
Seaglider (iRobot)	4	2008
Senster Philips	5	1970
Seropi (KITECH)	21	2010
Shadow Hand	20	2004
Shakey the robot	2	1966

Sojourner (NASA)	15	1997
Spirit/Opportunity (NASA)	30	2004
T.R.A.C.I.E.	2	1998
TALON (Foster-Miller)	7	2003
Teletank [61]	2	1930
TIOSS	6	1962
TOPIO Dio	28	2010
Topo	5	1983
Tornado	3	1999
Turtle (Hello World for Robots)	2	1949
UNIMATE (GM)	6	1954
Upuant Project	4	1992
UWA Telerobot	2	1994
Voyager 1 and Voyager 2	5	1977
VSR-2: Talos FG	26	2010
Walking truck (GE)	16	1968
Warrior (iRobot)	13	2008
Wheelbarrow	5	1972
Whegs	10	2006
WonderBorg	2	2000
XBC	3	2005
XM1216 (iRobot)	10	2009
youBot (KUKA)	10	2010

Appendix E. Increasing Degrees of Freedom

Degree of freedom (DOF) is the number of independent parameters that can be varied in a mechanical system. Simply put the DOF of a robot is the number independent joints the robot contains. In modern robot implementation each joint is controlled by an actuator. Each actuator is controlled by the controller. The more DOF the more complex the controller.

In the early 1930s simple two DOF robots were used by the soviets as explosive devices[61]. These robots were remote control and had no mind of their own. In 1948 William Grey Walter created the first autonomous robots *Tortoise Elmer* and *Elise*[56]. Each of these simple two DOF robots were programmed in hardware to go towards a light source. This was referred to as BEAM Robot (Biology, Electronics, Aesthetics, and Mechanics) because of how the hardware configuration mimicked the electrical connections in an animals brain. In later years robots were being programmed in software to help create the first industrial robot UNIMATE which was a 6 DOF arm created by General Motors in 1954[62]. Lunokhod 2, a soviet lunar rover which landed on the moon in 1973, was equipped with a laser ranging system and a TV camera. It contained 11 DOF and had automated systems onboard, however it was primarily a remote controlled vehicle. By 1986 Rodney Brooks, co-founder and CTO of iRobot Corp., created Allen, a 18 DOF humanoid robot. The complexity and number of DOF keeps on increasing. By 1997 Honda completed the 28 DOF P3, a early version of what will become ASIMO [58]. Today with the

presents of HUBO, ASMO and HRP-4C it is common place for a robot to contain upwards of 40 DOF. A study done on 180 robots from the early 20^{th} century to the present projects that by the year 2020 it will be as common to have a 70 DOF robot as it is to have a 40 DOF robot in 2013, see Fig. E.1. *The trend of increasing DOF in robots makes creating a control structure for these systems timely.*

These high DOF robots require complex control systems and strategies.

Creating controllers for these high degree of freedom complex systems is essential for development of the next generation of robots. Due to the inherent complexity and often high expense of these systems, controllers must be able to be tested and verified.

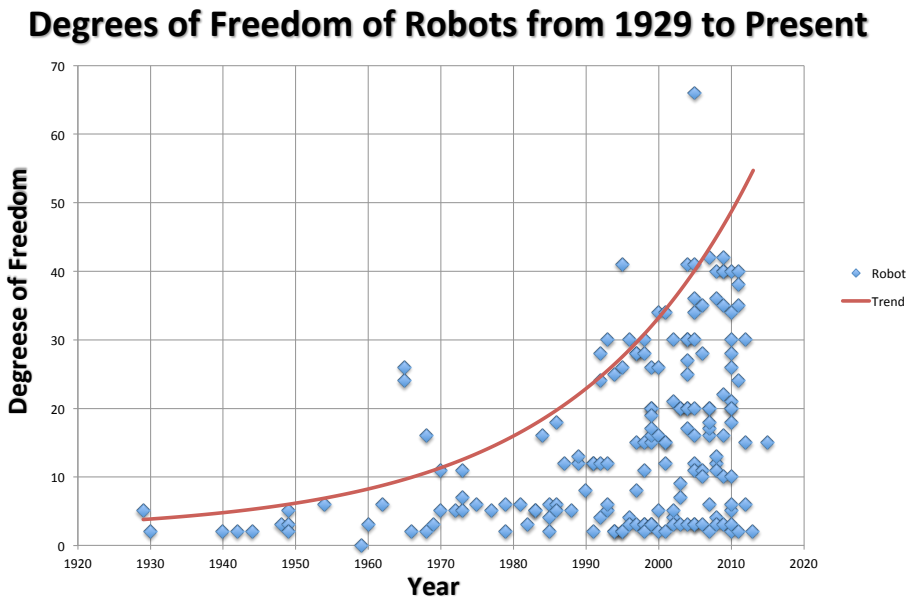


Figure E.1: Number of degrees of freedom for robots form 1929 to the present.

Appendix F. Inspiration: DARPA Robotics Challenge

In October 2012 DASL received officially become a Track-A team for the DRC. The team is now called DRC-Hubo. In December 2012 Hubo-Ach was chosen as the primary controller for the DRC-Hubo team. This would be another source of verification and validation of the Hubo-Ach system.

One of the keys to the team's success is collaboration. The DRC-Hubo team consists of Drexel University, WPI, Georgia Tech, University of Delaware, Swarthmore, Purdue, Ohio State (check that) and RAINBOW (a company that rose from the Hubo Lab at Korea Advanced Institute of Science and Technology (KAIST)). Each partner would be responsible with one event. Efforts will then combine creating one master controller that is capable of doing all the given tasks. Having this unified framework gives them the ability to share their controllers without having to integrate their code.

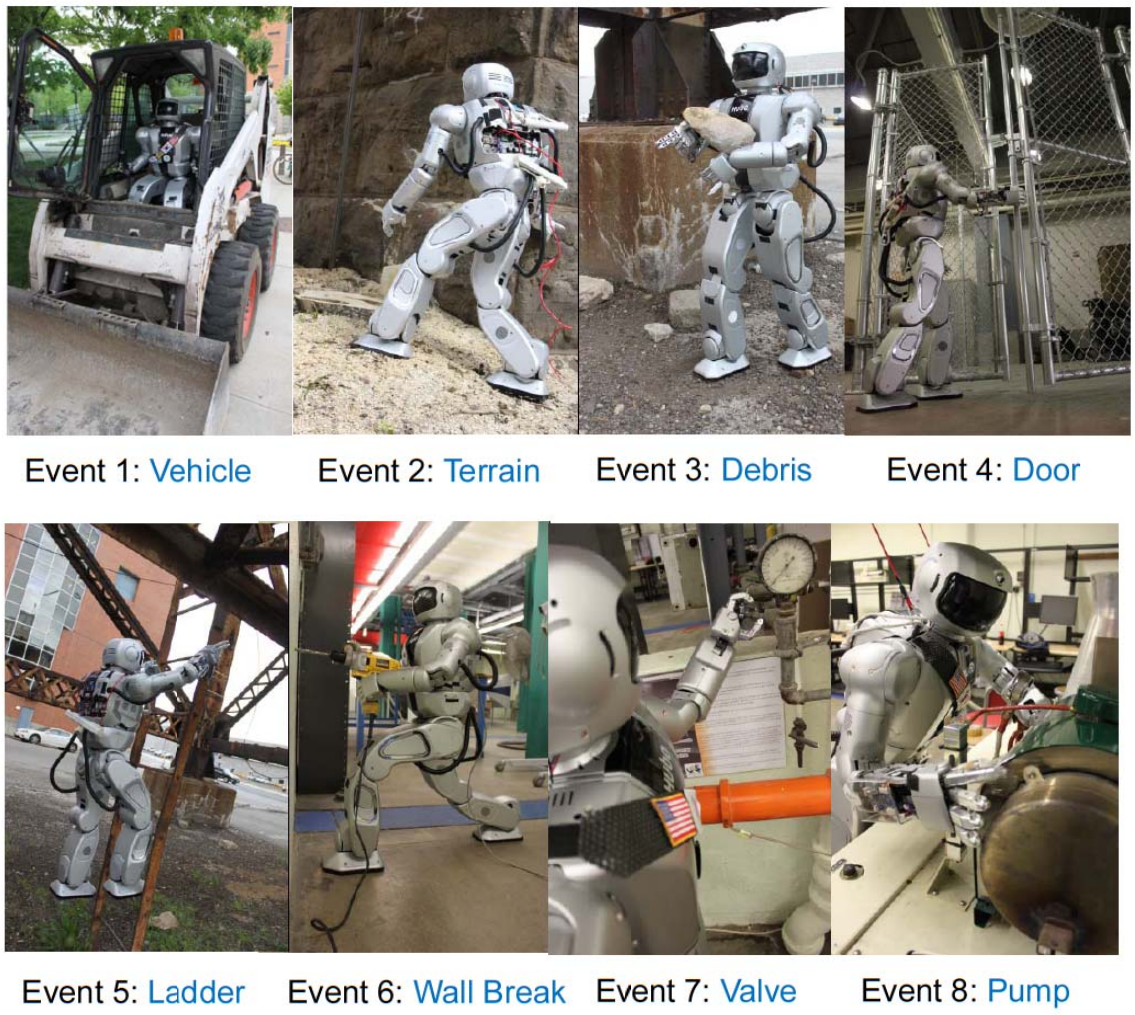


Figure F.1: DARPA Robot Challenge Events. Pictures depict the Hubo2+ (KHR-4) preforming the eight given tasks. The photographs are meant to help you *imagine* that the robot is capable of preforming these tasks. The events are - Event 1: Driving an un-modified human vehicle; Event 2: Walking over rough, un-even terrain; Event 3: Removing debris from regions of interest; Event 4: Opening and navigating through multiple doors and hallways; Event 5: Climb an industrial ladder; Event 6: Break through a wall using un-modified human tools; Event 7: Turn a valve; Event 8: Replace a pump (note: this was replaced by a hose insertion task). All photographs were staged and taken by Daniel M. Lofaro. Picture montage taken from Dr. Paul Oh's meeting to DARPA at the DRC Kickoff meeting, October 23-25, 2012.[63]

Appendix G. Balancing: Zero-Moment-Point (ZMP)

The past years of research in humanoids robotics has resulted in a stability criteria that must be followed for bipedal robots to stay stable. This is known as the Zero Moment Point criteria commonly referred to as ZMP [64]. ZMP is ubiquitous in the humanoid robotics community. The ZMP criteria states that a system is statically stable (balanced) if there is no moment acting on the connection between the end effectors touching the ground and the ground. This means that if the center of mass is over the support polygon there will be no moment. The support polygon is defined by the are formed by connecting the out most portions of the end effectors (typically feet) that are touching the ground and/or walls, rails etc. If the zero moment point, the location of the center of mass (COM) projected in the direction of gravity, is located within this support polygon then the system is considered statically stable. Fig. G.1 gives an example of the zero moment point on a bipedal robot in a single support phase and a double support phase.

Single Support Phase: The single support phase of a bipedal robot is when a single foot is touching the ground. This creates a smaller support polygon.

Double Support Phase: The double support phase of a bipedal robot is when two feed of a bipedal robot are on the ground. This creates a larger support polygon. In addition there is a stable path that the ZMP can move from above one foot to the other. This allows the robot to guarantee stability

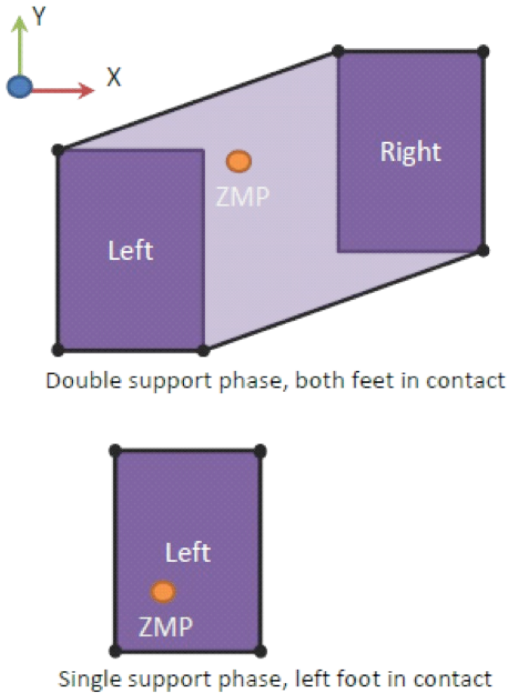


Figure G.1: Example of the zero moment point on a bipedal robot in a single support phase (bottom) and a double support phase (top). If the zero moment point, the location of the center of mass (COM) projected in the direction of gravity, is located within this support polygon then the system is considered statically stable.

while walking (static walking).

Appendix H. Balancing

Each of the methods used have to be stable through the motion in order for the system to be stable (i.e. not to fall down). The well known zero-moment-point (ZMP) criteria is what each method must adhere to in order to stay statically stable[65]. To handle perturbation an active balance controller was added. The active balance controller is applied on top of the pre-defined trajectories. Hubo is modeled as a single inverted pendulum with the center of mass (COM) located at length L from the ankle. The compliance of the robot is composed of a spring K and a damper C, see Fig. H.1. An IMU located at the COM gives the measured orientation.

The dynamic equation of the simplified model is assumed to be the same in both the sagittal and coronal plane.

$$mL^2\ddot{\theta} + C\dot{\theta} - K\theta = Ku \tag{H.1}$$

This can be linearized and made into the transfer function:

$$G(s) = \frac{\Theta(s)}{U(s)} = \frac{\frac{K}{mL^2}}{s^2 + \frac{C}{mL^2}s + \frac{K-mgL}{mL^2}} \tag{H.2}$$

Prior work on the model and controller for the Hubo by Cho et. al. calculated K=753 $\frac{Nm}{rad}$ and C=18 $\frac{Nm}{sec}$ using the free vibration response method[66].

The control law is as follows

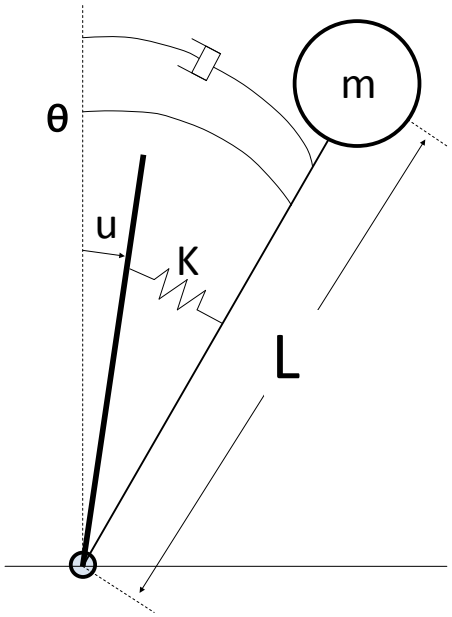


Figure H.1: Hubo modeled as a single inverted pendulum with COM located a distance L from

$$\theta_n^{x_a} = \theta_t^{x_a} + \left(K_p^x + sK_d^x\right)\left(\sum_{x \in t} \theta_t^x - \theta_c^x\right) \tag{H.3}$$

Where θ_t is the desired trajectory of the lower body (pitch or roll), x denotes pitch or roll and x_a denotes pitch or roll on the ankle. θ_c is the orientation of the center of mass in the global frame. θ_n is the resulting trajectory. K_p and K_d are the proportional and derivative gains. The resulting control allows for a stable stance even with perturbations from upper body motions.

Fig. H.3 shows the example of the Hubo balancing using the above method.

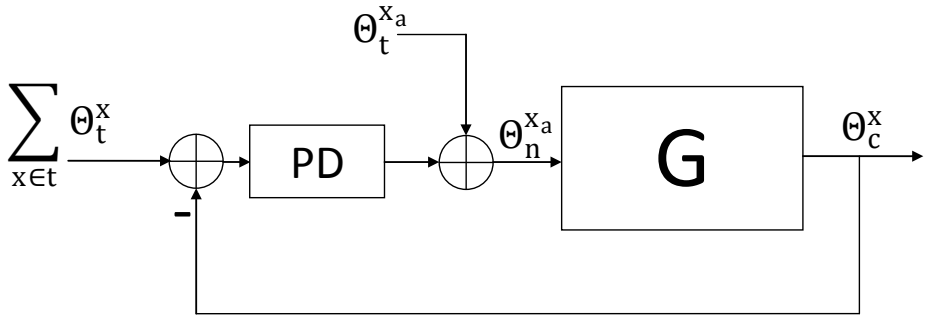
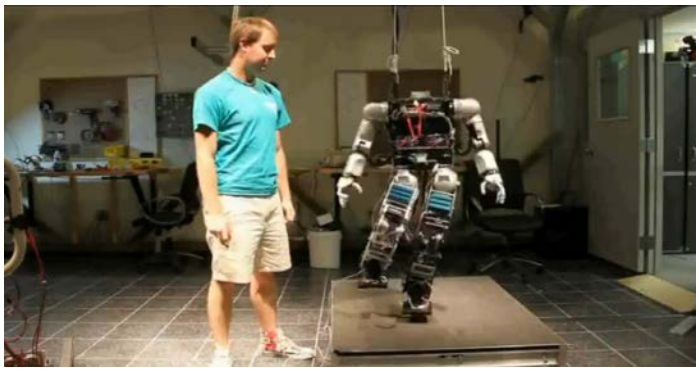


Figure H.2: Block diagram of the balance controller used to balance Hubo in this work.



Video: http://danlofaro.com/phd/balance/

Figure H.3: Hubo Balancing using method discribed in Section H

Appendix I. Hubo Dynamic Walking - Developed in 5 Days Using Hubo-Ach

Fig. I.1 shows Hubo2+ dynamic walking using Hubo-Ach as the primary controller. The standard ZMP walking algorithms were implemented by our partners Mike Sillman and Matt Zucker at Georgia Gech and Swarthmore respectively. All control was implemented using Daniel M. Lofaro's Hubo-Ach system.

Video: http://danlofaro.com/phd/walking/#Walking5Days

Figure I.1: Hubo dynamic walking using Hubo-Ach as the primary controller. The standard ZMP walking algorithms were implemented by our partners Mike Sillman and Matt Zucker at Georgia Gech and Swarthmore respectively. All control was implemented using Daniel M. Lofaro's Hubo-Ach system.

Appendix J. Kinematic Planning Background

This section gives brief background of the methods used in this document Section J.1 and J.2 gives the background for the methods used for inverse kinematics and throwing on high DOF robots. This is the background to the development of the control system that made Hubo throw the first pitch at a Major League Baseball game in 2012 as seen in Section 3 and multiple examples given in Section 3.5. Finally Section G gives a brief description of the Zero Moment Point criteria which is used for humanoid walking and shown in the examples in Section 3.5.

J.1 Kinematic Planning

Kinematic planning focuses on creating and testing valid trajectories for series kinematic manipulators. The focus of this research is on high degree of freedom (DOF), high-gain, position controlled mechanisms. High-gain position controlled mechanisms are the focus because the experimental platform used for this work is a that type of robot. This limits the work because it is crucial that the joint-space acceleration profile is correct or the system will over-torque and shutdown.

The works are chosen as it pertains to end-effector velocity control. Throwing and hitting are examples of end-effector velocity control. The goal is to have the end-effector moving at a specific rate in a specific direction. In most cases it demands whole-body coordination to achieve a desired end-effector velocity. Whole-body coordination is different for planted robots

and un-planted robots.
Fixed robots are robots where the base is attached to the ground or the base is significantly more massive then the manipulator. Planted robots do not have to worry about balance consternates.
Un-fixed robots are robots that have an manipulator that is not significantly lighter then the base. In addition the robot is not physically attached to the ground. This results in the robot needing to satisfy balance constraints. In the static case if the robot satisfies the zero moment point (ZMP) criteria it will remain stable [67]. When the manipulator moves quickly, as in the case of pitching or throwing, such upper-body motions if not coordinated with the lower-body, can cause the humanoid to lose balance.

J.2 End-Effector Velocity Control

End-effector velocity control (EEVC) is the act of moving your manipulator at a given speed through space at a given velocity. EEVC is being looked at as the mass of the end-effector does not change. Thus by controlling the velocity we also control the inertia. In addition I will be exploring EEVC as it pertains to manipulating objects. Through my research I have found that end-effector velocity control can be broken up into four major categories: *Time and location sensitive*, *location sensitive*, *time sensitive*, and *time and location insensitive*.

Time and Location Sensitive: If the velocity controler is time and location sensitive it means that your end effector needs

to have a given velocity at a specific time in a specific location or the task fales. Hitting a baseball with a bat is an example of *time and location sensitive* EEVC. If the bat has the correct velocity but not at the correct time it will not hit the ball or the ball will not go in the desired place. The same goes for if it does not have the correct location but does have the correct velocity. It is important to note that the manipulator only has instantaneous control over the object at the instant of contact. Other examples include playing the piano, hitting a tennis ball with a racquet, a moving soccer ball with a foot or any other task that requires to *hit* a *moving* object.

Location Sensitive: If the velocity controler is location sensitive it means that it only matters that the velocity occurs at a given location. The time it takes to reach that velocity will not effect the results. Hitting a nail with a hammer is a prime example of location sensitive EEVC. The nail is not moving but it does need to be hit in a given location with a given velocity. The vector of the velocity is determined by the required angle the nail needs to be hit at. In this example the nail is not time dependent and can be hit any time. Hitting it a $t = N$ or $t = N+1$ will not effect the results. It is important to note that the manipulator only has instantaneous control over the object at the instant of contact. Other examples of location sensitive end-effector velocity control are hitting a golf ball with a club, hitting a pool ball with the cue, and other activities that require a given location and direction of manipulation but are not time dependent.

Time Sensitive: If the location were the end-effector achieves a given velocity is not required to complete the task but the time when it happens is required it is considered *time sensitive* EEVC. This means that the end-effector can move in any region it desired as long as the end effector achieves a given velocity at a given time. The end-effecter's velocity can be dependent on the location achieved but the location is an independent variable and the velocity is the dependent variable. It is important to note that the manipulator control over the object during the entirety of the motion. This typically means that the manipulator is holding the object until the release stage. An example of this is throwing a baseball to first base to get someone out. Throwing the ball side arm, over arm, or even underarm does not matter as long at it is released at the correct time with the correct velocity to get it ball to the first-baseman to get the runner out. Other example of time sensitive EEVC are any other instance where an object is thrown within a given time.

Time and Location Insensitive: If the location and the time of when the end-effector achieves a given velocity does not matter it is considered time and location insensitive. The end-effecter's velocity can be dependent on the location achieved but the location is an independent variable and the velocity is the dependent variable. In this case the manipulator has control over the object until the release stage. Examples of this would be pitching a baseball, bowling, throwing a grenade or horseshoes etc. Throwing is an example of when the end-effector's velocity holds a higher priority over the position.

Mechanisms with only a single degree of freedom are re-

stricted to throwing in a plane. 2-DOF mechanisms are able to throw in R^3 space with the correct kinematic structure. Such a mechanism can choose its release point or its end-effector velocity but not both. Mechanisms containing 3 or more DOF with the correct kinematic structure are able to throw in R^3 and choose both the release point and the end-effector velocity simultaneously.

In recent work Mori et al. [68] has show his ability to control the translational velocity, angular velocity and direction in a 2-dimension plane independently with a single DOF mechanism. The only input is torque to the manipulator. The concept consists is to map the input torque that will change only one of the kinimatic variables and not the other two. This map is done over a given space and thus you can independently chose your translational and angular velocity as well as direction as long as it is in the valid search space.

Senoo et al.[69] used a torque controlled 3-DOF arm to create a high speed throwing trajectory. This arm falls into the *time and location insensitive* category of throwing. Senoo used a kinematic chain approach based on how humans throw. Doing this Senoo was able to achieved an end-effector velocity of 6.0 m/s and can throw in R^3 space. This is done via the use of a planted robot arm made by Barret Technology Inc consisting of 3-DOF with a 360^o rotation base yaw actuator.

Low degree of freedom throwing machines/robots are common. Typical throwing robots have between one and three degrees of freedom (DOF) [68, 70–80]. All of these mechanisms are limited to throwing in a plane.

These low degree of freedom throwing robots are either phys-

ically attached/planted to the mechanical ground or have a base that is significantly more massive then the arm.

Haddadin et al.[81] used their 7-DOF arm and a 6-DOF force torque sensor with standard feedback methods to dribble a basket ball. In addition Zhikun et al. [82] used reinforcement learning to teach their 7-DOF planted robot arm to play ping-pong. Likewise Schaal et al. [83] taught their high degree of freedom (30-DOF) humanoid to hit a tennis ball using an on-line special statistical learning methods. Visual feedback was used in the basketball throwing robot by Hu et al. [84] achieving accuracy of 99%. All of the latter robots were fixed to the ground to guarantee stability.

Kim et al. [85, 86] takes the research to the next level with finding optimal overhand and sidearm throwing motions for a high degree of freedom humanoid computer model. The model consists of 55-DOF and is not fixed to mechanical ground or a massive base. Motor torques are then calculated to create both sidearm and overhand throws that continuously satisfies the zero-moment-point stability criteria [87].

The above works require forms of inverse kinematics. Most of the works use manipulators less then 6 or 7 DOF. This is because for those that have less then 6 DOF (7 DOF in some cases) closed form solutions can be solved for[44, 88–91].

For higher DOF IK solving methods such as Constrained Bi-directional Rapidly-Exploring Random Tree (CBiRRT) [92], neural nets[93–98] or learning methods[99] could be used. These methods do not guarantee any convergence and/or stability of the solution.

For high DOF IK to guarantee convergence if there is a so-

lution it is possible to use the inverse Jacobian transpose IK solving method[100–103]. Using this iterative method requires that there is only a small change from the end effector's current position and the goal position. If the latter is the case, the solver will converge on a solution if one exists. It is important to note that the initial configuration must be known to obtain a solution.

The end-effector velocity control technique described in Section K.1 uses the principles of the inverse Jacobian transpose IK method along with forward kinematics and Lofaro et. al.[15] Sparse Reachable Map (SRM) to create a high DOF work space end-effector velocity controller.

Appendix K. Throwing

In early February 2012 the director of the Philadelphia Science Festival asked the Drexel Autonomous Systems Lab (DASL)[1] if they could have their full-size humanoid Jaemi Hubo throw the ceremonial first pitch at the second annual *Science Night at the Ballpark*. On April 28th, 2012 Hubo successfully threw the first pitch at the Philadelphia Phillies vs. Chicago Cubs game, see Fig. K.1. According to the USA Today were 45,196 fans at the game and thousands more were watching it on television.

Hubo was the first full-size humanoid to throw the inaugural pitch at a Major League Baseball game. This task poses challenges in the area of fully-body locomotion, coordination and stabilization that must be addressed. This paper describes how the latter was done via the analyses/tests of three different approaches and the resulting final design. Section 2 gives a brief introduction to work already done in the field as well as states the requirements for the pitch. Section K describes the three different methods tested where: Section H discusses the balancing methods and criteria used. Section K.2 describes the human-robot kinematic mapping approach that uses a motion capture system to capture a human's throwing motion then mapping that to a full-size humanoid. Section K.1 describes a fully automated approach that uses the sparse reachable map (SRM) to provide viable full body throwing trajectories with the desired end effector velocity[39]. Section K.3 describes the final method explored which is based on key-frame trajectories.

[1]Drexel Autonomous Systems Lab: http://dasl.mem.drexel.edu

Video: http://danlofaro.com/phd/baseball/

Figure K.1: Hubo successfully throwing the first pitch at the second annual Philadelphia Science Festival event Science Night at the Ball Park on April 28th, 2012. The game was between the Philadelphia Phillies and the Chicago Cubs and played at the Major League Baseball stadium Citizens Bank Park. The Phillies won 5-2.

Section L.1 describes the finial design in detail and the modifications needed to make the robot's pitch reliable. Finally Section L.2 gives final thoughts and possible improvements for future years.

K.1 Throwing Using Sparse Reachable Map

A Sparse Reachable Map (SRM) is used to create a collision free trajectories while having the end-effector reach a desired velocity as described in Lofaro et. al.[39]. The SRM has been shown to be a viable method for trajectory generation for high degree of freedom, high-gain position controlled robots. This remains true when operating without full knowledge of the

reachable area as long as a good collision model of the robot is available. The end-effector velocity (magnitude and direction) is specified as well as a duration of this velocity. The SRM is created by making a sparse map of the reachable end-effector positions in free space and the corresponding poses in joint space by using random sampling in joint space and forward kinematics. The desired trajectory in free space is placed within the sparse map with the first point of the trajectory being a known pose from the original sparse map.

$$L_d(0) \in SRM \tag{K.1}$$

$L_d(0)$ is known both in joint space and in free space. The Jacobian Transpose Controller method of inverse kinematics as described by Wolovich et al.[104] is then used to find the subsequent joint space values for the free space points in the trajectory.

$$q_1 = q_0 + \dot{q}_0 = q_0 + kJ^T e|_{x_0}^{x_1} \tag{K.2}$$

Where q_0 and x_0 is the current pose and corresponding end-effector position respectively. q_1 is the next pose for the next desired end-effector position x_1. Each desired end-effector position x must be within a euclidean distance d (user defined) from any point in the SRM.

$$min\left(|x - SRM|\right) < d \tag{K.3}$$

If one of the points in x fails this criteria a new random point is chosen for $L_d(0)$ and the process is repeated.

Each pose in the trajectory is checked against the collision

model to guarantee no self-collisions. The collision model is based on the OpenRAVE model of the Hubo platform called OpenHUBO, see Fig K.2.

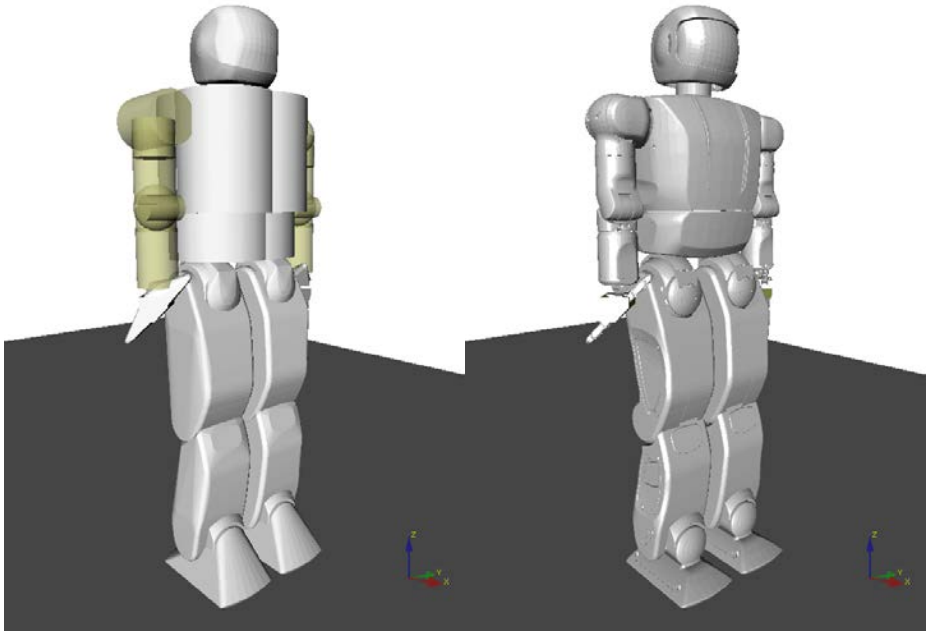


Figure K.2: OpenHUBO - OpenRAVE model of Hubo KHR-4. Left: Collision Geometry. Right: Model with protective shells[39].

The commanded trajectory produces the desired velocity of 4.9 m/s at 60^o. This was then tested on the OpenHUBO and on the Jaemi Hubo platform, Fig K.3 and Fig K.4 respectively.

To ensure balance throughout the motion the balance controller as described in Section H was applied and the static ZMP criteria was checked for the entire trajectory. This method worked as desired. In approximately 10% of the tests one or more joints would over torque and shutdown. This is due to the system not taking the robots power limitations into account.

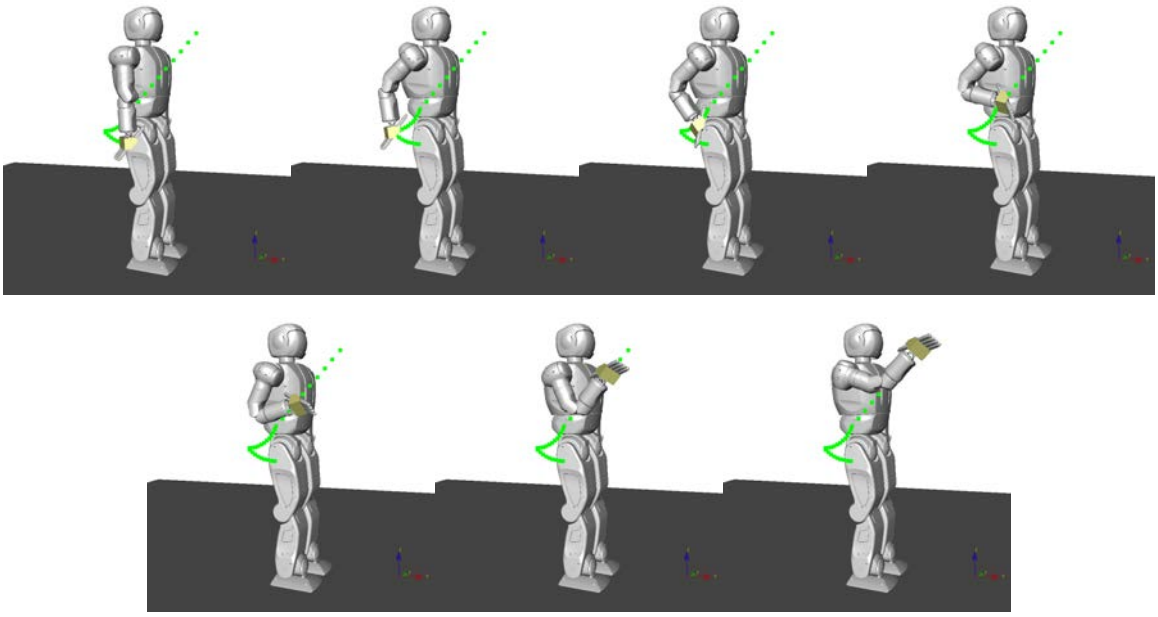


Figure K.3: OpenHUBO running the throwing trajectory immediately after the setup phase is completed. x_0 is top left. Frames are read left to right and have a Δt of 0.15s[39]

K.2 Human to Humanoid Kinematic Mapping

Motion capture (MoCap) systems are commonly used to record high degree of freedom human motion. Athletic trainers in baseball, football and cycling use motion capture to analyze and improve throwing and lower limb motions[105–108]. MoCap systems are also used to generate human-like motions and map those motion to humanoids[109, 110]. Fig. K.5 shows the Hubo's kinematic structure (left) and the human (MoCap) kinematic structure(left). The human has 3-DOF at each joint while the humanoid has limited DOF at each corresponding joint. Some of the challenges in mapping between the human kinematic structure (from MoCap) to a humanoid's kinematic structure are:

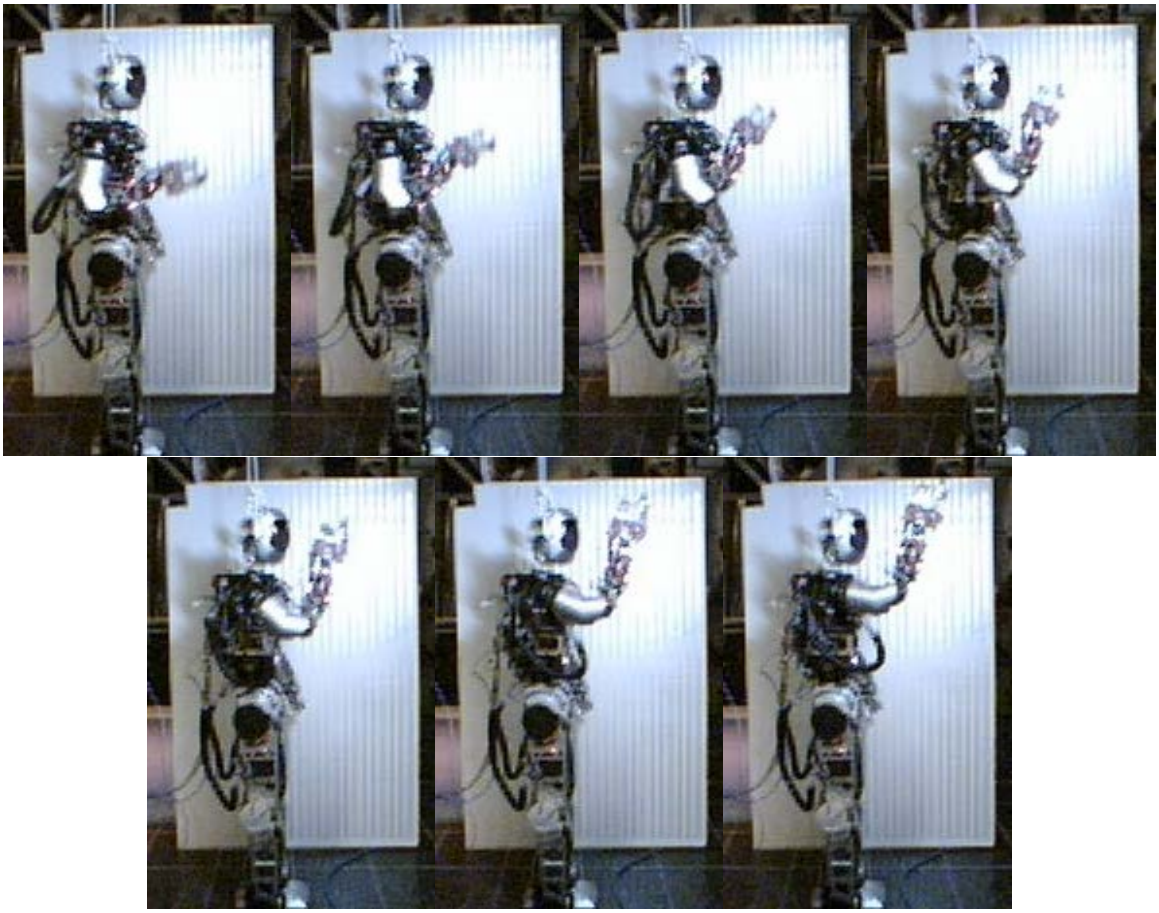


Figure K.4: Jaemi Hubo running the throwing trajectory immediately after the setup phase is completed. x_0 is top left. Frames are read left to right and have a Δt of 0.15 sec[39]

- The difference in the total degree of freedom (DOF).
- The difference in the kinematics descriptions.
- The different Kinematic constraints.

Gaertner et. al.[111] uses an intermediate model (Master Motor Map) to decouple motion capture data for further post-processing tasks. Our approach is to: a) Chose a set MoCap model. b) Preform motions where the pitch motions are decoupled (roll and yaw stays constant), avoids singularities and

robot joint position limitations. c) Combine joint values for near by joints (reduce the model to the same DOF as the robot). d) Some tests require the addition of static offsets to joints to ensure the zero-moment-point (ZMP) criteria is satisfied as stated in Section H

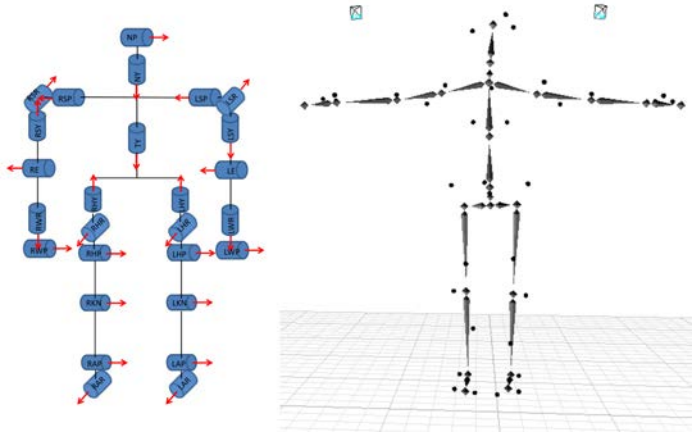
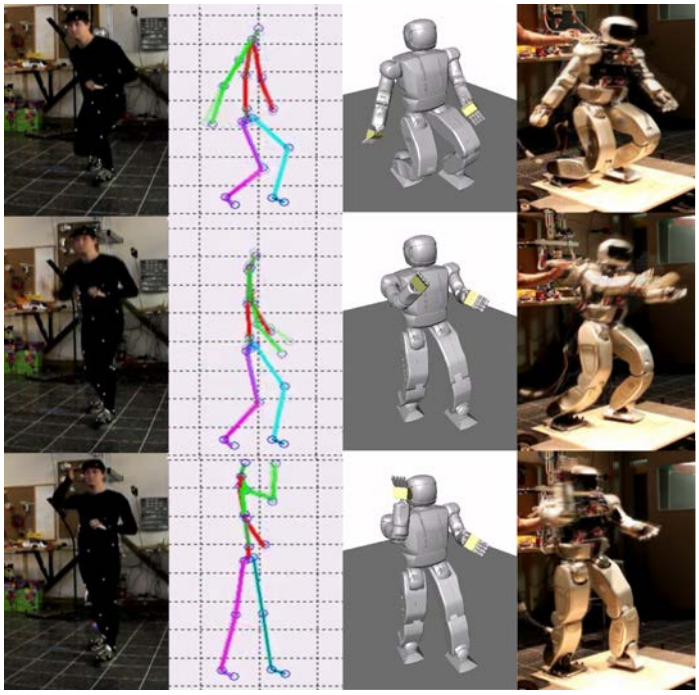


Figure K.5: Left: Jaemi Hubo joint order and orientation using right hand rule. Right: Motion capture model of human figure

To test this method we used a human subject to throw a ball using upper and lower body movements. All motions were in the sagittal plane to keep pitch joints decoupled. To avoid the robot's joint limit of $\pm 180^o$ an underhand throwing motion was used. Fig. K.6 shows the human throwing the ball and the robot throwing the ball to the mapped motion of the human.

To ensure balance throughout the motion the balance controller as described in Section H was applied and the static ZMP criteria was checked for the entire trajectory. The human subject threw the ball approximately eight feet (244 cm). The mapping of the latter motion caused the robot to throw the ball



Video: http://danlofaro.com/phd/underarmthrow/

Figure K.6: (Left to Right): (1) Human throwing underhand in sagittal plane while being recorded via a motion capture system. (2) Recorded trajectory mapped to high degree of freedom model. (3) High degree of freedom model mapped to lower degree of freedom OpenHUBO. (4) Resulting trajectory and balancing algorithm run on Hubo.[112]

approximately five feet (152 cm). The discrepancy comes from the proportional difference in limb length from the human to the robot. A side by side video of the human and the robot throwing the ball is available for viewing on the this papers's homepage[2].

K.3 Key-Frame Motion

Key-frame motion profiles for humanoids borrows from the animation industries' long used techniques. When making an animation the master artist/cartoonist will create the character in the most important (or key) poses. The apprentice will draw all of the frames between the key poses. We borrowed this technique when we: posed the robot in the desired pose, record the values in joint space, and make a smooth motion between poses. In place of the apprentice, forth order interpolation methods were used to make smooth trajectories between poses. Forth order interpolation was used in order to limit the jerk on each of the joints. The resulting trajectory is a smooth well defined motion as seen in Fig. K.7.

To ensure stability throughout the motion the balance controller as described in Section H was applied and the static ZMP criteria was checked for the entire trajectory. The resulting end effector velocity was 4.8 $\frac{m}{s}$ at the release point. Fig. K.8 shows the plot of the magnitude of the end effector's velocity. It should be noted that at the instance of release the velocity vector is at an elevation of 40^o from the ground.

[2]MoCap to Robot (Video): http://danlofaro.com/Humanoids2012/#mocap

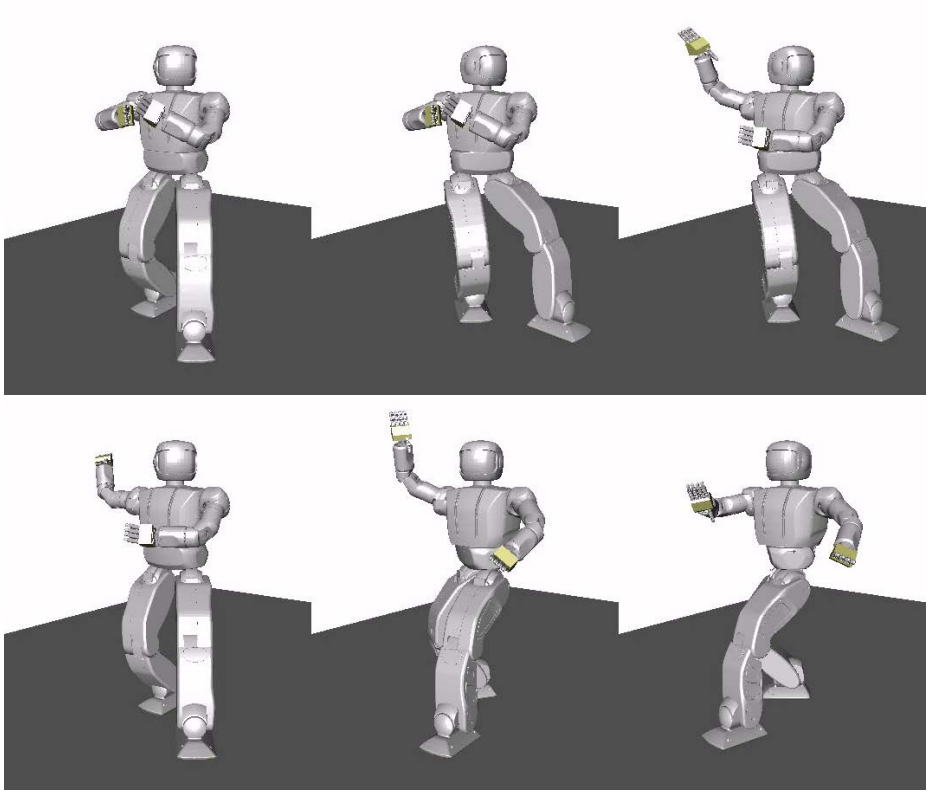


Figure K.7: OpenHUBO using key-frame based method for throwing trajectory creation. Frames are read from top left to bottom right. Video of the above trajectory can be found at http://danlofaro.com/Humanoids2012/#keyframe

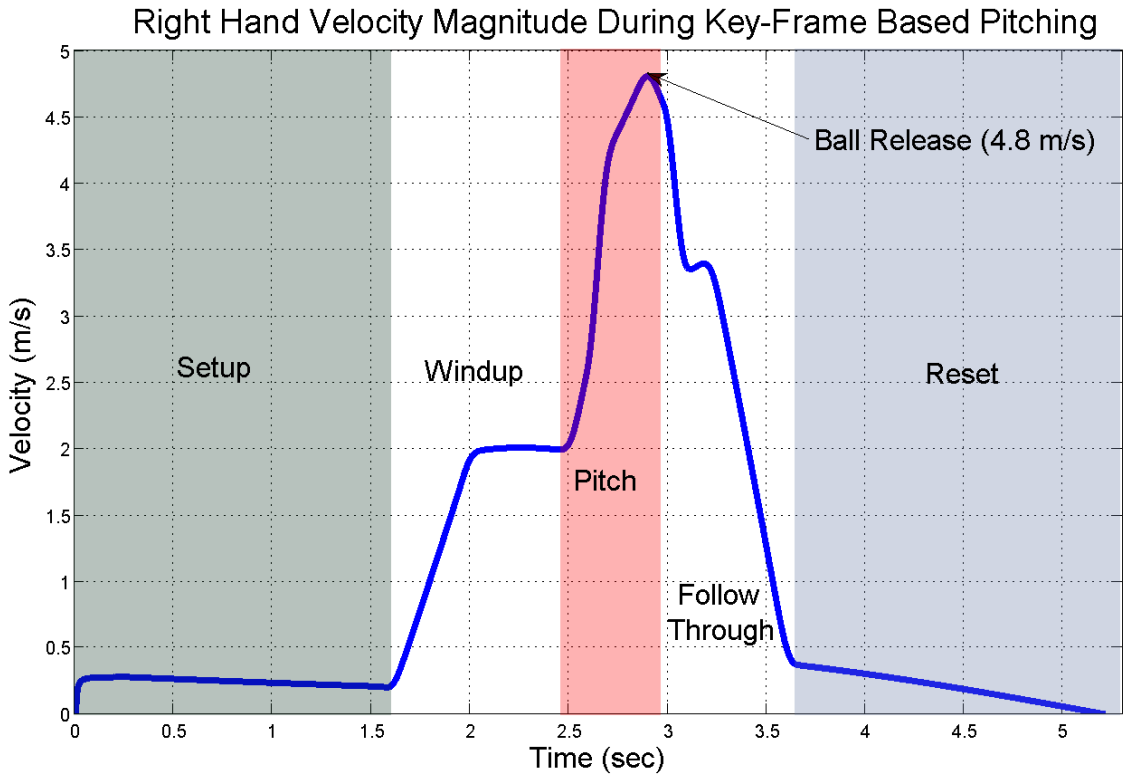


Figure K.8: Velocity vs. Time graph showing the magnitude of the end-effector's velocity for the key-frame based throwing motion. The six different stages of pitching are also shown. Setup: move from the current position to th throw stance. Windup: end effector starts to accelerate from the throw stance and move into position for the start of the pitch state. Pitch: end effector accelerates to release velocity. Ball Release: the ball leaves the hand at maximum velocity (4.8 $\frac{m}{s}$) at an elevation of 40^o from the ground. Follow Through: reducing velocity of end effector and all joints. Reset: moves to a ready state for anther throw if needed.

Appendix L. Sparse Reachable Map Velocity Space Inverse Kinematics

Low degree of freedom throwing machines/robots are common. Typical throwing robots have between one and three degrees of freedom (DOF) [68, 70–73]. All of these mechanisms are limited to throwing in a plane. Sentoo et al.[69] achieved an end-effector velocity of 6.0 m/s and can throw in R^3 space using it's Barret Technology Inc 4-DOF arm with a 360^o rotation base yaw actuator. These low degree of freedom throwing robots are either physically attached/planted to the mechanical ground or have a base that is significantly more massive then the arm.

Kim et al. [86] takes the research to the next level with finding optimal overarm and sidearm throwing motions for a high degree of freedom humanoid computer model. The model consists of 55-DOF and is not fixed to mechanical ground or a massive base. Motor torques are then calculated that both allows for a sidearm or overarm throw and continuously satisfies the zero-moment-point stability criteria[87].

To create a valid throwing trajectory for a high-DOF, high-gain, position controlled robot, a desired line in R^3 in the direction of the desired velocity must be created. Each point in the line is temporally separated by the robot's command period T_r. All points in this line must be reachable. Each point in the line must have poses that do not create a self-collision. A valid throwing trajectory is created when the latter criteria are met.

L.0.1 Self-Collision Detection

Self-collision is an important when dealing with a high DOF robot. Unwanted self-collisions can cause permanent damage to the physical and electrical hardware as well as causing the robot not to complete the given task.

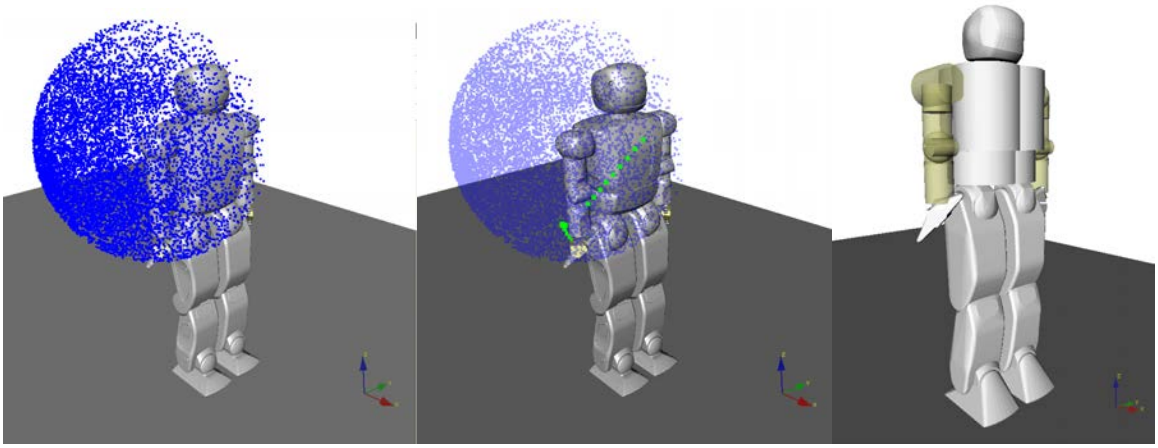


Figure L.1: OpenRAVE model of Hubo KHR-4. Left: Model with SRM of right arm. Center: SRM (blue) with setup and velocity phase trajectories (green) Right: Collision Geometry

To aid in the detection of self-collisions a detailed model of the Hubo KHR-4 was made in the widely used open-source robot simulation environment OpenRAVE[16]. The model was created by exporting the three dimensional schematics that the physical robot was created with, to a format that OpenRAVE can use. This was done in order to ensure an accurate and detailed model. For these experiments we needed the external boundaries only; the internal geometry was replaced with a simplistic representation. The external shell is the only part

now visible, see Fig L.1 (Center). The Proximity Query Package (PQP) was used to detect collisions between any two pieces of the robot's external shell. Due to the high polygon count of the external shell the computation time of detecting a collision was on the magnitude of seconds. It is advantageous to reduce this time if the system is to run live on the robot. Computation time is decreased significantly when boundary/collision geometries are simplified due to the lower polygon count. The collision geometries were further simplified to decrease computation time by making them primitives such as spheres, cylinder and boxes, see Fig L.1 (Right).

Joint limitations are added to the model to mimic the physical robot. The model can be commanded the same configurations as the physical robot. A pose is commanded to the model, PQP searches for any collisions. With the simplified collision geometry self-collisions are detected on the order of milliseconds. If there are no collisions then the pose can be applied to the physical robot. A 5% increase in volume between the simplified collision geometry and the high polygon geometry was added to ensure all of the physical robot's movements will not collide due to minor calibration errors.

L.0.2 Reachable Area

The desired end-effector velocity must be achieved with all joint limits and self-collision constraints satisfied at all times. Typical methods of determining reachability is to move each joint through its full range of motion for each DOF[113, 114]. Due to the high DOF of the Hubo KHR-4 this method is not

desirable. A sampling method described in this work is similar to Geraerts et al.[115]. It was used to accommodate the high DOF system. Both active and static joints must be defined to calculate the reachable area of a manipulator at a discrete time N. The static joints are assumed to hold a fixed position at time step N. Active joints are free to move to any position as long as it satisfies the joint angle limitations and does not create a self-collision. A uniform random number generator is used to assign each active joint with an angle in joint space. Each random angle assigned is within the valid range of motion of the respective joint. The self-collision model described in Section L.0.1 is used to determine the self-collision status with the randomly assigned joint angles. If there is no self-collision the end-effector position and transformation matrix T are calculated using forward kinematics.

$$\chi_{\mathbf{i}} = \begin{bmatrix} R_i & \Gamma_i \\ 0 & 1 \end{bmatrix} \tag{L.1}$$

$$T = [\chi_1 \cdot \chi_2 \cdot ... \cdot \chi_n] \tag{L.2}$$

where χ_i is the transformation between joint $i-1$ and i, R_i is the rotation of joint i with respect to joint $i-1$ and Γ_i is the translation of joint i with respect to joint $i-1$, and n is the number of joints in the kinematic chain.

The end-effector position and the joint angles used are recorded. This process is repeated multiple times to form a sparse representation of reachable end-effector positions in R^3 and the corresponding joint angles in joint space. The resulting representation is called the Sparse Reachable Map (SRM). Fig. L.2

shows a cross section of the SRM about the right shoulder between -0.40 m to 0.40 m on X, -0.40 m to 0.40 m on Z, and -0.21 to -0.22 m on Y. The blue points show valid end-effector locations with known kinematic solution in joint space. Fig. L.1 shows the SRM of the entire right arm. The SRM is used to calculate valid movement trajectories.

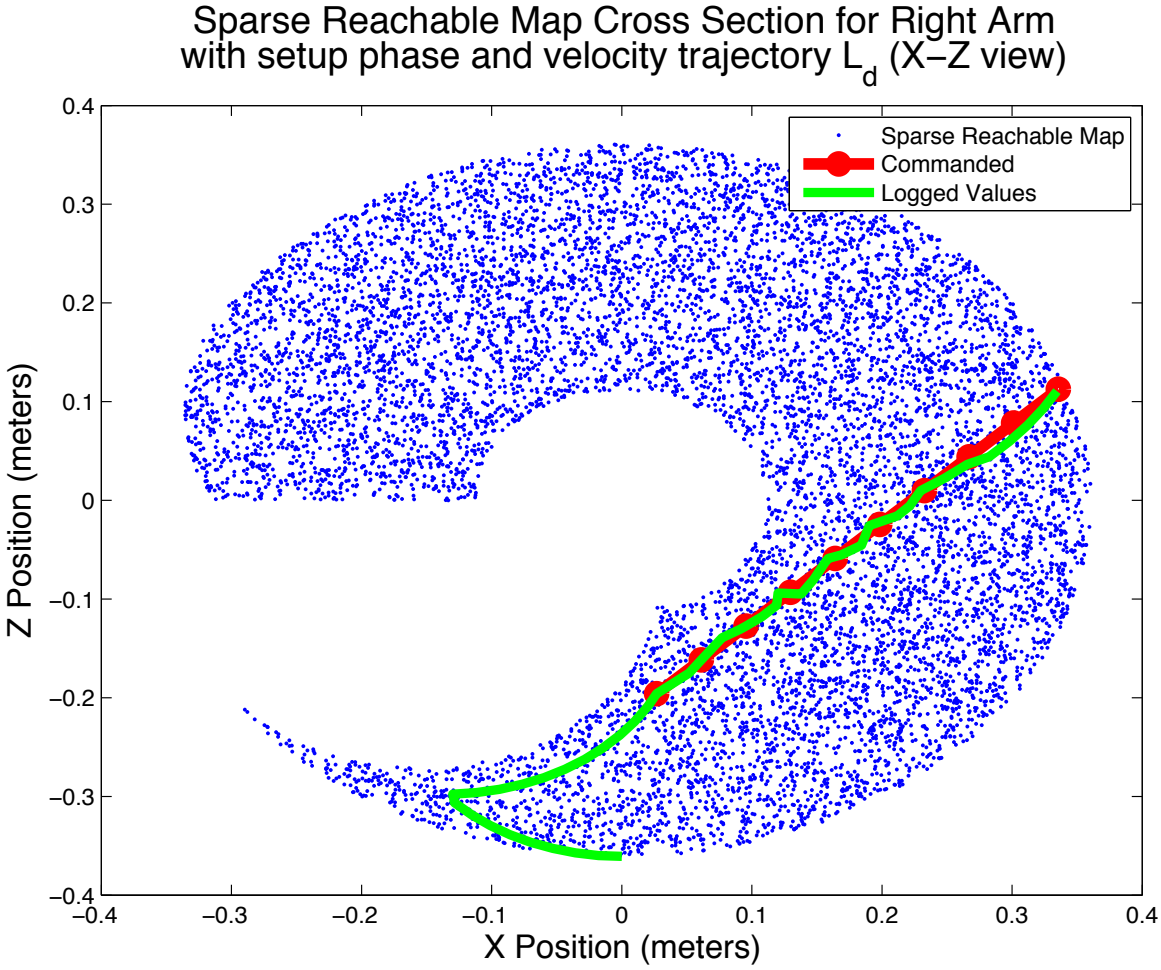


Figure L.2: Cross section of the SRM about the right shoulder between -0.40 m to 0.40 m on X, -0.40 m to 0.40 m on Z, and -0.21 to -0.22 m on Y. (Blue) show valid end-effector locations with known kinematic solution in joint space. (Red) Commanded right arm end-effector position in R^3. (Green) The logged joint space values converted to R^3 using forward kinematics.

L.0.3 Trajectory Generation

An end-effector velocity, $\vec{V_e}$, is chosen based on target location, the well known equations of projectile motion, and the required velocity duration t_e. $\vec{V_e}$ must be held for a time span of t_e. The release point must be within the time span t_e. The magnitude of the velocity in the direction of $\vec{V_e}$ immediately preceding time span t_e must be less than or equal to the magnitude of $\vec{V_e}$ during t_e. t_e must be an integer multiple of the robot's actuator command period T_r.

A line $\vec{l_d}$ in R^3 that passes through (X_0, Y_0, Z_0) in the direction of $\vec{V_e}$ is created. $\vec{L_d}$ is the discrete representation of $\vec{l_d}$. Each point in $\vec{L_d}$, (X_0, Y_0, Z_0), $(X_1, Y_1, Z_1) \cdots (X_n, Y_n, Z_n)$, are separated by a time span T_r.

The desired velocity is defined as

$$\vec{V_d} = [V_x\hat{i}, V_y\hat{j}, V_z\hat{k}] \tag{L.3}$$

The line $\vec{L_d}(n)$ is defined as

$$\vec{L_d}(n) = [X_n\hat{i}, Y_n\hat{j}, Z_n\hat{k}] \tag{L.4}$$

where n is the current zero based time step index value for the time span t_e. The change in $\vec{L_d}$ between time step 0 and n must be equal to our desired velocity $\vec{V_d}$.

$$\frac{\Delta\vec{L_d}|_0^n}{n \cdot T_r} = \vec{V_d} \tag{L.5}$$

thus

$$\vec{V_d} = \frac{\vec{L_d}(n) - \vec{L_d}(0)}{n \cdot T_r} \tag{L.6}$$

The line $\vec{L}_d$ at time step n can now be defined in terms of $\vec{V}_d$, T_r, the origin $\vec{L}_d(0)$, and the current zero based time step index value n.

$$\vec{L}_d(n) = n \cdot T_r \cdot \vec{V}_d + \vec{L}_d(0) \tag{L.7}$$

where

$$\vec{L}_d(0) = [X_0, Y_0, Z_0] \tag{L.8}$$

The line $\vec{L}_d$ is the trajectory the robot's end-effector must follow during the time span t_e. The starting point $\vec{L}_d(0)$ must be found so that $\vec{L}_d$ is within the reachable area. $\vec{L}_d(0)$ is set to a random starting points chosen within the SRM.

$$\vec{L}_d(0) \in SRM \tag{L.9}$$

All subsequent points in $\vec{L}_d$ must fall within some Euclidean distance d from any point in SRM. If one of the points in $\vec{L}_d$ fails this criteria a new random point is chosen for $\vec{L}_d(0)$ and the process is repeated.

Once an $\vec{L}_d$ is found that fits the above criteria the inverse kinematic solution must be found for each point and checked for reachability. Smaller values of d will increase the probability $\vec{L}_d$ is within the reachable area defined in the SRM however more iterations will be required to find a valid $\vec{L}_d$. Larger values of d will decrease the number of iterations needed to find a valid $\vec{L}_d$ however the probability of $\vec{L}_d$ being in the reachable area is decreased. In addition larger values of d decreases the system's ability to properly map near sharp edges in the SRM. Increasing the number of samples in the SRM will allow for larger values

for d.

L.0.4 Inverse Kinematics

The trajectory $\vec{L}_d$ has one point with a known kinematic solution in R^3 and in joint space, $\vec{L}_d(0)$. The joint space kinematic solutions for points $\vec{L}_d(1) \rightarrow \vec{L}_d(n)$ are unknown. Mapping the robot's configuration $\vec{q} \in Q$ to the desired end-effector goal $\vec{x}_g \in X$, where Q is the robot's configuration space and X is in R^3, is done using Jacobian Transpose Controller used by Weghe et al.[116]. Weghe shows the Jacobian as a linear map from the tangent space of Q to X and is expressed as

$$\dot{\vec{x}} = J\dot{\vec{q}} \tag{L.10}$$

The Jacobian Transpose method is used because of the high DOF of the Hubo KHR-4. Under the assumption of an obstacle-free environment the Jacobian Transpose Controller is guaranteed to reach the goal. A proof is shown by Wolovich et al.[104].

To drive the manipulator from its current position $\vec{x}$ to the goal positions $\vec{x}_g$ the error $\vec{e}$ is computed and the control law is formed.

$$\vec{e} = \vec{x}_g - \vec{x} \tag{L.11}$$

$$\dot{\vec{q}} = kJ^T\vec{e} \tag{L.12}$$

where k is a positive gain and self-collisions are ignored. The instantaneous motion of the end-effector is given by

$$\dot{\vec{x}} = J\dot{\vec{q}} = J(kJ^T\vec{e}) \tag{L.13}$$

The final pose $\vec{q}$ for our goal position $\vec{x}_g$ can now be found.

The Jacobian Transpose method works best when there is a small difference between the current position $\vec{x}$ and the goal position $\vec{x_g}$. $\vec{L}_d(0)$ is known both in X and in Q and is the starting point.

$$\vec{x} = \vec{L}_d(0) \tag{L.14}$$

$$\vec{q_0} = SRM\left(\vec{L}_d(0)\right) \tag{L.15}$$

The goal position $\vec{x}_g$ is set to the next point in $\vec{L}_d$

$$\vec{x}_g = \vec{L}_d(1) \tag{L.16}$$

The pose $\vec{q_1}$ can now be calculated

$$\vec{q_1} = \vec{q_0} + \dot{\vec{q}}_0 = \vec{q_0} + kJ^T\vec{e}|_{\vec{x}}^{\vec{x}_g} \tag{L.17}$$

where $\vec{x}_g = \vec{L}_d(0)$ and $\vec{x} = \vec{L}_d(1)$. $\vec{L}_d(1)$ is now known both in X and in Q. Now $\vec{x} = \vec{L}_d(1)$ and the process is repeated until all points in $\vec{L}_d$ are known both in X and Q.

L.0.5 On-Line Trapezoidal Motion Profile

The robot's starting position $\vec{x}_0$ is not guaranteed to be the same as the first point in the velocity trajectory $\vec{L}_d$. To avoid over large accelerations when giving this step input from $\vec{x}_0$ to $\vec{L}_d(0)$ an on-line trapezoidal motion profile (TMP) was used to

generate joint space commands with the desired limited angular acceleration and velocity. The TMP was only active during the setup phase where the robot's end-effector moves from $\vec{x}_0$ to $L_d(0)$. This is because the TMP's inherent nature has the potential to adversely effect the desired velocity in R^3 under high angular velocity and acceleration conditions in joint space.

The TMP was designed to limit the applied angular velocity and acceleration in joint space and to prevent over-current/torque. An important advantage over simply limiting output velocity and acceleration is that the TMP has little to no overshoot. When a clipped and rate-limited velocity profile is integrated, the resulting position trajectory may over or undershoot due to this non-linear system behavior. The TMP accounts for the imposed limits inherently, and will arrive at a static goal without overshoot. Table L.1 describes the three regions that make up the TMP.

Table L.1: Trapezoidal Motion Profile Regions

Region 1	Accelerate at maximum acceleration in direction of goal
Region 2	Achieve and hold maximum velocity
Region 3	Decelerate to zero velocity to reach goal

The area under the velocity trapezoid in region 1-3 is the total displacement achieved by the profile. By shaping this profile based on initial and goal conditions, any goal position can be precisely reached, even if velocity clipping occurs. The shape of the profile can be challenging to identify, since it is not always a trapezoid. For large velocity and acceleration limits

and small displacements, the profile will only reach a fraction of maximum velocity, and will be triangular. The varying shape of the profile means that calculating and storing complete motion profiles for each update may be required. This paper's method removes the need for complete profile generation and storage.

Regions one and two of the velocity profile are bounded by the maximum acceleration, a_m, and maximum velocity, v_m, respectively. In these regions the joint moves towards the goal as fast as the limits allow. In region three the joint has reached a deceleration distance d_s from the goal. It now accelerates at $-a_m$. When the velocity reaches zero, the joint has exactly arrived at the goal position. d_d is the integral of the velocity profile in region three, given by (L.18).

As long as the distance to the goal d_g and d_s are equal then the controller needs to decelerate at the maximum rate to come to rest at the goal. Conversely, for the current goal distance, there is a critical velocity v_c such that, if the joint began moving at this velocity in the following time-step τ, it could decelerate at a_m to reach the position goal. The controller minimizes the error between v_c and v_0 at each time-step.

Since the joint is moving with velocity v_0 during a current time-step, some initial distance d_i (L.19) is traveled before the joint can be affected. Defining $\hat{u}$ as the sign of the distance to the goal, v_c is related to d_g and d_i quadratically in (L.21). This equation assumes simple trapezoidal integration. Solving for v_c using the quadratic formula generally produces complex roots due to the possibility of negative v_0 or d_g. In (L.22), $v_0 \cdot \hat{u}$ is the current velocity relative to the goal direction, producing a positive term if the signs of both terms match. This result will

always produce a real value for v_0 and d_g.

$$d_s = \frac{v_0^2 sign(v_0)}{2a_m} \tag{L.18}$$

$$d_i = v_0\tau + \frac{v_c - v_0}{2}\tau \tag{L.19}$$

$$\hat{u} = sign(d_g) \tag{L.20}$$

$$v_c^2 = 2a_m\left(d_g - d_s\right) \tag{L.21}$$

$$v_c = \hat{u}a_m\left(\sqrt{\frac{a_m\tau^2 - 4\hat{u}v_0\tau + 8|d_g|}{4a_m}} - \frac{\tau}{2}\right) \tag{L.22}$$

L.1 Final Design

The final goal is the have an end-effector velocity of 9.47 $\frac{m}{s}$ at 45^o. The key-frame method was tested to throw at 4.8 $\frac{m}{s}$. To increase the end-effector velocity the upper body motion was kept unchanged but the lower body added a stepping motion with its legs. The stepping motion consists of lifting the left foot up, pushing forward with the right and move the left forward 10 cm. Stepping with your non-dominant foot, and pushing with the dominant, when throwing overhand is common practice to increase the distance you can throw a ball. Jaemi Hubo throws with its right hand and steps with its left. This increased the end-effector velocity from 4.8 $\frac{m}{s}$ to 7.1 $\frac{m}{s}$. Fig. L.3 shows the stepping motion of the robot.

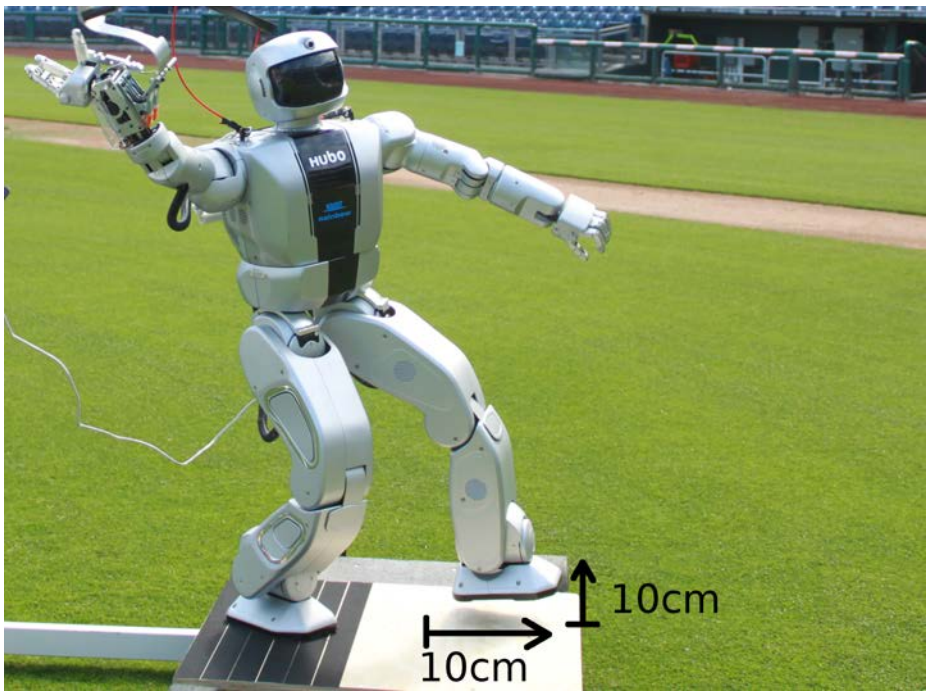


Figure L.3: Hubo stepping 10 cm up and forwards increasing the end effector velocity by 2.3 $\frac{m}{s}$.

The addition of pushing off with the right foot and stepping forward introduced two problems. 1) The ZMP criteria is not satisfied throughout the motion and 2) the right foot would slip when pushing its body forward. To avoid slip *hook and loop* was paced on the bottom of the right foot (non-dominant) and on the throwing platform. This did not permanently attach the robot to the platform but it did allow for more friction between the foot and the ground. This allowed the balancing controller to function adequately for the short step and maintain stability. The platform was added to ensure a more consistent ground for the robot to balance on than the baseball field can inherently provide.

An additional 2.5 $\frac{m}{s}$ was needed to give a proper throw. Bor-

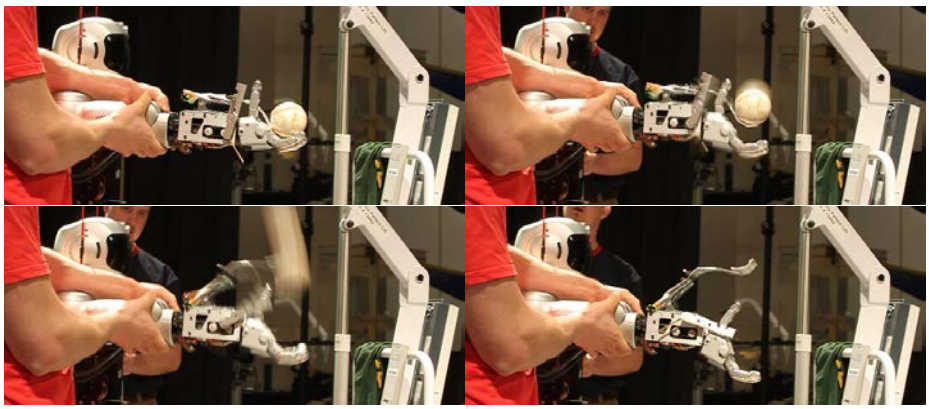


Figure L.4: Spring loaded mechanism test launching the baseball. Top-Left: Pre-launch. Top-Right/Bottom-Left: Launch. Bottom-Right: Pos-launch. The mechanism added 3.0 $\frac{m}{s}$ to the end-effector velocity at its release point.

rowing from the GRASP Lab and their high powered pneumatic wrist on their PhillieBot, a spring loaded mechanism was added to Hubo's wrist, see Fig. L.4. The addition of this mechanism allowed the robot to achieve an end-effector velocity magnitude of 10 $\frac{m}{s}$. Fig. L.5 shows a frame overlay of the the Hubo throwing a regulation baseball 10 m (32.8 feet). Fig. K.1 shows the same throw at Citizens Bank Park on April 28^{th}, 2012.

L.2 Conclusion

Throwing using the key-frame based method was the most reliable and successful. The system was open-loop in respect to the location where it would throw the ball. This is why it over threw the ball during the real pitch, see Fig. L.5. The lessons learned when performing the throwing task is that

1. A unified algorithmic framework is needed for three tier testing

Figure L.5: (TOP) Pitch at Phillies Game. (BOTTOM) Practice pitch at Drexel. Frame overlay of the Hubo throwing overhand a distance of 10 m (32.8 feet) with a release angle of 40^o and a tip speed of 10 $\frac{m}{s}$. Captured at 20 fps with a shutter speed of 1/30 sec. Each of the white dashes of in the image is the actual baseball as picked up by the video camera.

2. Using such a framework the loop has to be closed on the ball's final location

3. System for having a stable landing when taking a step

Section 3 describes the unified framework that answer #1 above. As stated before throwing is a full body locomotive task. Section 5.2 closes the loop using visual methods. This shows how the unified algorithmic architecture can be used for

visual servoing a full body locomotive tasks. This visual servoing example answer #2 above. The use of active damping as seen in Section 5.3 allows the robot to land on the ground. This answers #3 above.

Appendix M. Validation: Peer Survey on Hubo-Ach

This section shows the peer survey taken by users of Hubo-Ach. Thirteen independent users were surveyed. The overwhelming conclusion was that the system is useful, was the unifying algorithmic framework as advertised and helped with development. Out of a score from 0-10 on the question *"Would you use Hubo-Ach again the the future when programming Hubo"* received an average of 9.23 (see Table M.11. Table M.1, M.3, M.5, M.7, M.9, M.11).

Table M.1: Q1: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

Opinions about Hubo-Ach:
10 = Agree, 0=Disagree
Sample Size = 13

Question	Average Rating (0-10)
It is easy to use Hubo-Ach	8.77
It is easy to integrate Hubo-Ach into your existing controllers/systems	7.69
Hubo-Ach makes it conducive for you to use pre-existing tools (such as ROS, OpenRAVE, DART, Custom Software, etc.)	8.31
The multi-process methodology of Hubo-Ach makes it easy for you to implement your controllers in any language you desire.	8.85
Hubo-Ach is easier to use then other high DOF real-time robot software you have used in the past	8.62

Table M.3: Q2: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

Hubo-Ach and your controller implementations:
10 = Agree, 0=Disagree
Sample Size = 13

Question	Average Rating (0-10)
You successfully integrated Hubo-Ach into your existing controllers/systems	9.15
The latency in Hubo-Ach does not have a noticeable effect on your controllers	9.38
The sampling frequency does not have a noticeable effect of your controllers	9.15

Table M.5: Q3: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

What programming languages do you interface with Hubo-Ach:
10 = Often, 0=Never
Sample Size = 13

Question	Average Rating (0-10)
C/C++	9.3
Python	7.69
MATLAB	3.15
Other	1.92

Table M.7: Q4: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

What simulators do you use in conjunction with Hubo-Ach:
10 = Often, 0=Never
Sample Size = 13

Question	Average Rating (0-10)
DART	3.23
OpenHubo	7.54
RobotSim	2.54
Other	3.92

Table M.9: Q5: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

Choice of Hubo Software: Given the choice, how likely is it that you would use the following software platforms to implemented your controllers on Hubo.:
10 = Very Likely, 0=Unlikely
Sample Size = 13

Question	Average Rating (0-10)
ACES/Conductor	2.69
Hubo-Ach	9.51
Maestro	5.42
RAINBOW (Windows)	3.46
RAINBOW (Xenomai)	4.00

Table M.11: Q6: Survey on the Unified Algorithmic Framework for Complex System and Humanoids, Hubo-Ach:

Effects of Hubo-Ach:
10 = Agree, 0=Disagree
Sample Size = 13

Question	Average Rating (0-10)
Without Hubo-Ach physically implementing your controllers on Hubo would have been much more difficult	9.00
Hubo-Ach was a key component is quickly implementing your controllers on the physical Hubo	9.15
You would use Hubo-Ach again in the future when programming Hubo	9.23