KINEMATIC CONTROL OF A BIOMIMETIC ROBOT ARM THROUGH MOTION CAPTURE AND INVERSE KINEMATICS FOR TELEROBOTIC APPLICATIONS

Vishal Gattani

Master of Technology Thesis
September 2020

ज्ञानमुत्तमम्

International Institute of Information Technology, Bangalore

KINEMATIC CONTROL OF A BIOMIMETIC ROBOT ARM THROUGH MOTION CAPTURE AND INVERSE KINEMATICS FOR TELEROBOTIC APPLICATIONS

Submitted to International Institute of Information Technology,
Bangalore
in Partial Fulfillment of
the Requirements for the Award of
Master of Technology

by

Vishal Gattani
IMT2015508

International Institute of Information Technology, Bangalore
September 2020

Dedicated to my mother and to the memory of my father

Thesis Certificate

This is to certify that the thesis titled **Kinematic Control of a Biomimetic robot arm through Motion Capture and Inverse Kinematics for telerobotic applications** submitted to the International Institute of Information Technology, Bangalore, for the award of the degree of **Master of Technology** is a bonafide record of the research work done by **Vishal Gattani**, **IMT2015508**, under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma. The thesis conforms to plagiarism guidelines and compliance as per UGC recommendations.

[illegible]

Dr. Madhav Rao

Bengaluru,

The 29th of September, 2020.

KINEMATIC CONTROL OF A BIOMIMETIC ROBOT ARM THROUGH MOTION CAPTURE AND INVERSE KINEMATICS FOR TELEROBOTIC APPLICATIONS

Abstract

In this study, two control strategies to drive an 8-DoF biomimetic robot arm are implemented using motion capture technology and inverse kinematics to help reproduce the desired actions given by the human operator or to generate new human-like motion to interact physically with objects in the environment placed at a certain distance. First, the kinematics of the human upper limb while performing random arm motion are investigated and modeled within a game engine. Then, using this information, the solution for the inverse kinematics problem for the robot arm is implemented within the game engine to position the end-effector in the three-dimensional space using human-like joint configurations. Second, the human motion is analyzed and recorded using a motion capture system and later, applied onto the humanoid robot arm. By performing real experiments using this arm as a platform, it was proved that the above-mentioned control strategies result in human-like upper limb motion. The proposed telerobotic system is integrated with a user-friendly interface using Blender Game Engine for human–machine interaction purposes. Finally, a series of experiments with different control strategies are conducted on the robotic system successfully and the experimental results are presented and discussed. The biomimetic robot design and control strategies makes it highly suitable for telerobotic medical and surgical applications in the future.

Acknowledgements

Throughout the writing of this dissertation I have received a great deal of support and assistance.

This work is supported by the E-Health Research Center (EHRC), IIITB.

I would first like to thank my supervisor, Dr. Madhav Rao, whose expertise was invaluable in formulating the research questions and methodology. You gave me an opportunity to pursue my interests and your insightful feedback pushed me to sharpen my thinking, bringing my work to a higher level.

In addition, I would like to thank my parents for their continuous support. Finally, I could not have completed this dissertation without the support of my brother, Vineet Gattani, and my friends, Nihal Kudligi, Akshay Pant, and Pranav Kedia, who provided stimulating discussions as well as happy distractions to rest my mind outside of my research.

Contents

List of Figures

List of Tables

List of Abbreviations

BGE	Blender Game Engine
DoF	Degrees of freedom
DIP	Distal interphalangeal joint
FK	Forward Kinematics
FoV	Field of View
IIITB	International Institute of Information Technology Bangalore
IK	Inverse Kinematics
MCP	Metacarpophalangeal joint
MoCap	Motion Capture
NIR	Near infra-red
OSC	Open Sound Control
PID	Proportional-Integral-Derivative
PIP	Proximal interphalangeal joint
SDK	Sofware Development Kit
SL	Structured light
SVM	Support Vector Machine
ToF	Time of Flight
TTL	Transistor-Transistor Logic

CHAPTER 1

INTRODUCTION

Telerobotic devices are developed for environments that are dangerous, uncomfortable, limiting, repetitive, or costly for humans to perform [10]. These remote manipulators are nowadays commonly used for inspection, maintenance, search and recovery, waste management and to handle radioactive materials in nuclear power plants. In addition, they have been proposed to support medical staff and enabling doctors to perform surgeries and various medical check-ups in remote locations. However, teleoperation of such devices to perform tasks becomes increasingly complex as, in most of these cases, the robots are remotely controlled. Furthermore, the control interface systems should be as intuitive as possible to the operator. In these remote scenarios, the operator gives the body motion (arm motion) which the robotic arm would replicate to accomplish a certain task. Through addition of visual and tactile feedback, useful information from the surroundings of the telerobotic device can help enhance the operator perception and aid task completion [11]. The goal of telerobotics is to allow a human to control a robot in a situation where it is inconvenient or unsafe for a human and difficult to program a robot to autonomously perform complex operations.

In certain complex industrial tasks, stable, fast and accurate robot positioning is required, while in a number of nonindustrial tasks (e.g. robotic-assisted surgery) dexterity and intelligent positioning is required to avoid obstacles [12], joint limits or singular configurations. For all these reasons, redundant robots have received increased attention during the last decades, along with their associated problem of complex kinematics.

Therefore, dexterous 7-DOF arms are ideally suited for a variety of robotic tasks that include, but are not limited to:

- Inspection, maintenance, and servicing operations
- Assembly and disassembly in space
- Construction of habitats and other structures
- Instrument placement in hard-to-reach locations
- Sampling and sample-transfer operations

A redundant robotic arm consists of more degrees of freedom (DoF) than required to achieve a particular end effector configuration and its kinematic control involves setting of a desired end-effector trajectory in the task space and then computing the respective joint angle trajectories such that the robotic manipulator completes a certain task [13]. Moreover, it is important to understand that the redundancy factor of the robotic arm depends on the particular type of task to be executed where the number of variables which identify the task are lesser than the number of active joints. For instance, a 3-DoF planar manipulator becomes redundant if the tip orientation angle is of no concern for a two-dimensional motion task. In a non redundant robotic manipulator, a given position and orientation of the end-effector corresponds to a finite set of joint angles and associated robot configurations with distinct poses. Therefore, given a trajectory and an end-effector, the motion of the robot is uniquely determined. However, when this motion is hindered due to the presence of obstacles, or reaching the joint limits, there are no available degrees of freedom to reconfigure the robot to reach the desired pose around the obstacles [14].

So, the extra degrees of freedom yields increased dexterity and versatility for performing a task due to the infinite number of joint motions which result in the same end-effector trajectory. However, the increased capability and dexterity of these arms also makes them harder to control. In order to exploit the capabilities of redundant manipulators, effective control schemes are needed to be developed to utilize the redundancy in some useful manner. During recent years, several methods have been suggested to resolve the redundancy problem of manipulators. Thus, the arm can be reconfigured to find better postures for an assigned set of task requirements.

In recent years, many researchers have devoted their attention to creating a teleoperation system with high dexterity that gives the overall system operator-centered control and intuitive feedback. The robotic arm, one of the robotic parts which consists of a set of rigid bodies or links connected by means of revolute or prismatic joints integrating a kinematic chain, is widely used for several fields including telerobotics and in medical rehabilitation to assist a disabled people.

However, during the design of an anthropomorphic human arm most of the salient features of the human arm are lost in the mechanizing process leading to discrepancies between the human and the robotic arm [15]. Even so, the introduction of hinges and linkages to simplify the human arm structure helps in understanding and approximating the kinematics of the human arm in general.

1.1 Thesis Contributions

The motivation of this thesis is to implement control strategies on a Hybrid Prosthetic arm that combines the motion capture technology derived from the Kinect through implementing a marker-less motion capture system, which will complement the equipment from Trossen Robotics and the prosthetic design from [1]. In doing so, two novel control strategies are compared and performed - Motion Capture (MoCap) based control and Endpoint control through Inverse Kinematics using Blender Game Engine.

The contributions of this thesis are as follows:

1. A graphical representation of the left human arm, as far as the degrees of freedom are concerned, is presented for visualizing and simulation purposes.
2. A 3D printed 8-DoF biomimetic robot arm is used as the platform for conducting simulations, experiments and deriving observations.
3. A mechanism to apply human motion to control the kinematics of a redundant robot arm through Motion Capture using Kinect V2 and Blender Game Engine features is presented and discussed.
4. Computer simulations using Blender Game Engine features with the redundant robot arm is implemented making the system capable of producing various arm movements for finer control and precision and used to validate the proposed kinematic-control method.

This study presents several proofs of concept where a telerobotic system with high robot dexterity is operated with various control strategies for human-driven robotics. The proposed methods enables a user to teleoperate the robotic arm to track a specified end-effector trajectory, apply a recorded human motion onto the robotic arm and perform simple tasks by interacting with the objects in the environment.

The following sections introduce the related literature, system architecture, and then describe the robotic system and their control methods, the graphic interface used for visualization and simulation purposes, and the experiments conducted to quantify the ability of the system respectively. Also, the ability to control different bones shows the promise of the Blender Game Engine as a graphical interface to program and model different human motions. Finally, this study presents the results of the teleoperation experiments. This robotic platform is primarily intended for, but not limited to, applications in telerobotics and rehabilitation engineering.

CHAPTER 2

LITERATURE SURVEY AND PREVIOUS WORK

The development of mechatronics has promoted the rapid improvement of human-computer interaction (HCI) leading many researchers and robot engineers to implement upper limb humanoid systems with prosthetic hands. While the human hand is highly dexterous and a complex musculoskeletal structure capable of performing movements including reaching, grasping and manipulation, implementing such an arm as assistive device depend on complex parameters [16] like the number of degrees of freedom (DOF), working envelope and working space that the arm covers, kinematics, payload, speed and acceleration, accuracy, repeatability, and motion control of the arm. For a human operator to maximize the potential of an advanced robotic arm, the implementation of various control strategies remains a key challenge in the field of humanoid robotics.

As modelling and simulation provide a test bed for experimenting with and learning concepts related to a robotic manipulator, a virtual representation of the manipulator provides a cost-effective and flexible solution compared to its mechanical counterpart. The availability of several game engines such as Unity and Blender Game Engine enable virtual simulations to estimate the working of the manipulator in a virtual scenario and provide tools to generate biomimetic (human-like) behaviour for robotic arms. Therefore, virtual environments play an important role in telerobotic supervisory control. According to [17], while taking in to consideration HCI, following features of a game engine are considered important:

1. An easy-to-use graphical user interface for animating objects and controlling interaction.
2. The ability to communicate with external hardware and process multimedia sensory data.

First, this chapter reviews the field of telerobotics, its development and requirements, and provides a few examples of telerobotic devices. Second, this chapter reviews Motion Capture, improvements in tracking human skeleton, and different modes of motion capture; marker based and marker less motion capture for specific applications. Next, we compare the two motion capture technologies; Kinect V1 and Kinect V2, relevant to human motion and skeletal tracking. Next, we will review challenges in modeling, robot kinematics and motion, and bio-mechanical simulations that incorporate Inverse Kinematics relevant to skeletal designs using Blender Game Engine. Finally, we will look at relevant work on design and development of various prosthetic arms and the control strategies implemented for driving them.

In this study, we use the terms Human-Computer Interaction (HCI) and Human-Robot Interaction (HRI) interchangeably to refer to the interaction with the prosthetic arm.

2.1 Telerobotics

It is important to understand the distinctions between a teleoperational system and a telerobotic system. Robotic teleoperation enables an operator move about, sense and mechanically manipulate objects at a distance by utilizing human intelligence. A telerobot is defined for our purposes as a robot controlled at a distance by a human operator, regardless of the degree of robot autonomy [10]. A telerobotic system acts as a robot for short periods, but is monitored by a human supervisor and reprogrammed from time to time. [18] states the distinction between telerobots depends on whether all robot movements are continuously controlled by the operator (manually controlled teleoperator), or whether the robot has partial autonomy (telerobot and supervisory control).

In telerobotics, the human cognitive capabilities for planning and human sensory motor capabilities for control are imposed on the robot device and it responds to functional commands. A robot can learn sequences of operations, then repeat these operations as instructed by the human operator. A typical "teach and repeat" operation include the following tasks:

- defining points in the work space
- defining paths between points
- opening or closing the end-effector (a gripper)
- completing higher level tasks

Using sensors and proper control programs, the robot can also react to and interact with its environment. Essentially, the operator is no longer required to input every move, instead, he or she supervises the operations at some level of control. Virtual environments play an important role in telerobotic supervisory control. A large part of the operator's task is planning, and the use of computer-based models plays a crucial role in the development of a teleoperation system. The virtual environment is an effective way to simulate and render hypothetical environments to impose extreme scenarios in order to determine the abilities of the system, run the experiment, and observe the consequences.

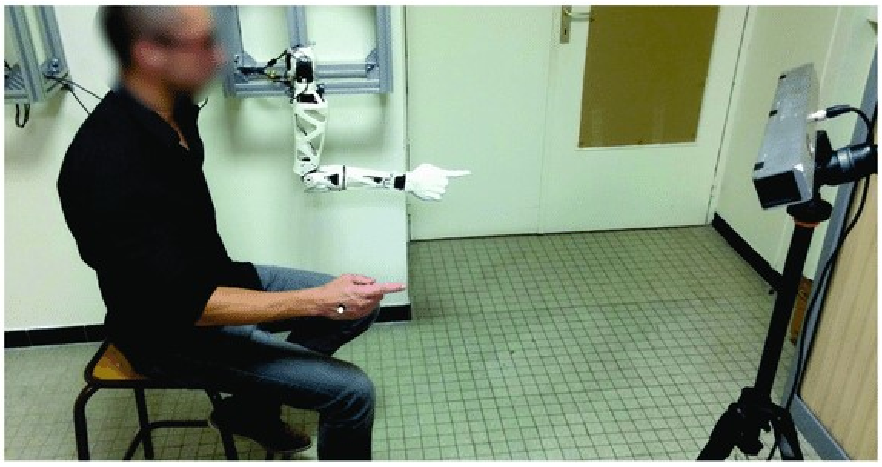
More complex robotic systems have only recently seen use in teleoperation settings such as NASA's Robonaut, the first humanoid robot in space. It is currently on-board the

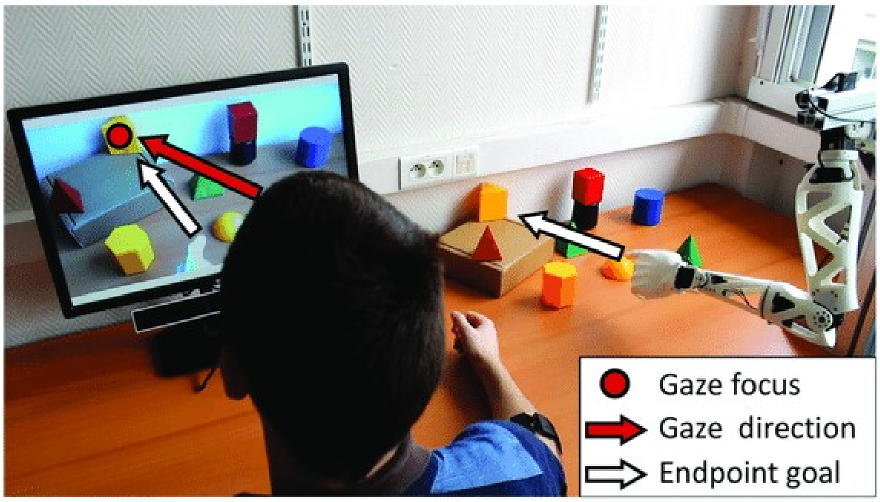
International Space Station and used for tests on how teleoperation, as well as, semiautonomous operation, can assist the astronauts with certain tasks within the station [19]. In [20], telerobotic operations were accomplished by detecting and avoiding obstacles as the operator controls the end-effector using an accelerometer attached to their hand and performing grasping actions by measuring EMG signals on the operator.

In [1], the teleoperation was achieved by continuously tracking a subject's hand trajectory and simultaneously mapping it on the robot's hand (see Figure FC2.1). By computing the instantaneous displacement vector of the subject's hand and mapping this vector in the robot's frame, the robot's hand target position is computed and the arm is put into motion by computing the inverse kinematics solution. In addition to this, the robot was interfaced with gaze-tracking and image processing tools, to allow the user to drive robot arm by the movement of their eyes instead of limbs (see Figure FC2.1b). By locating the focus of the subject's gaze on the plane of the screen, and identify the corresponding object in the robot's reaching space, the robot is put in motion toward the desired object's position by detecting a specific muscle activation pattern using an electromyography device to trigger a movement of the robot.

The GummiArm, a bio-inspired robot arm, comprises of 10 tendon-driven joints, actuated by 19 Dynamixel motors. With 8 of its joints having variable stiffness, this arm can perform movements similar to that of a human arm. Figure FC2.2 shows its ability to absorb impacts, to be teleoperated accurately with high stiffness, to write on a keyboard, and to open a drawer while in motion.

Moreover, the pieces of software operating these devices are should be open and make it feasible for a user to replicate or customize, modify or adapt to a given use case. Therefore, the design from [1] benefits from its open-source software architecture providing a plethora of interfacing options and external softwares can be integrated into the system.


(a) Marker-Based Teleoperation.



(b) Gaze Driven Control.

Figure FC2.1: Reachy Teleoperation and Gaze Driven control of the arm [1].

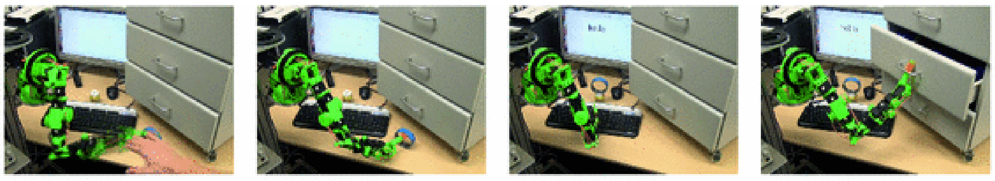
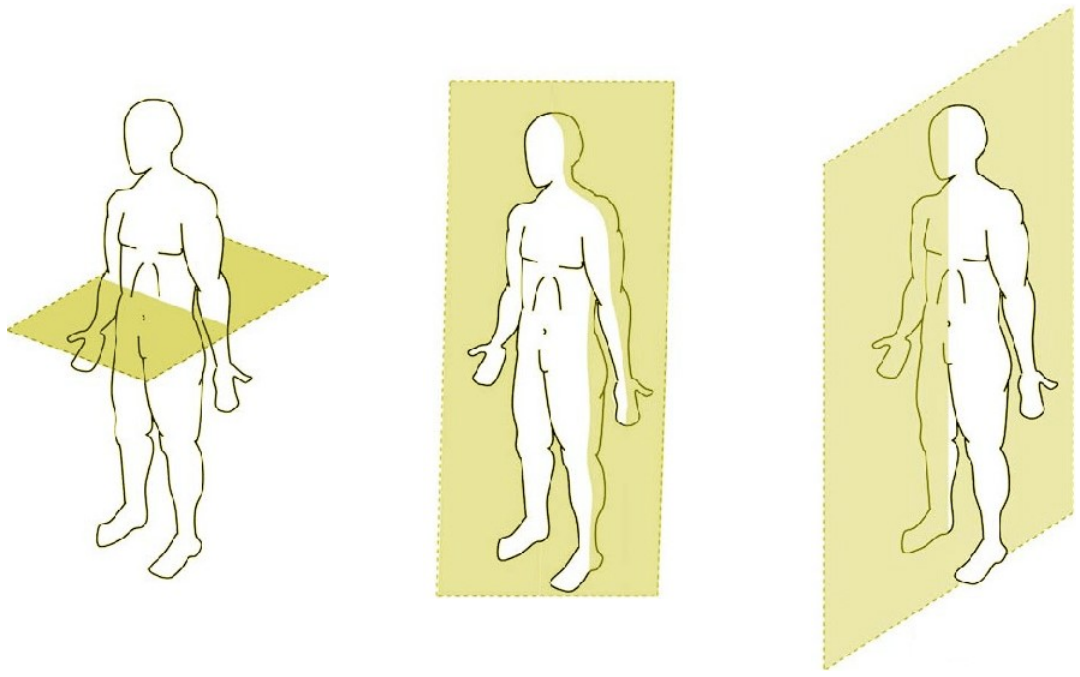

(a) Absorbing physical impacts. (b) Performing a precision move. (c) Typing on a computer keyboard. (d) Opening an office drawer.

Figure FC2.2: GummiArm [2].

2.2 Motion Capture

[21] defines “Motion capture” or “MoCap” as ”the process of recording a live motion event and translating it into usable mathematical terms by tracking a number of key points in space over time and combining them to obtain a single three dimensional representation of the human body motion”. During human motion analysis, [22] states that the human skeleton can be represented as a series of linked body segments to create a spatial human model. In [23], the movement of each body segment in the spatial human model can be described in terms of location and orientation in space based on six degrees of freedom which include forward and backward motions in the sagittal plane (see Figure FC2.3c), side to side in the frontal plane (see Figure FC2.3b), or inward or outward in the transverse plane (see Figure FC2.3a).



(a) Traverse Plane (b) Frontal Plane (c) Sagittal Plane

Figure FC2.3: Planes of Motion

Most common methods for accurate capture realistic motions of three-dimensional human movement require a laboratory environment and the attachment of markers or fixtures to the body's segments. However, such laboratory conditions can prove to be cumbersome and may cause unknown experimental artifacts. It has been recognized that skin movement is the most significant source of error in human movement analysis using markers as they not only impede the subject's natural movement but also deviate from their original position when placed on the skin. Moreover, conventional marker-based motion capture systems often require precise, tedious, and time consuming tasks such as placing markers on the subject's body to analyse human motion and calibration of the space where the task will be performed.

The development of a non-invasive and markerless system, originating from the fields of computer vision and machine learning, has greatly expanded the applicability of human motion capture by eliminating the need for markers and thereby, reducing the patient's preparatory time and enabled time-efficient, and potentially meaningful assessments of human movement in research and clinical practice. [24] demonstrated the feasibility of accurate measurement of human motion using markerless motion capture systems on the basis of visual hulls. Additionally, the markerless framework introduced by [24] expands the applications of human movement capture, minimizing patient preparation time, and reducing experimental errors caused by, for instance, inter-observer variability[1].

As human joint measurement and posture recognition is a crucial area in the field of biomechanics and rehabilitation engineering, the need for a low-cost and efficient motion capture system led to the development of sensor technologies such as the Microsoft Kinect. There are plenty of studies which highlight Microsoft Kinect's applications in the field of biomechanics.

First, [25] highlights that the Kinect is an adequate sensor technology to build an inexpensive and comfortable system that classifies the Parkinson's disease into three

[1]Inter-observer variation is the amount of variation between the results obtained by two or more observers examining the same material

different stages related to Freezing of Gait (FoG). The relevant features used in this classification are related to left shin angles, left humerus angles, frontal and lateral bents, left forearm angles and the number of steps during a spin. Second, [26] presents a non-intrusive gait analysis system using Kinect sensor to extract gait information such as the stride information and the measurement of arm kinematics. [27] also presents a human gait monitoring system whose accuracy in calculating the parameters required for human fall detection is comparable to that of Vicon marker-based motion capture system.

Moreover, [28] compares the performance of a low-cost Microsoft Kinect and a high-cost multi-camera lab-based system OptiTrack. The experimental results from this study concluded that the Kinect sensor was able achieve a comparable performance as of the OptiTrack in terms of motion tracking and it could prove to be a promising Virtual Reality (VR) neurological rehabilitation tool for use in the clinic and home environment.

[29] states that the availability of a low-cost marker-less motion capture device such as the Kinect has made recognition and quantification of human movements more accessible. [30] recognised the potential to develop applications suitable for use in healthcare settings to detect problems that patients have in coordination of movements by developing a non-invasive home monitoring and evaluation system for patients with musculo-skeletal disorders using the Kinect. [31] proposed an approach for a real-time visual feedback helping the user to correct their posture in either sports training or medical rehab exercises in a virtual reality environment, thereby removing the need for a real instructor to provide an assessment of the exercise.

[32] presented a framework for efficient Physiological Function Assessment by measuring the degree of joint mobility and investigating the abnormality of actions of upper limbs using Kinect V2. In [33] and [34], the Microsoft Kinect sensor is used to

recognize different body gestures, generate a virtual interface and perform HRI without having the need to solve the inverse kinematics problem in order to make the robot arm follow the posture of human arm through transmission of joint angles.

Additionally, [35] proposes an approach to transfer human arm movements to an upper-body humanoid. By improving on the skeleton data provided by the Kinect, an inverse kinematics approach is implemented assuming that the robot task is specified in joint coordinates. This study highlights the potential of the depth sensor in providing motion data to teach an upper-body humanoid to perform reaching and manipulation tasks. Also, it is important to note the results of [36]'s work that, though the Kinect sensor was accurate in measuring the timing and gross spatial characteristics of clinically relevant movements, it could not achieve that accuracy in classifying minor movements like hand clasping and toe tapping. Therefore, tasks that require manipulation need post-processing computer vision methods for motion data to segment out minor movements of the hand.

Through the tracking and simulation of the movement of the human body by the motion recognition system, high-risk and difficult working environments in the future could be replaced through the transferring of skills from humans to robotic systems.

2.3 Microsoft Kinect

In 2010, Microsoft, in cooperation with PrimeSense released a structured-light (SL) based range sensing camera, the so-called Kinect V1, as an accessory for the Xbox 360 video game platform in 2010. While it was initially developed as a gaming interface, it found its use as a markerless motion capture system. Kinect V1 consists of of two cameras, i.e. a color RGB and a near infra-red (NIR) camera, and an NIR projector and it measures depth of the objects is measured using the structured light (SL) range sensing principle. From [37], a sequence of known infrared pattern is projected from the NIR projector onto an object, which gets deformed by geometric shape of the object. The object is then observed from a camera from a different direction. By analyzing the distortion of the observed pattern, i.e. the disparity from the original projected pattern, depth information can be extracted as seen in Figure FC2.4.

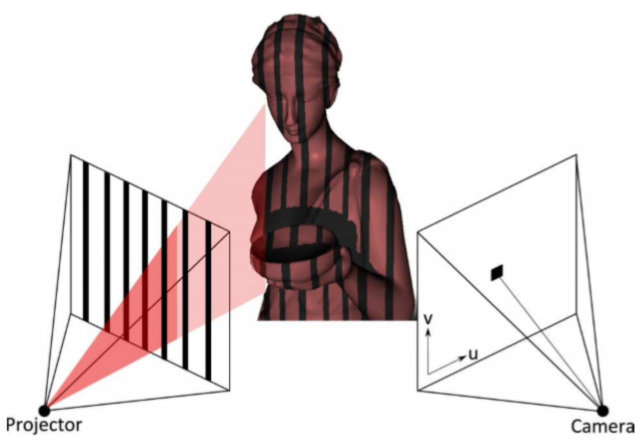


Figure FC2.4: Kinect V1 structured light principle.

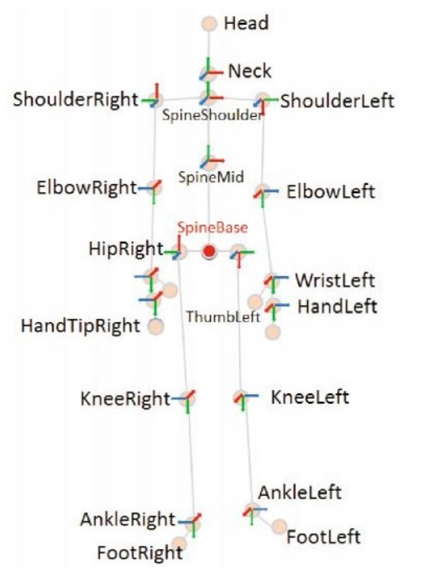
In 2014, Microsoft released the second version of Kinect (Kinect v2) with improved RGB and IR camera resolution having a larger field of view. In Kinect V2, the depth is measured using the time-of-flight (ToF) technology based on measuring the time that light emitted by an illumination unit requires to travel to an object and back to the sensor array. This means that Kinect V2 computes the depth of objects it has in front of it by emitting infrared light rays and computing the time taken by these rays need to reflect from surfaces and find its way back to the sensor. According to [38], this method is more stable, precise and less prone to interference. The depth camera can acquire data in the range from 0.50 to 4.5 m (from [38]. Furthermore, The sensor works properly in an environment with low ambient IR light, making Kinect v2 suitable for outdoor motion capture. The difference between Kinect v1 and Kinect v2 can be seen in Table TC2.1.

[39] states that the Kinect V2 has an integrated Sofware Development Kit (SDK) function for markerless human-motion capture based on [3]'s algorithm on Support

Table TC2.1: Comparison between Kinect v1 and Kinect v2 from [7] and [8].

	Kinect v1	Kinect v2
Sensor	Structured Light	Time of Flight
Range	1.2 3.5m	0.5 4.5m
Joints	20/people	25/people
Body Tracking	2	6
FoV (Color)	62°x 48.6°	84.1°x 53.8°
FoV (Depth)	57.5°x 43.5°	70.6°x 60°

Vector Machines (SVMs) and Randomized Decision Forests which can fully track up to 6 human body simultaneously, defined with 25 joints as shown in Figure FC2.5a, with respect to the reference system defined by the Kinect V2 sensor.



(a) Joints represented by the Kinect

(b) Microsoft Kinect V2

Figure FC2.5: Human body skeleton tracking with Kinect V2 by [3]

In [3], a single input depth image is segmented into a dense probabilistic body part labeling, with the parts defined to be spatially localized near skeletal joints of interest invariant to pose, body shape, clothing, etc. The algorithm generates (possibly several) confidence-weighted proposals for the 3D locations of each skeletal joint by re-projecting the classification result into the world space.

While it may not be anatomically correct, [3] reduces the depth image to a representation of a stick skeleton estimating a total of 25 joints (refer to Figure FC2.5a) and it has shown to have a high correlation with the marker-based motion capture systems according to [36]. [21] states that marker-based motion capture systems offer accurate motion tracking, however, they cannot be widely used due to their high cost and operational complexity.

There are several studies that compare how these two sensor technologies cause the Kinect to possess specific error sources. [37] offers detailed descriptions under which conditions one of these devices is superior to the other. A few of the error sources include Ambient Background Light and Multi-Device Interference are applicable to this study. Though the addition of multiple Kinect devices may lead to better tracking of human motion, Kinect V1 is more prone to interference between multiple data streams. This is due to the fact that the signal shape can be altered in order to prevent multi-device interference in ToF cameras. Also, the depth sensor stream data from Kinect V2 can lead to a problem with USB 3 controller bandwidth forcing a maximum of one Kinect V2 connection per system.

In Kinect V1, the skeleton tracking feature does not track the thumbs of the hand. The Kinect V2 features the tracking of 5 more joints which include the thumb tracking and also allows the skeletal tracking of more people than Kinect V1. Moreover, the skeleton tracking in Kinect V2 appears more natural and precise compared to that of Kinect V1 as seen in Figure FC2.6. Since Kinect V2 offers better precision and tracking features, data acquisition for motion capture is done using a single Kinect V2 for all purposes in this study.

Although unable to support finger tracking, the Kinect sensor can detect an open or closed fist as shown in Figure FC2.7. The red circle stands for a closed fist and the green circle stands for an open fist. Moreover, there has been a lot of work on finger recognition by external observations for extracting 3D poses from an image sequence.

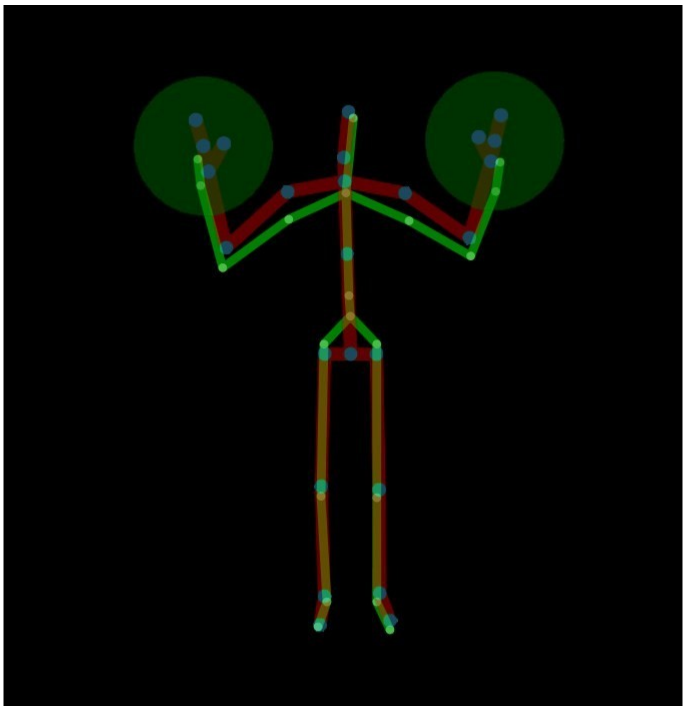


Figure FC2.6: Skeleton detected with Kinect v2 (red and blue) super-imposed to its Kinect v1 version (green).

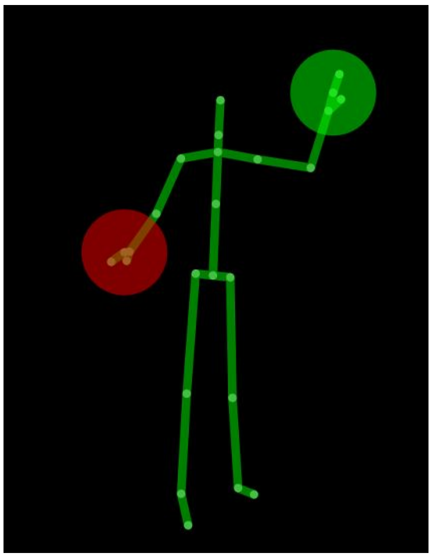


Figure FC2.7: Fist Detection using Kinect. Image taken from Kinect 2 Server - Github.

Typically, from [40], the Kinect sensor is used for motion-sensing, fingers and gestures recognition, such as [41], and the steps used are as follows: (1) depth thresholding; (2) contour extraction; (3) curves detection; (4) fingertips detection; (5) gesture recognition.

In addition to this, [4] searches within the 3D area that is limited between the Hand and the Tip joints (10-15 cm, approximately) and determines the edges of the convex hull to detect the fingers as shown in Figure FC2.8.

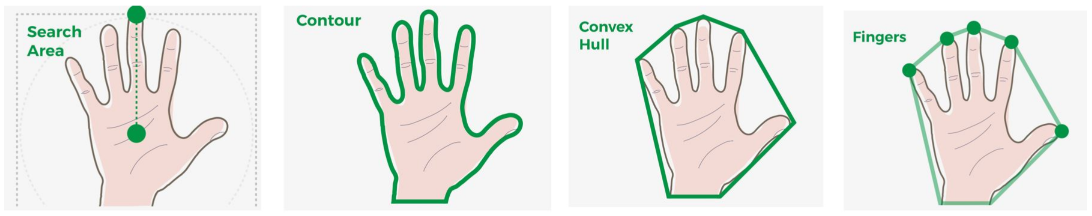


Figure FC2.8: Finger tracking using convex hulls from [4].

With the help of the Kinect's software development kit (SDK), the Kinect V2 can be employed in many cases as an alternative low cost marker-less solution as it provides a model of a 3D skeleton with 25 joints of the human whose full body is placed within the field of view of the Kinect IR camera. According to [42], there exists poor correlation between Kinect V1 skeleton data and commercial motion capture systems which is reported in [43] and [44] during the assessment of lower extremity motions. Due to the low technological specification of Kinect V1, there has been technological improvement in the specifications and accuracy of the Kinect V2 which has led researchers ([45], [46], [47]) to be able to track the sagittal plane's joint angles of the human motion during functional movements and also to evaluate the Kinect's accuracy in capturing joint angles in the anatomical plane beneficial to the field of bio-mechanics.

The field of human motion analysis encompasses the areas of hand movement and sign language recognition, rehabilitation engineering including gait analysis, medical applications and health care applications such as development of assistive robot prostheses, and fall detection. However, this study limits to the use of skeleton tracking data for the analysis and replication of motions in a clinical setting.

In addition to being relatively cheap and easy to setup and operate, [21] mentions that while the Microsoft Kinect sensor can achieve real-time 3D skeleton tracking, this system suffers from the following limitations:

- It is designed to track the front side of the user. The front and back sides of the user cannot be distinguished by the Kinect, i.e., even if the user's back is towards the Kinect, the Kinect is might mistake whether the user's pose is facing towards the camera or not.
- Tracking suffers from occlusions (e.g. self-occlusion by other body parts), and non-distinguishing depths (limbs close to the body) as it is a depth camera.

On the other hand, [48] examined the accuracy of Kinect depth data for static objects, showing that the average error of the depth measurement ranges from a few millimetres up to about 4cm at the maximum range of 5 meters and suggested for mapping applications the data should be acquired within 1–3 meters distance to the sensor as at larger distances, the quality of the data is degraded by the noise and low resolution of the depth measurements. Figure FC2.9 shows the estimation of the depth error of the two versions of the Kinect sensors as a function of the distance between the sensor and the object as determined by [5].

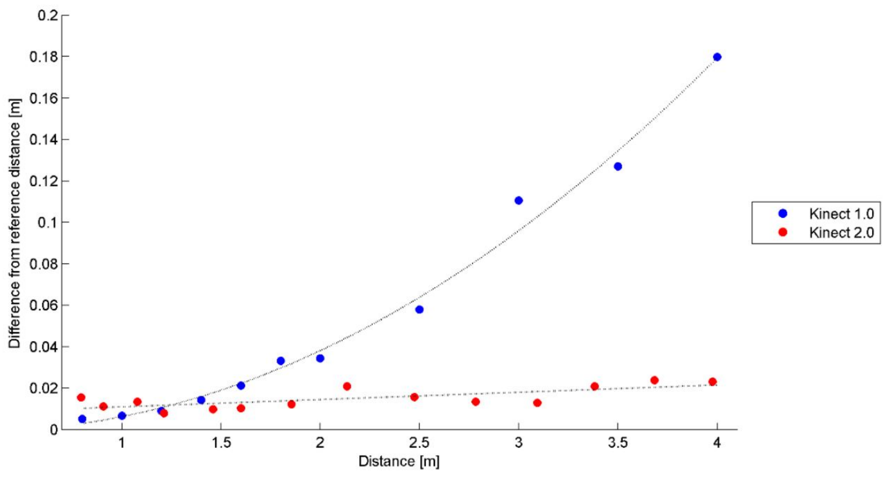


Figure FC2.9: Estimation of the error by the two Kinect sensors as a function of the distance between the device and the object [5].

2.4 Blender Game Engine

Before the motion data from MoCap can be edited by any system, it usually needs to undergo pre-processing to ensure that correct hierarchical connections and bio-mechanical constraints have been taken into account. In bio-mechanical simulations, the primary focus is to implement physiologically accurate representations of skeletal systems. Human body modelling is a problem that arises in ergonomics and in computer graphics applications which involves a complex hierarchical model consisting of many joints, with each joint having different degrees of freedom (DoF) and various possible restrictions, as stated in [49].

A robotic manipulator can be thought of as a chain composed of rigid links connected at their ends by rotating joints. Any transformation applied, either translational or rotational, on the i^{th} joint affects the translation and rotation of any joint placed further along the chain, i.e. $(i+1)^{th}$ in the chain. The design of such a model is an assumption approximating reality and it is required to be analytically and anatomically correct in order to control the available movements of the human body while following various joint constraints. These models are mainly characterised by the number of parameters such as the number degrees of freedom which describe the motion space and are usually constrained by the joint limits and joint structure.

The major difficulty with upper-limb design is the kinematic interpretation of human joints and the development of mechanisms that can mimic human motion. A human arm and hand, without the fingers, has 7 degrees of freedom altogether, six of which are enough for achievement of a desired hand position and orientation. Because of the complex structure of the human arm, most of the proposed joint models are simplified or approximated by more than one joint. Also, the challenge to simulate a movement of a robotic arm, for e.g. to perform a pick and place operation require the information of the total degrees of freedom the robotic arm can have for simulation to work correctly.

While there are many tools that already exist to simulate robots in virtual environments and to visualise sensory data, a robotics simulator based on Blender 3D and Python as scripting language in [50] forms the basis of performing a pick and place operation by a 6-DoF robot in the virtual environment. Visualization, in a graphically appealing manner, is an important prerequisite to understand the internal kinematic changes that occur while keeping the design intact in bio-inspired robotics. Therefore, this study employs the use of Blender Game Engine (BGE) as a graphical interface to perform motion capture and bio-mechanical simulations. From [51], Blender is known for:

1. **Visualisation**: To show a virtual version of the real robot world.
2. **Simulation**: Allows you to run a program in Blender that could also run on a real setup, in order to quantify the robot's behavior and effectiveness.
3. **Emulation**: Allows you to try a control algorithm first on a Blender version instead of on a real robot enabling fine tune controls and checking different behaviors in a variety of situations that may prove to be expensive or dangerous in the real world.

In Blender, a rig is defined as a single chain of bones built to deform a surface mesh alongside a few miscellaneous bones needed for animation control. The process of adding bones to an object or model is called rigging. Once a model of the arm structure is defined, the required motion of the arm structure can be achieved through applying a series of rotational or translational transformations to move end-effector(s) of the chain to obtain a desired position in space. Following the rigging process, there are usually two options to animate a robot arm: either by using Forward kinematics (FK) - moving each bone and joint positions to obtain the desired orientation in space, or by using Inverse kinematics (IK) - moving the end-effector(s) and computing the joint configurations.

Inverse Kinematics (IK) is a method for computing the posture via estimating each individual degree of freedom in order to satisfy a given task; it plays an important role in the computer animation and simulation of articulated figures. Also, it has been used in rehabilitation medicine in order to observe asymmetries or abnormalities alongside Motion Capture technologies. For instance, [52] present an Inverse Kinematics for Upper Limb Compound Movement Estimation in Exoskeleton-Assisted Rehabilitation by approximating the rendering of the actual posture during the elbow-shoulder compound movement. These studies can help in understanding and evaluating the patient limb posture through tests on Exoskeleton-based platforms.

Blender not only provides tremendous support for IK based kinematics and capturing human motion.

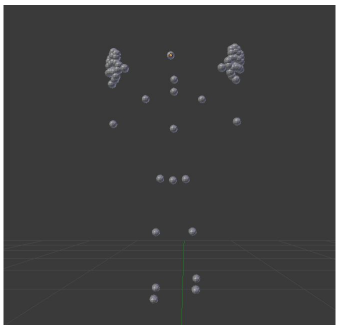
2.4.1 Delicode NI Mate

NI-Mate is a powerful software platform that takes real-time motion capture data from a Kinect and turns it into two industry-standard protocols: Open Sound Control (OSC) and Musical Instrument Digital Interface (MIDI). Amongst these two, NI Mate software makes extensive use of the OSC (Open Sound Control) protocol. With the advent of Kinect's skeletal tracking capabilities, NI Mate can extract skeletal joint tracking data and import it into widely used motion capture software solutions such as Motion Builder, Animata, Blender, Maya, etc. NI Mate enables sampling of multiple sensor data at 30 Hz and transmission of Open Sound Control (OSC) messages through socket programming to a PC which will be used to control and animate 3D models in Blender. This is extremely beneficial in systems that consist of real-time 3D spatial tracking of a physical object.

Since the end goal is to drive a robot arm, NI Mate allows for Skeleton Tracking to track the necessary human body joints. The software allows to choose the point of origin in 3D space when it is tracking joint coordinates. The NI-Mate app captures your movement in the camera and converts it to MoCap data that can be imported into Blender.

Some sensors such as Leap Motion contain hand tracking, NI-Mate is able to output the hand joints in addition to the normal skeleton. However, only the hand tip and the thumb tip joints are available from the Kinect V2. Even so, this option can be enabled to determine whether the fist is open or closed during motion capture.

If the hands option is enabled within NI-Mate, the fingers estimated through the software are added at the end of the arm but are very susceptible to variations due to noise and occlusion. In Figure FC2.10b and Figure FC2.10a, the closing and opening of the fist is detected as tracked through motion-capture. Using these features, the results are as shown in Figure FC2.10.


(a) Open fist captured in Blender using Kinect V2 and NI-Mate add-on.

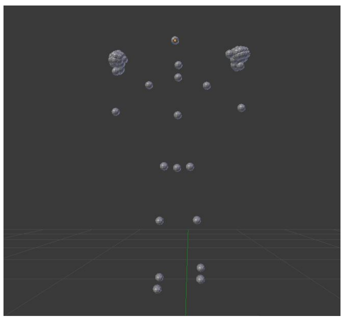

(b) Close fist captured in Blender using Kinect V2 and NI-Mate add-on.

Figure FC2.10: NI-Mate add-on with tracking of fingers enabled.

Therefore, Blender has been chosen as the graphical interface as it supports the entirety of the 3D pipeline—modeling, rigging, animation, simulation, rendering, and motion tracking. In this study, it provides the necessary support for real-time motion capture, and controlling the arm using different modes of kinematic control.

CHAPTER 3

DESIGN AND METHODOLOGY

A visual representation of the upper arm is graphically modelled and represented in Blender which will form the basis for simulating and analysing the shoulder-elbow-wrist mechanism. These representations in the virtual environment helps in studying the different trajectories the arm's joints traverse during any motion. The model representation in Blender is observed to produce smooth, continuous motions that mimic a human arm through the control mechanisms discussed in the next chapter. The system designed can be summarised in Figure FC3.1.

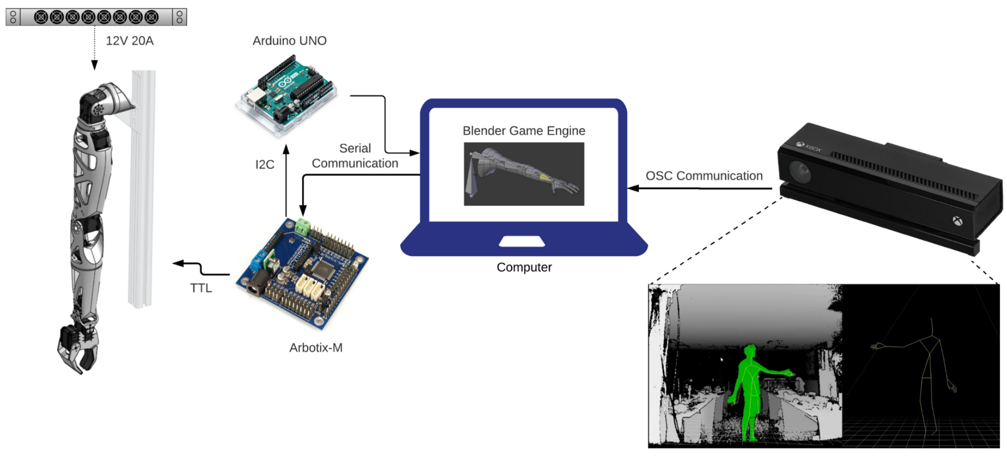


Figure FC3.1: System Design.

First, this chapter reviews the system design. Second, the features of Blender Game Engine are discussed and the implementation of the interface is described. This section consists of two sections - the motion capture interface and the graph editor to analyze the human motion recorded from Kinect respectively. respectively. Next, the design of the kinematic structure of the human arm is modeled in Blender through the process of rigging.

The physical arm design and structure of the robotic arm is derived from [1], and extends into an implementation of a novel control strategy by employing the use of a Kinect for Windows 2 for skeleton tracking and the Blender Game Engine to capture human motion and translate the captured or real-time motion onto the prosthetic arm. This chapter The joints present in the robotic arm used in this study are revolute joints or rotating joints. The study aims to study the kinematics of the 8-DoF robotic arm concerned with the positions and angles of the joints without consideration of the forces or moments that are causing the motion. In particular, how the motion of a particular joint of the arm is related to the motion of the other joint in the arm itself is discussed.

This thesis presents different control strategies to operate the prosthetic arm; tele-operation using Motion Capture (MoCap) using Kinect V2, forward kinematics by sequentially adjusting the orientation of all the joints to attain the desired pose and end-point control through Inverse Kinematics (IK), which converts the values obtained from the 3D configurational space into the actuator space using the Blender Game Engine. The terms human motion refer to the motion of the human arm being tracked and/or recorded using the Kinect.

3.1 Motion Capture and Animation

The process to setup a motion capture system using the Microsoft Kinect V2 is explained in Appendix A in the Figure FA1.1. This section explains the integration of Microsoft Kinect V2 and NI-Mate with Blender Game Engine to achieve real-time motion capture.

3.1.1 Blender 2.78

There are 25 joints of the human body tracked by the Kinect using the NI-Mate software. These joints are represented in the Blender 3D environment as points, referred to as empties, as shown in Figure FA1.8. These empties represent the joint coordinates estimated by the Kinect sensor are mapped onto a bone structure created inside Blender to further analyse any captured or recorded motion.

These empties are connected to form a 3D rendition of a skeleton that will mimic the movement of the arm captured from the Kinect. The addition of the bones to form a 3D model is called Rigging and it simplifies the animation process. Here, the empties obtained from NI-Mate are interconnected to form the digital bones in Blender. The complete rendition of the bone structure in blender is called an Armature and it can consist of many bones. The robot arm is represented as a kinematic chain of such bones (links) connected by revolute joints. These bones can undergo many transformations such as translation and rotation and a combination of these motions associated with the bones will move or deform in a similar way.

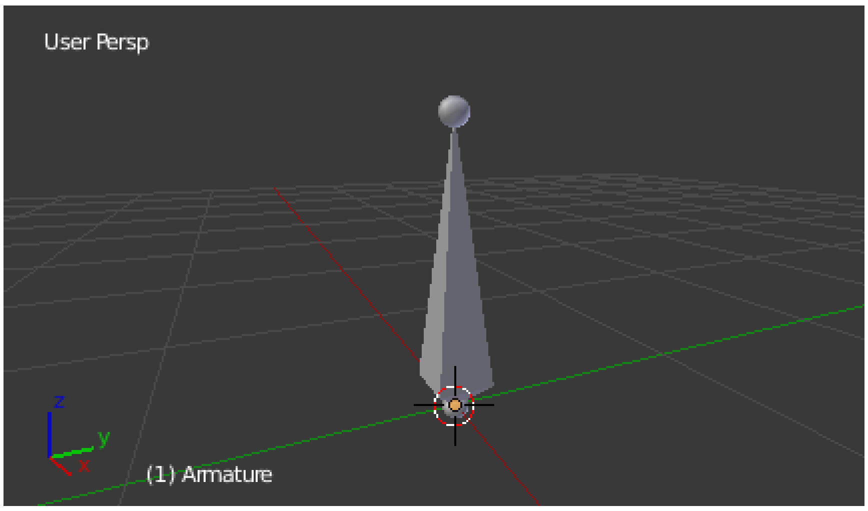


Figure FC3.2: Armature Object in Blender. Image is taken from `https://docs.blender.org/manual/en/2.79/rigging/armatures/introduction.html`

This feature of Blender makes it easier to understand and control the prosthetic arm movement in a 3D environment. The properties of rigging which the study will primarily focus on are Forward and Inverse Kinematics, describing the relationship between the joint positions and the end-effector position and orientation. These features will be discussed in the next section. Blender receives the sensor data (geometric coordinates and orientation) of the joint from the NI mate add-on, transmits it to Blender Game Engine (BGE) to update the virtual environment within Blender. Through the use of such features available in Blender, we can simulate the process of an arm movement captured in real-time.

To create a human model in Blender, we make use of a rig that either can be generated using the Rigify add-on or the NI Mate Mocap Rig. Blender also provides tremendous support for Rigging and Animation. The *Rigify* feature generates an armature with fingers which can be used should there be a way to track fingers. As Kinect does not individually track all the fingers of the body being tracked, the latter rig is used. This rig consists of three collections namely:

- Coordinate Data: The raw XYZ location data from the NI Mate add-on as shown in Figure FC3.3.
- Capture Armature: The interconnected structure of the body used for capturing motion as shown in Figure FC3.4.
- Retargeted Rig: An armature which is manually reoriented to make real-time motion tracking easier and accessible to the links between the joints transmitted to Blender from the NI mate add-on as shown in Figure FC3.5.

When a person is in the Kinect's field of view (FoV), the skeletal data can be transmitted using the NI mate add-on into Blender and the rig mimics the person's motions through tracking the empties. These movements can be recorded in Blender by storing

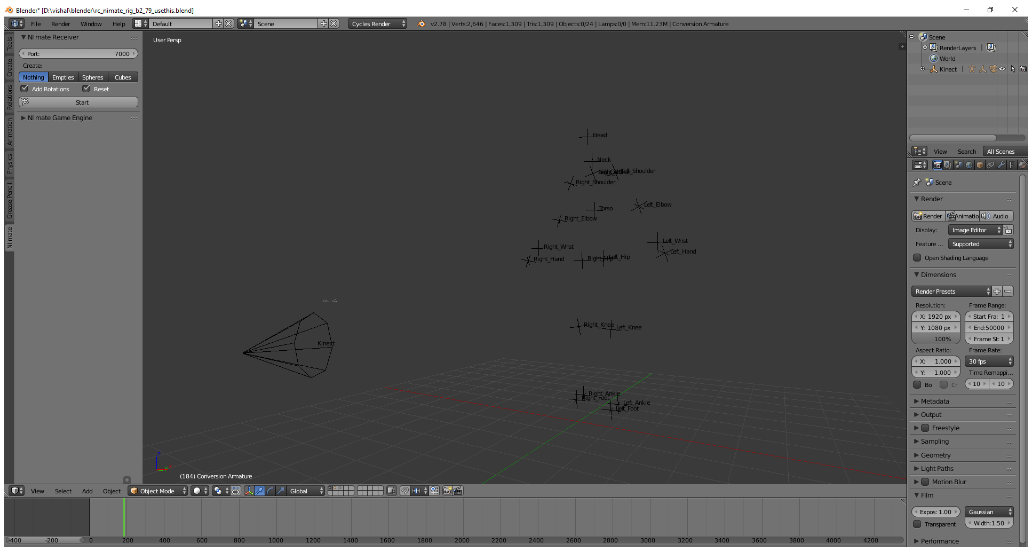


Figure FC3.3: The raw XYZ location data from the NI Mate add-on

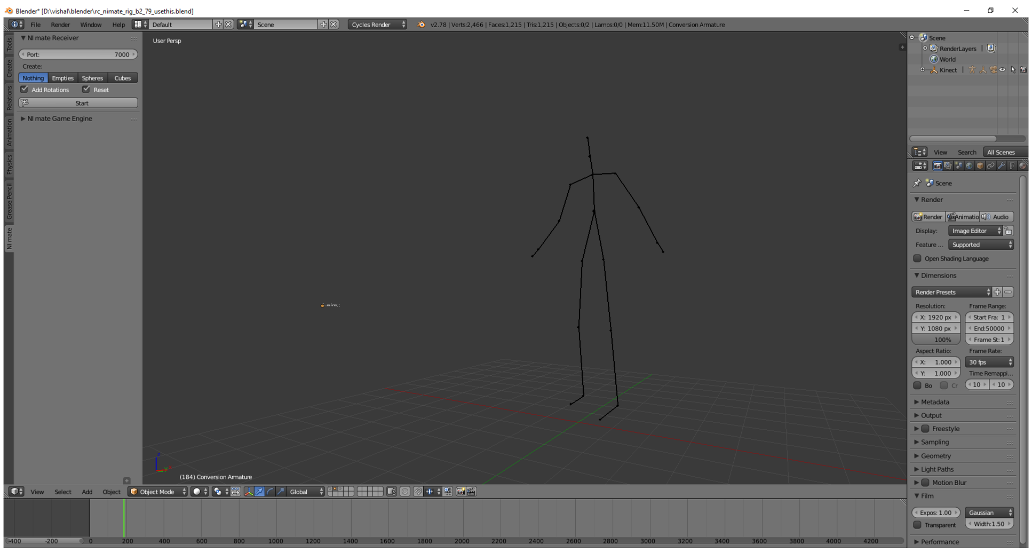


Figure FC3.4: Capture Armature

them as keyframes in 3D space. The recorded motion can be replayed from the first recorded keyframe.

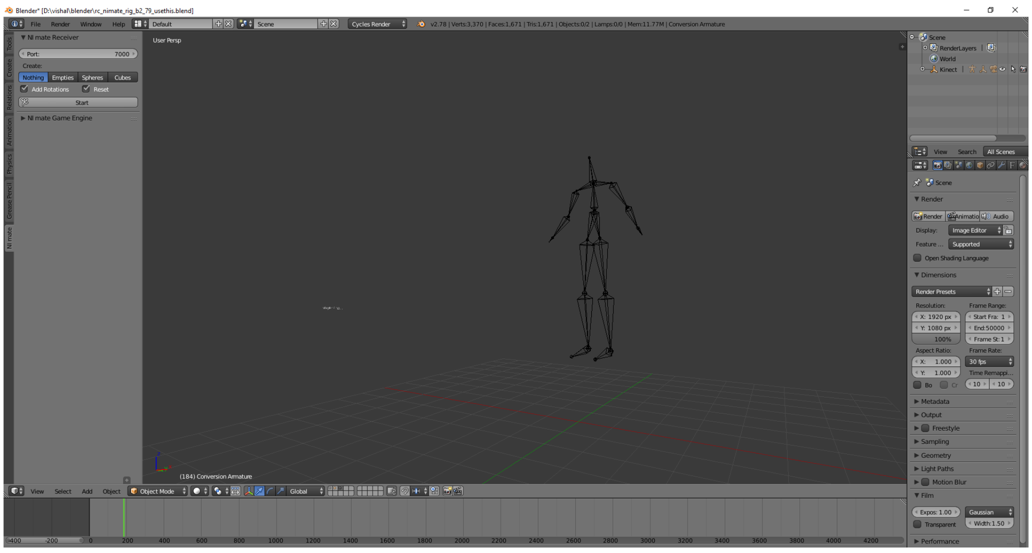
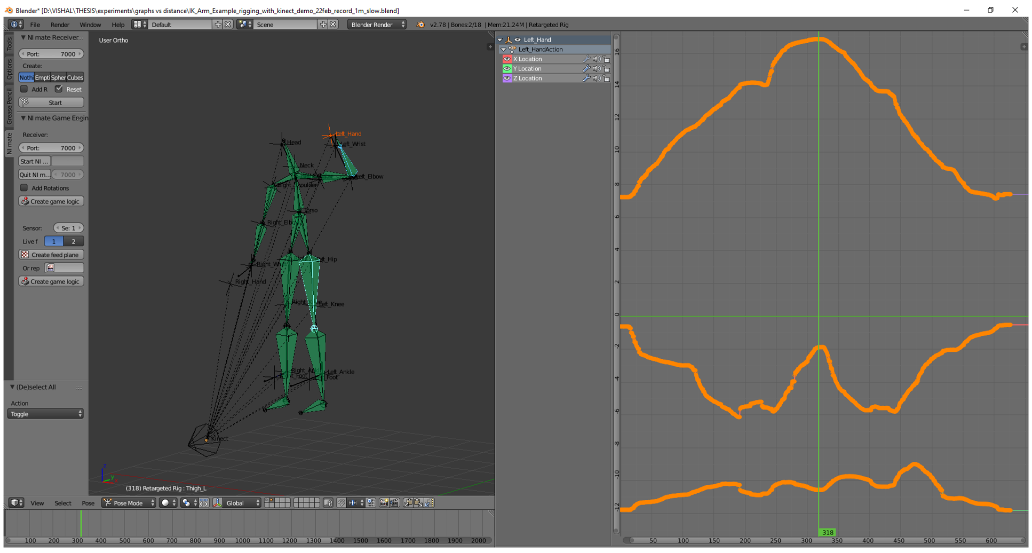


Figure FC3.5: Retargeted Armature

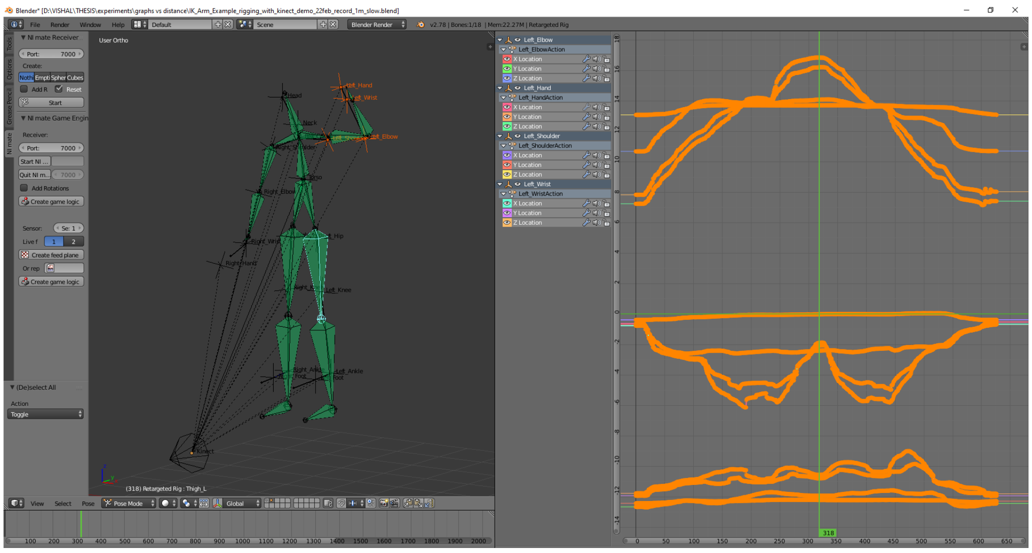
3.1.2 Motion Analysis using Blender

In this section, human motion captured using the Kinect is analysed with the help of Blender. Human Motion Analysis is very important when it comes to quantifying performance and determine any irregularities in a person's recorded motion. Moreover, a certain action replicated by different people, although similar to the naked eye, will produce varied results which will be useful in understanding the variations in kinematics of human motion.

In order to program and control the arm, it is required to know its spatial arrangement and a means of reference to the environment. The coordinates in Figure FC3.3 can be used to plot graphs in the coordinate system with respect to the Kinect sensor as origin. In addition to this, Blender has an inbuilt UI which enables the user to view graphs using the Graph Editor as shown in Figure FC3.6. These graphs can be exported from Blender and can be processed and filtered further to draw correlations between different motions to either analyse a person's movements such as their gait or in fall detection. Blender Game Engine has an embedded Python interpreter which provides access to it's data, objects, classes, and functions. Therefore, Blender's python interpreter allows flexibility in accessing the captured motion data.


Blender* [D:\VISHAL\THESIS\experiments\graphs vs distance\IK_Arm_Example_rigging_with_kinect_demo_22feb_record_1m_slow.blend]
User Ortho
Left_Hand
Left_HandAction
X Location
Y Location
Z Location
(318) Retargeted Rig : Thigh_L

(a) Single Joint Graph


Blender* [D:\VISHAL\THESIS\experiments\graphs vs distance\IK_Arm_Example_rigging_with_kinect_demo_22feb_record_1m_slow.blend]
User Ortho
Left_Elbow
Left_ElbowAction
Left_Hand
Left_HandAction
Left_Shoulder
Left_ShoulderAction
Left_Wrist
Left_WristAction
(318) Retargeted Rig : Thigh_L

(b) Multiple Joints Graph

Figure FC3.6: Graph Editor within Blender Game Engine.

The joint locations as tracked in Retargeted Armature, in Figure FC3.5, have been mapped onto an upper arm rig (see Figure FC3.7) and simulations and data acquisition will be performed on this simplified rig itself. Although the joint tracking from Kinect V2 may seem fairly real-time, the system is susceptible to error due to the variations in accuracy of the joint being tracked as the distance from the sensor increases. From [53], although the Kinect v2 can physically sense depth at a distance of 8 meters, the skeleton tracking range is 0.5 meters to 4.5 meters, and it has trouble finding a skeleton at closer than 1 meter because of the field of view of the camera. However, 4.5 meters is where the system can reliably track body joints. Anything beyond 4.5 meters will lead to inconsistent results in body joints tracking.

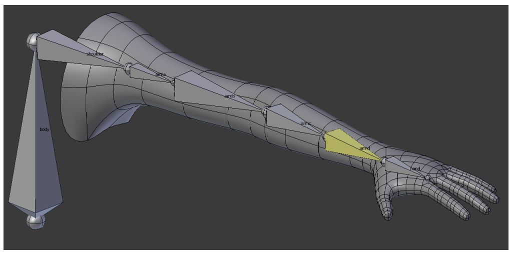


Figure FC3.7: Upper Arm Rig

In addition to variations in tracking with distance, the Kinect SDK does not support individual finger tracking, i.e., the skeletal tracking API by Kinect's official SDK only has the hand joint - no provisions for finger tracking. The skeletal tracing feature in Kinect SDK 2.0 tracks the wrists, hands, index finger tip and thumb tip. Therefore, the prosthetic arm is more suitable for studying of reaching to an endpoint position than manipulation as mentioned in [1], and this thesis will aim to integrate a hand prototype from third-party projects to exploit it's full potential in performing manipulation in addition to reaching and grasping.

Although independent finger tracking is unavailable, the prosthetic arm will be programmed to open and close its end-effector through measuring the distance between the hand tip and thumb tip, and the same will be reflected using Inverse Kinematics in Blender.

To address finger-tracking using Kinect V2, three solutions have been identified. First, using the two joints on each hand, the hand-tip and the thumb, the end-effector can be driven to either close or open completely to represent the closing and opening of a human fist. Second, using Blender features such as Kinector, it is possible to approximate the tracking of independent fingers and then, dedicate servos to individually manipulate a certain finger based on motion tracking from Blender. Finally, an alternate solution from [40] and [4] would be to segment the Kinect's video stream, obtain the contour that belongs to a hand and find the convex hull. Consequently, the edges of the convex hull above the wrist define the fingers.

In this study, we address these challenges by introducing an approach to human pose estimation by detecting and applying the opening and closing of fist as a mode to control the end-effector. Also, the lengths of bones represented in Blender are independent of the person within the FoV of the Kinect's camera to ensure that the system is independent of the physique of the person whose movements are being tracked.

3.2 Rigging with Forward and Inverse Kinematics

Since this study is limited to the analysis of motion of the upper limbs, the idea was to reflect the motions captured by the Kinect onto a specialized rig that can represent the human motions in a more dexterous and realistic manner. This led to the using a upper limb rig from from Blender Documentation as shown in Figure FC3.8 that was modified to represent the actual motions of the person's arm being tracked or by performing endpoint control through Inverse Kinematics (IK) in Blender.

In robotics, we consider two kinematic problems. First, the forward kinematic problem computes the pose of the end-effector of the robotic arm given the angles of all the joints. Second, the inverse kinematic problem determines the joint angles that are required given the pose of the end-effector.

According to [49], these techniques are described as follows:

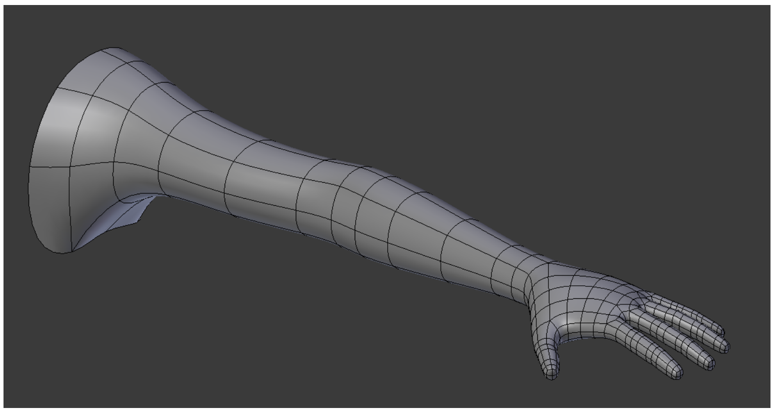


Figure FC3.8: Rendering of a human arm.

- Forward Kinematics (FK): can be defined as the problem of locating the end-effectors' positions after applying known transformations to the chain.
- Inverse Kinematics (IK): can be described as the problem of determining an appropriate joint configuration for which the end-effectors move to desired positions, named target positions, as smoothly, rapidly, and as accurately as possible.

In forward kinematics (FK), the joint angles are the inputs to the links in the kinematic chain, the outputs would be the coordinates of the end-effector(s). On the contrary, in inverse kinematics, given inputs are the coordinates of the end-effectors, the outputs that require calculation are the joint angles.

In Blender, to obtain the desired position, it is required to animate all the bones rotation sequentially to achieve the said position which is preferred in situations where bones require accuracy in their transforms in the local space. This process of sequentially adding transforms in a top-down hierarchy is called Forward Kinematics. Although this process takes longer, it provides the user entire control over the rig. Contrary to FK, from [54], the end-effector is the first thing that is set and the rest of the bone chain is calculated afterwards in the case of IK. According to [55], the process of inverse kinematics involves finding the positions for the joints in the chain to make the chain behave as it is required to. This is usually achieved by rotating the joints to reach the end position for the last link called end-effector.

To visualise the motion execution by the robot, the kinematic chain (or armature) underlying the prosthetic robot arm is modeled using *bones*, and the *mesh* defining the robot's appearance is attached to this kinematic chain (Figure FC3.9).

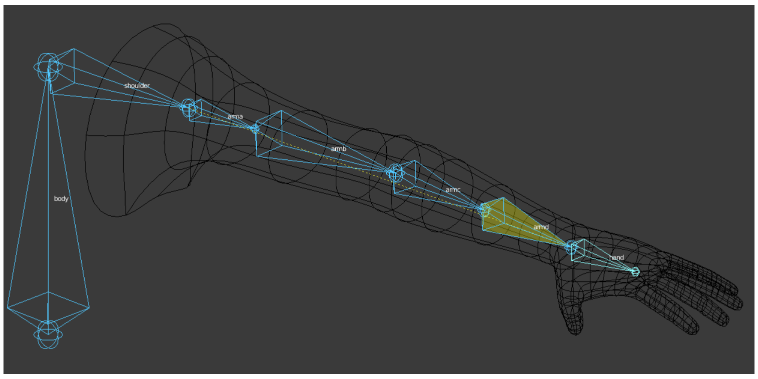


Figure FC3.9: The internal kinematic chain rendered arm

[56] mentions that FK is not very useful when you have a desired end-effector position, but need to know the joint angles required to achieve it. Given a change of angle of any servo actuator, only one effector moves in the chain. But if we are given a change of coordinate, the whole chain of effectors (servos) might have to move a certain angle for the end point to reach the desired position. Therefore, by having the end-effector treated like a constant point in space, it is easier to move bones higher up in the hierarchy and still have the chain end up at the same point.

From [57], Blender allows the user to choose from different possibilities, either by sending end-effector positions or joint angles, to control a kinematic chain. In case the end-effector positions are being sent, the user has the choice between the Blenders internal inverse kinematics (IK) solver functions or his own IK-solver and control the armature based on joint angles and translation vectors.

Although IK is difficult to implement, Blender creates an inverse kinematic solution to rotate and position links in a chains using two IK solvers: Standard IK Solver

(Jacobian) and iTaSC IK Solver; which allows manually setting up bone influences and relationships in the rig. In addition to this, Blender allows IK behaviour up through a controlled number of bones (chain length) in the chain whilst not impeding the individual movement or rotation of any bones not directly contained in part of the IK.

In the current study, the standard IK solver is used in order to calculate the positions of the other bones automatically by positioning the last bone in a bone chain.

The virtual model of the upper arm requires two bones to control the orientation of the entire kinematic chain. The end bone, as shown highlighted in Figure FC3.10, can be manipulated directly and controls the bending of the arm at the elbow in the direction of the Pole target. A pole target is a secondary target for a bone with an IK constraint. The first target, by manipulation of the end bone, is where the chain of bones is trying to get to, and the second target (pole target), is where the chain bends to to get to this target. Therefore, the placement of the pole target is important to decide in which direction should the chain of bones bend.

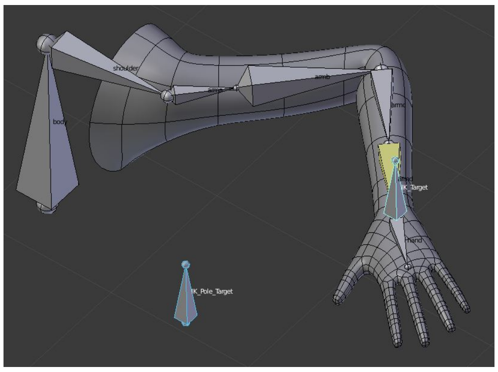


Figure FC3.10: Arm Rig uses two bones - IK_Target and IK_Pole_Target. The IK_Target enables the chain to be repositioned when it is manipulated.

When dealing with robotic arm animation, the vertices of different sections of the arm (mesh) are assigned different vertex groups with that of the bone so that they will move exactly with their bone. The mesh deformations occur when there are differences between the weights of different vertices on the arm which is shown in the Figure

FC3.11. Moreover, through the process of weight painting, the model can be accommodated with finger bones that allows us to have finer control by being able to control the fingers, proving to be very feasible for animation and control purposes. The weight painting process is further explained in Appendix C, in the Figure FA3.1, Figure FA3.2 and Figure FA3.3.

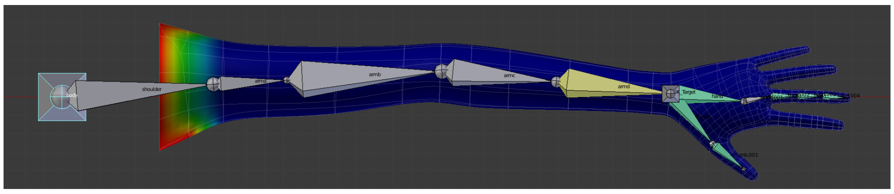
3.3 Robot Design, Hardware and Electronics

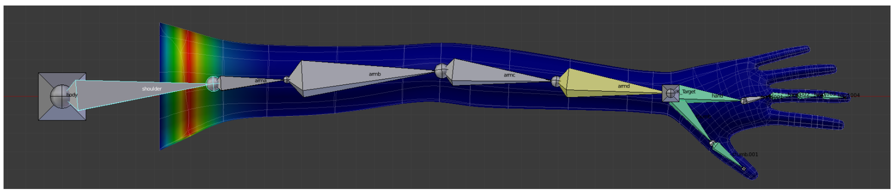
Once the rig is set to record and capture motion or to control the arm using Inverse Kinematics from Blender, the joint angle values from the rig will be sent to a micro controller through serial communication from BGE (see Figure FC3.1). As Blender provides Python modules such as *Pyserial*, serial communication is feasible from Blender to the ArbotiX-M micro-controller. This controller will drive the actuators which act as joints in the prosthetic arm.

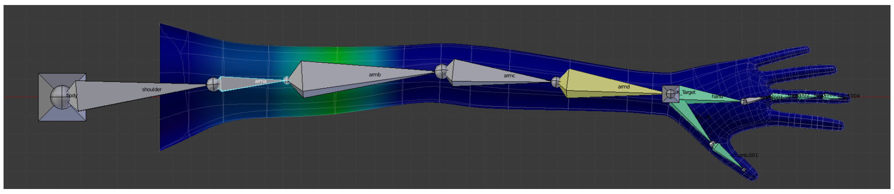
3.3.1 Reachy's Arm

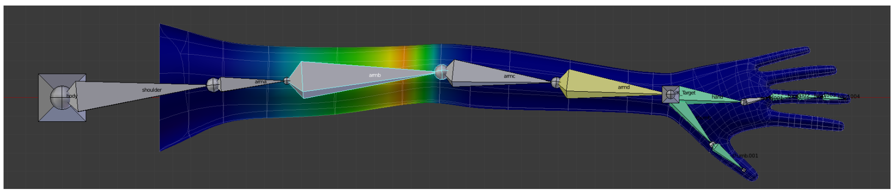
The standard model of the robotic arm, Reachy [1], is a 7-DoF prosthetic robotic arm with each of them actuated by a dedicated motor. In this study, Blender's Python environment is used as an interface to translate the data to an Arduino compatible board, the Arbotix-M. The arm is 3D-printed using Poly Carbonate material weighing nearly 1.4 kilograms and measuring 60 cm from shoulder to wrist [1]. However, with the addition of gripper to provide 8-DoF, nearly 70 cm-radius hemisphere is centered on its shoulder joint.

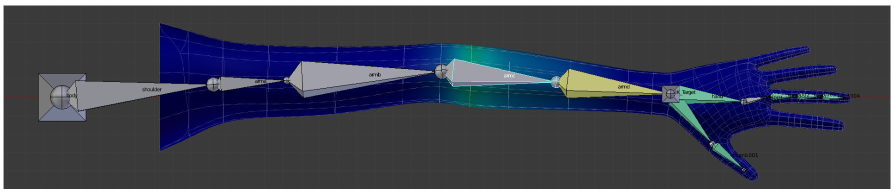
From [1], the first three motors perform three consecutive rotations as shown in Figure FC3.13: shoulder flexion-extension, shoulder abduction-adduction, and humeral rotation. The fourth motor controls the elbow flexion-extension and the fifth motor

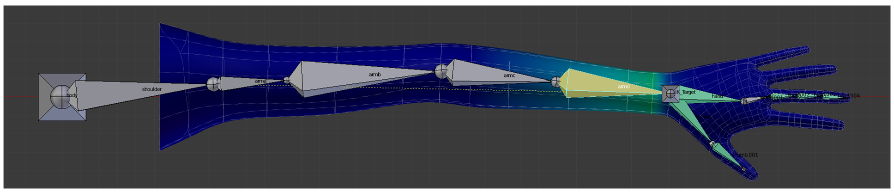

(a) ’body’

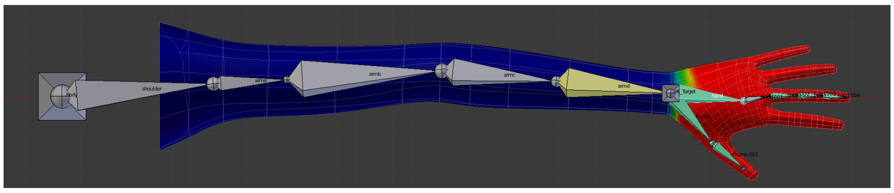

(b) ’shoulder’


(c) ’arma’

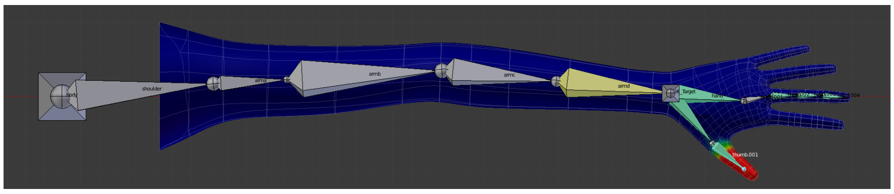

(d) ’armb’


(e) ’armc’


(f) ’armd’



(g) ’hand’



(h) ’thumb’

Figure FC3.11: Bone influence on the mesh deformations using weight painting.

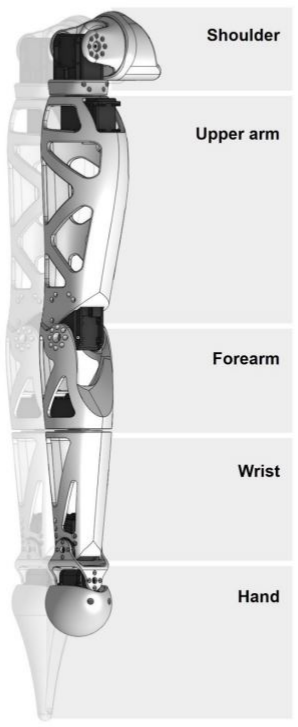


Figure FC3.12: Reachy Robotic Arm. Image taken from [1].

is responsible for the forearm’s pronation-supination. The sixth and the seventh motors operate at the wrist joint by performing consecutively radial-ulnar deviation (also known as abduction-adduction) and flexion-extension. The last motor controls the gripper at the end of the prosthetic arm which performs grasping tasks.

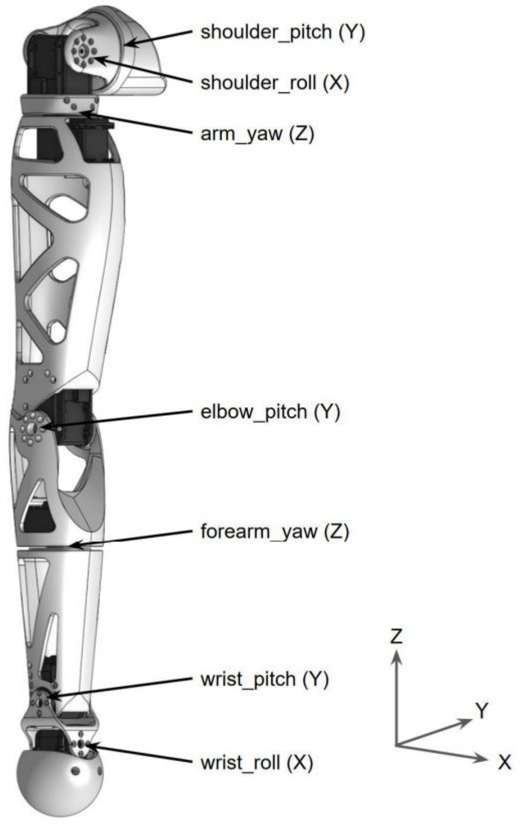


Figure FC3.13: Reachy comprising of seven independent DoF. Image taken from [1].

3.3.2 Hardware

3.3.2.1 Dynamixel Actuators

The robotic arm's design employs the use of Dynamixel servomotors to actuate the seven DoFs present on the arm.

There are three different models of Dynamixel motors used in [1]'s standard version. The shoulder's flexion-extension is controlled by the MX-106 AT while the other DoFs and the elbow joints are controlled using the MX-64 AT. All the other joints are operated by the AX-18A as these joints do not require as much power as the previous joints. In addition to this, the AX-18A's are lighter and smaller than MX servomotors. Therefore, the prosthetic arm's weight distribution decreases towards the end-effector.

The shoulder-to-elbow complex of the robotic arm is operated by the MX Dynamixel series as these four joints support the heaviest loads while the robot is put in motion. The forearm and wrist joints, which do not require as much power, are operated by AX-18. Therefore, the robotic arm's weight distribution gradually decreases towards the distal end (end-effector). These actuators have load and temperature sensors embedded within to automatically trigger resting phases. Also, these motors are allow the individual tuning of an internal Proportional-Integral-Derivative (PID) controller, moving speed, angle limits, maximum torque and they can be daisy chained to reduce wiring in the system. The values that these actuators provide as the feedback are Present Position, Instantaneous Speed, Load applied, Voltage levels, Temperature of the motor, and the current that is being consumed by the motor.

The process of setting up the Dynamixels and the components required to reconfigure the Dynamixels are explained in Appendix B.

3.3.3 Electronics

As the Dynamixels consume very high amounts of current, they are powered by a 12V x 30 A SMPS. The Dynamixels are daisy-chained using three pin connectors, out of which the **Input Voltage (VIN)** and the **Ground (GND)** are connected to the power supply (SMPS), while the Data line is untouched.

The MX-106 AT is connected to the micro-controller, Arbotix-M, which is plugged into a computer using a FTDI-USB Cable. The prosthetic arm is then manoeuvred by passing values through a serial port with a software interface provided by Blender Game Engine (BGE) using Arbotix-M micro-controller (see Figure FC3.14),to communicate with Dynamixel servomotors i.e., sending motor commands to drive the robot arm or retrieving data from embedded sensors. The FTDI cable is a USB to Serial (TTL level) converter allows for programming the board as well as to connect TTL interface devices such as the Dynamixel servo motors to USB interfaces. There is an XBee socket which allows for wireless serial communication to the ArbotiX-M micro-controller by plugging in an XBee. The XBee socket shares a serial port with the FTDI port, so only one of those can be used at a time. Therefore, the ArbotiX has 2 serial ports. One serial connected to both the FTDI header and the XBEE port and the other used for the Bioloid bus for TTL communication. In addition to this, the board also has an I2C port which was used to retrieve data from the Dynamixel actuators to analyse the operation of the Dynamixel motors, either ni real-time or by applying a certain recorded motion onto the arm.

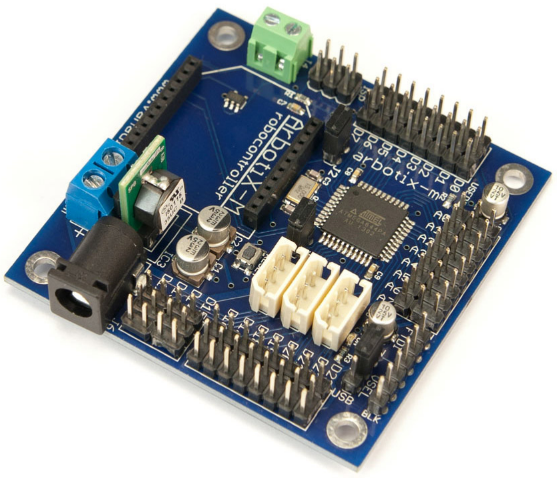


Figure FC3.14: The ArbotiX-M micro-controller. Image taken from Trossen Robotics.

The Arbotix-M is powered using an external power supply (12V, 5A). As the serial ports are used up in programming and in communicating with the Dynamixel Servos, I2C communication with another Arduino board is implemented to receive motor feedback as shown in Figure FC3.15.

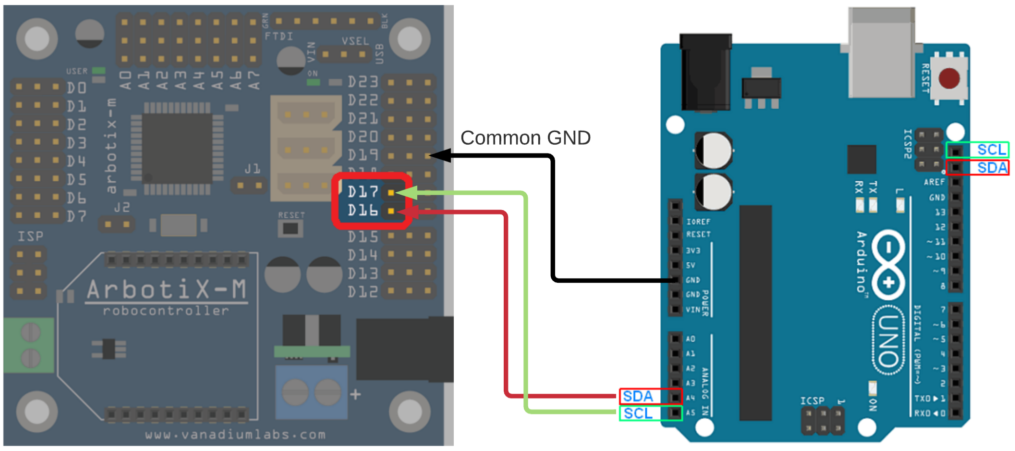


Figure FC3.15: Arbotix-M and Arduino I2C communication.

CHAPTER 4

EXPERIMENTS

Variations in Tracking Joints with Distance using Motion Capture

To determine the variations during tracking through motion capture in this study, a simple task to vary the **supplement** of carrying angle as shown in Figure FC4.1, the angle between the longitudinal axis of humerus and ulna. These actions was repeated over varied distances ranging from 1 meters away to 4 meters away from the sensor in steps of 1 meter.

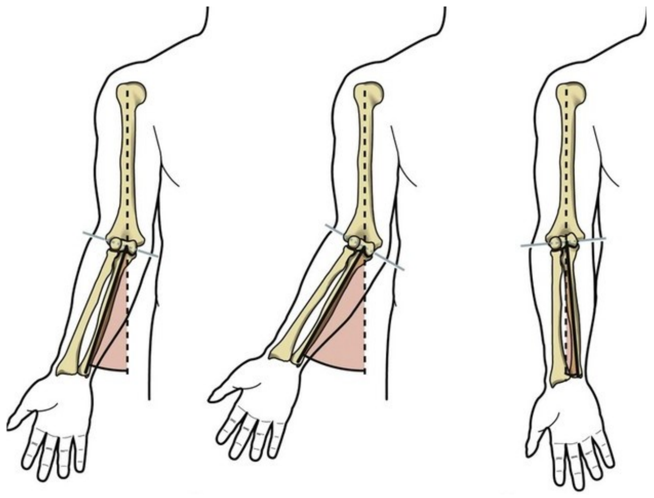


Figure FC4.1: Carrying angle. Image taken from [6].

A person was taken as the test subject who will be repeating the above mentioned tasks over varying distances ranging from 1 to 4 meters away from the Kinect sensor.

During the measurement of the supplement of carrying angle, i.e, 180° - carrying angle, the variations in the values of the joint angle was either due to the time delay in executing the action was either delayed or early, or due to the inconsistency in maintaining the pose for a certain duration of time during the repetition of the tasks over different distances resulting in either a lag or a lead in the action graph as shown in Figure FC4.3. The joint trajectory paths in the recorded skeletal tracking data is shown in the Figure FC4.2.

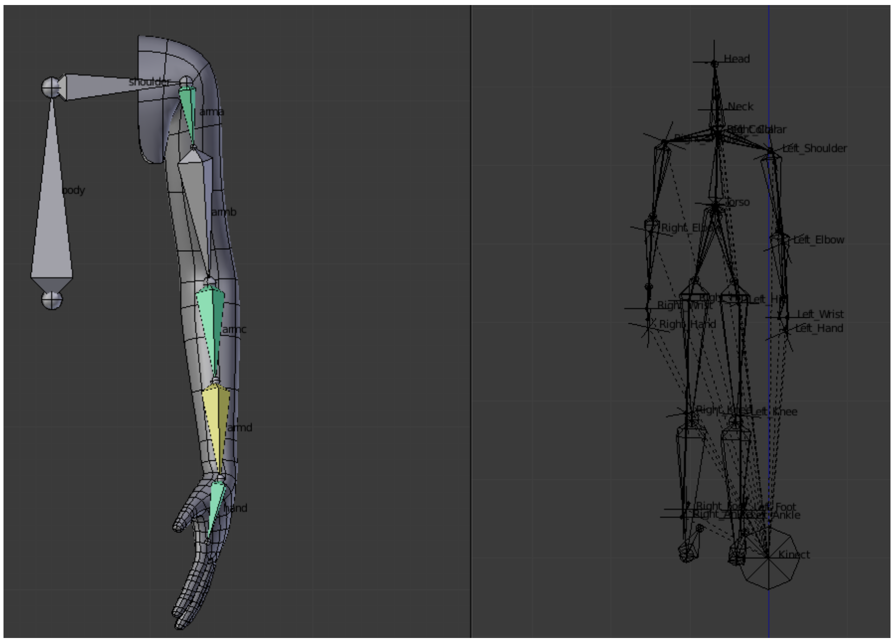
The variation of the joint angles when performed the similar action across various distances from the sensor can be seen in Figure FC4.4. The correlation of the average measurement value of joint angle is highest for distances which are 2-3 meters away from the Kinect sensor.

These angles are computed by transforming the joint coordinates with Kinect sensor as the origin in to vectors along the shoulder-elbow and elbow-forearm axes and performing the scalar product of these two vectors. These joint coordinates are also applied onto the graphic model present in the blender and the joint angle values are computed to check whether there are no errors in remapping the joint coordinates onto the armature. The variation in remapping the joint coordinates obtained from Kinect to compute the angles on the arm model in Blender is tabulated in the Table TC4.2.

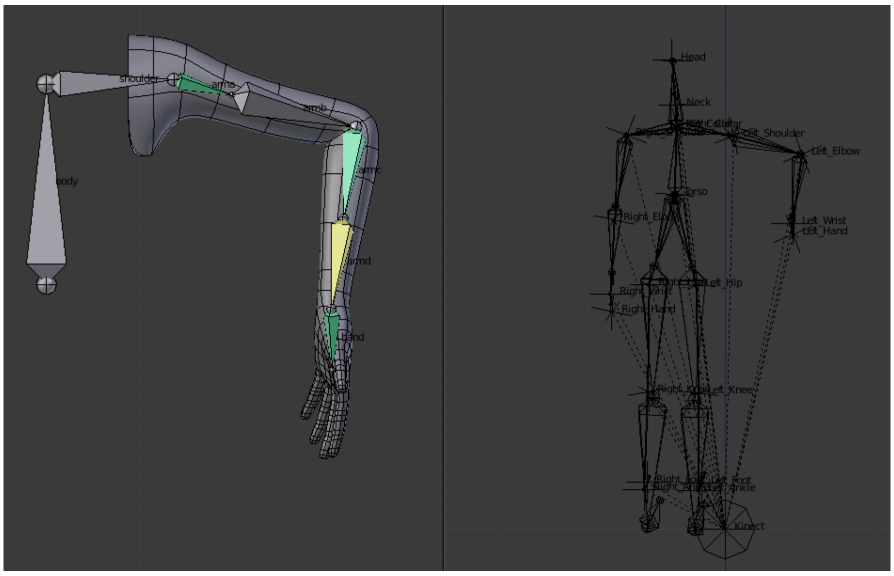
Table TC4.1: Difference in measurement of carrying angle with the average versus distance away from the Kinect Sensor.

Distance from sensor (meters)	Minimum (from average)	Maximum (from average)
1	-19.794817°	25.183552°
2	-14.351645°	21.518491°
3	-18.783611°	9.822832°
4	-14.183169°	13.938318°

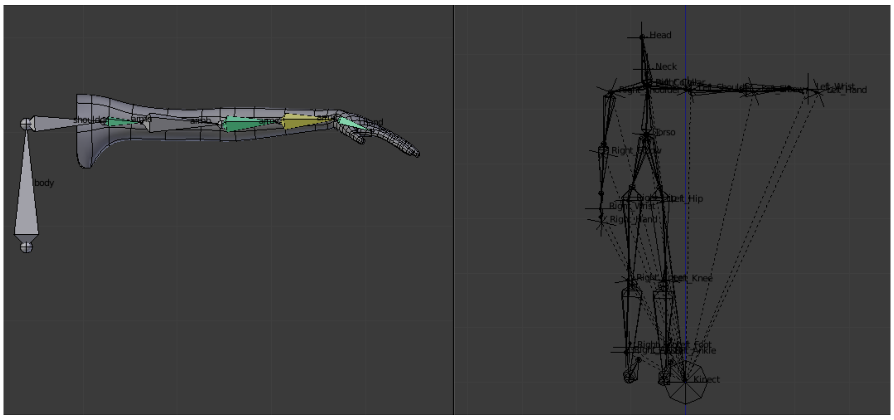
From the Table TC4.1, it is seen that for better tracking and for the minimum range of variations, the motion capture using Kinect v2 is better suited for measurements ranging from 3-4 meters. The higher peaks and very large variations are most likely


shoulder
arma
body
armb
armc
armd
hand
Head
Neck
Left_Shoulder
Torso
Right_Elbow
Left_Elbow
Right_Wrist
Right_Hand
Left_Wrist
Left_Hand
Left_Knee
Kinect

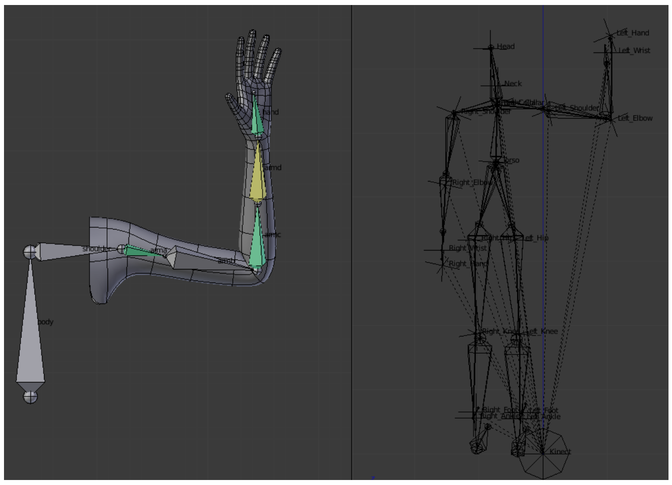
(a)


shoulder
arma
armb
body
armc
armd
hand
Head
Neck
Left_Shoulder
Left_Elbow
Torso
Right_Elbow
Left_Wrist
Left_Hand
Left_Hip
Right_Wrist
Right_Hand
Left_Knee
Kinect

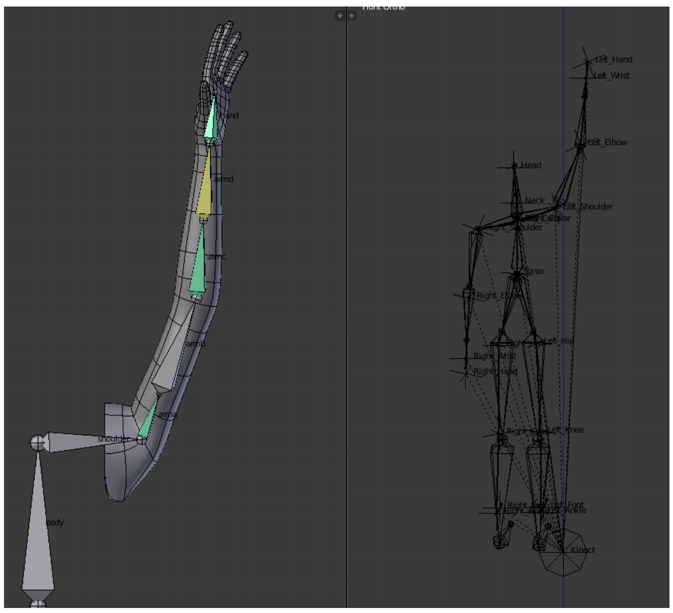
(b)


body
Head
Neck
Torso
Left_Hip
Right_Hand
Left_Knee
Kinect

(c)



(d)



(e)

Figure FC4.2: Skeletal Tracking Recorded Motion trajectory.

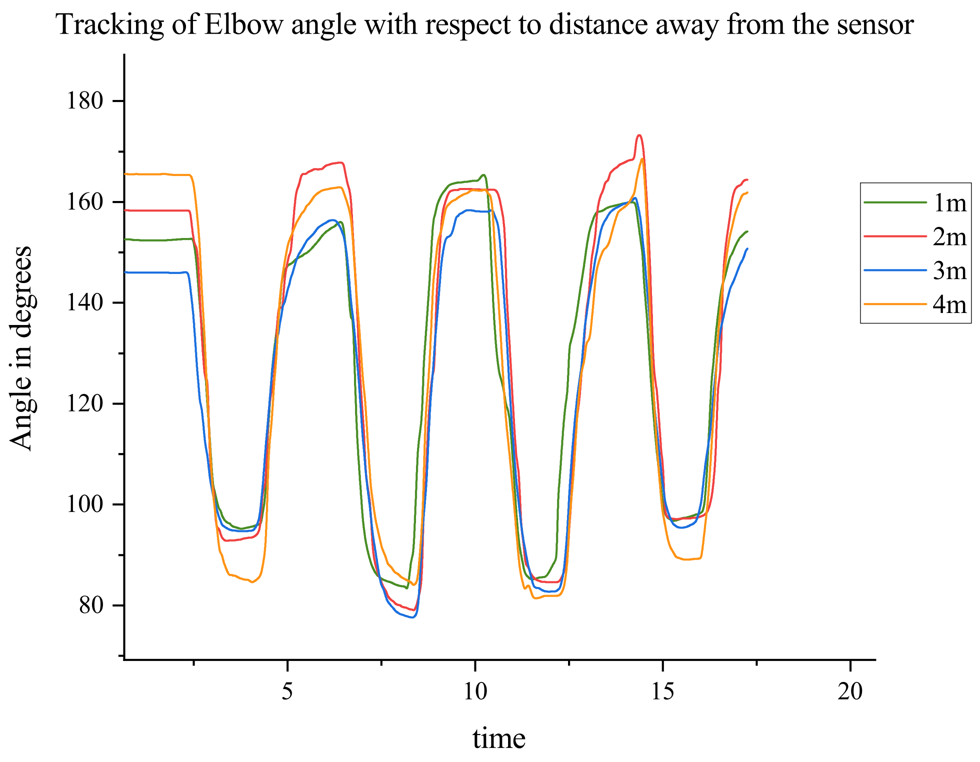


Figure FC4.3: Tracking of Elbow joint angle against distance from the Kinect sensor.

due to the contribution of noise as detected by the sensor.

Table TC4.2: Variation in remapping the joint coordinates obtained from Kinect to compute the angles in Blender.

Distance from sensor (meters)	Difference in angles after remapping (degrees)
1	0.778695
2	0.780936
3	0.776116
4	0.782673

Experiments

The forward and inverse kinematics of the 8 DoF robotic arm are simulated in Blender. Forward kinematics of manipulator allows in estimation of robot workspace

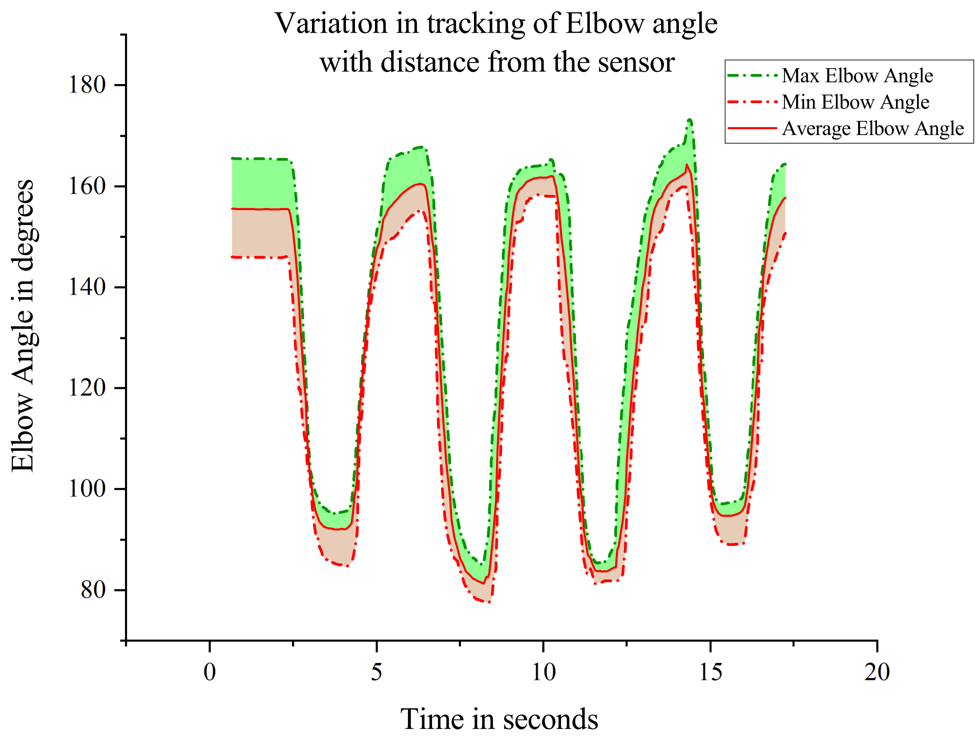


Figure FC4.4: Variation in tracking of Elbow angle with distance from the sensor

and in knowing the singular regions during operation, while the inverse kinematics of the manipulator would be needed for real-time control task. In particular, the relationship between the joints in the robot's arm and the pose of the robot's end effector are determined. As the various joints in the arm are moved, the pose of the end-effector of the arm changes. The concept of forward and inverse kinematics was used to determine the end-effector position for fixed joint angles, and the joint angles for a fixed end-effector.

The rest pose of the arm (with every bone's local axes) is shown in Figure FC4.5. All the simulations performed in Blender, including animations and recording of motion is performed at 24 frames per second.

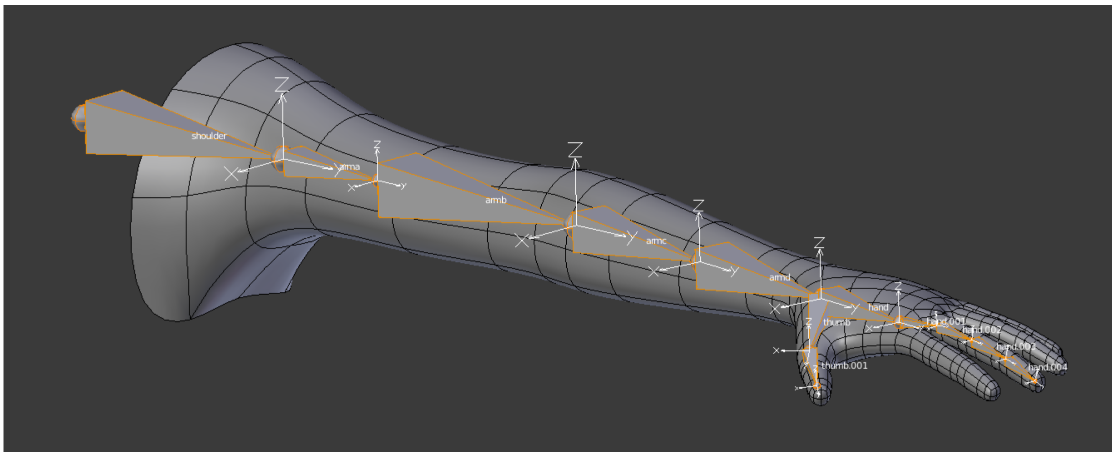


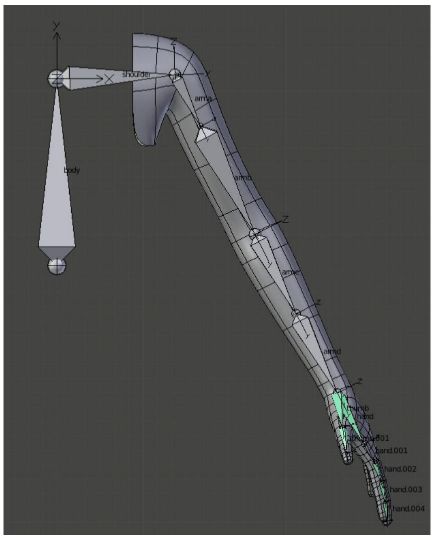
Figure FC4.5: Rest pose of the armature. ('Edit' mode in Blender)

4.1 Control Strategy 1: Forward Kinematics

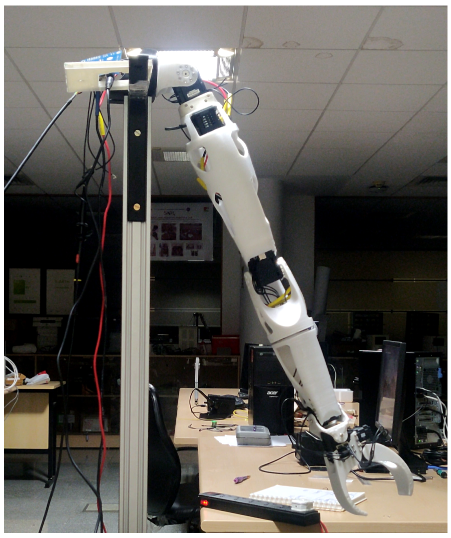
Here, the object that is to be animated is the arm as shown in the Figure FC4.6. The end effector's (i.e, the joint labelled as **hand**) position and orientation in the configuration space would be calculated from the angles of the shoulder, elbow, and the wrist joints. For instance, in order to achieve the position shown in the Figure FC4.6l, the arm has to undergo transformations manually from the shoulder to the wrist in the sequence as showing in Figure FC4.6.

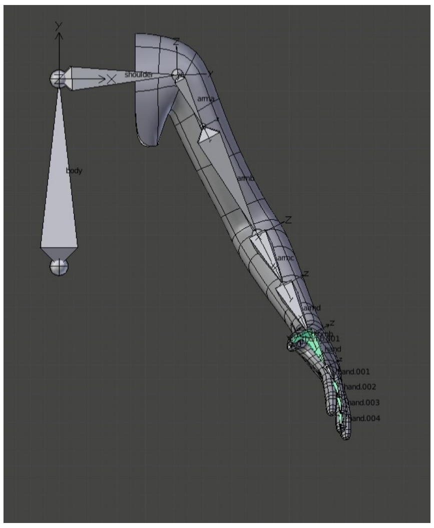
By forward kinematics, the desired position of the end effector would be achieved through setting bones sequentially starting from the root bone until the end bone is reached. When each parent bone is moved, its child bone would inherit its location and rotation making precise changes harder as one traverses the kinematic chain. Therefore, the forward kinematic problem to control the robotic arm is straightforward as the angles of the i^{th} joint are calculated with respect to the $(i-1)^{th}$ joint in order to pose the end effector with precision.

As the bones were oriented sequentially, their respective Euler orientations are plotted and shown in the Figure FC4.7. The angles are with respect to the rest position of

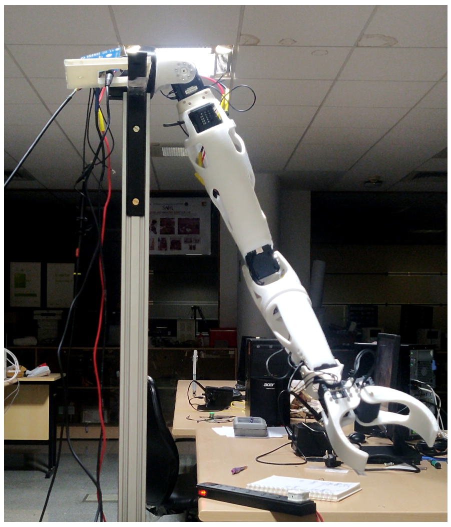


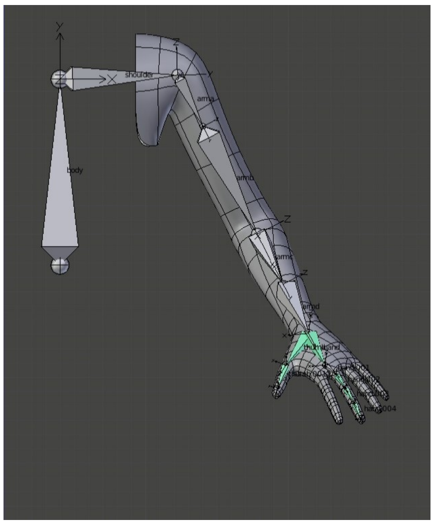
(a) Blender - Shoulder Roll


(b) robotic arm - Shoulder Roll

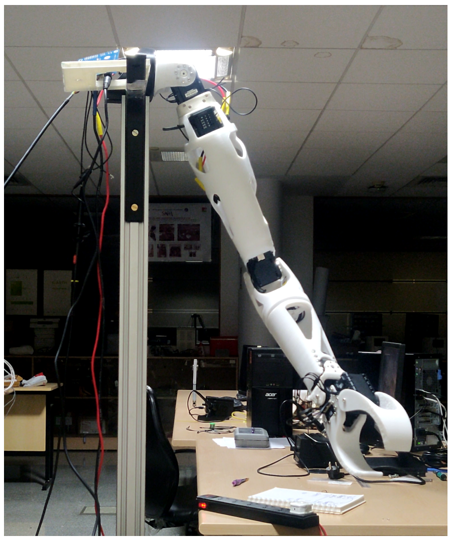


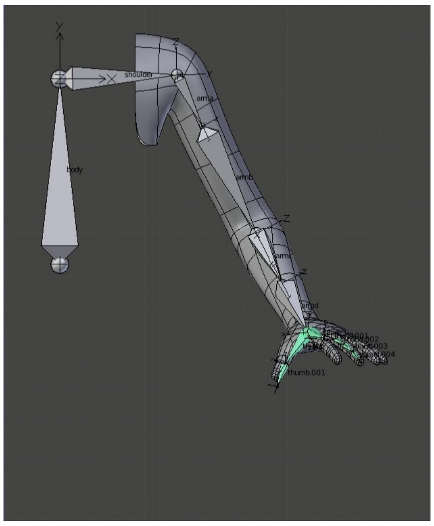
(c) Blender - Elbow Rotation

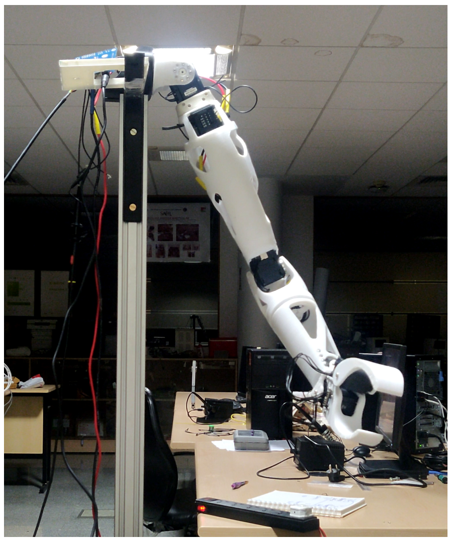

(d) robotic arm - Elbow Rotation

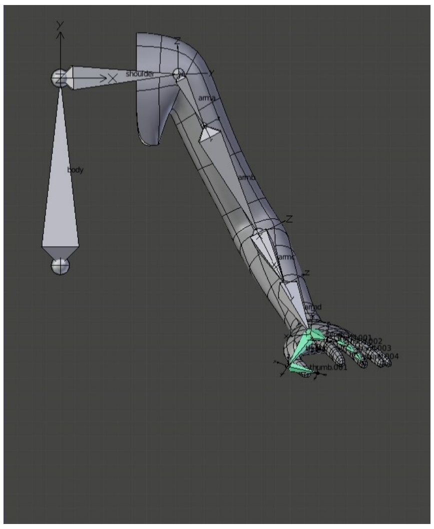


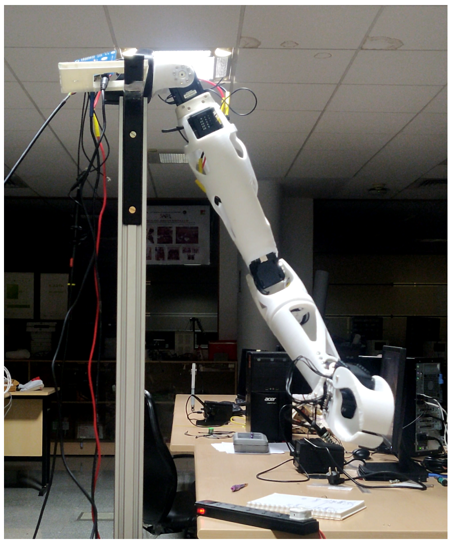
(e) Blender - Wrist Pronation

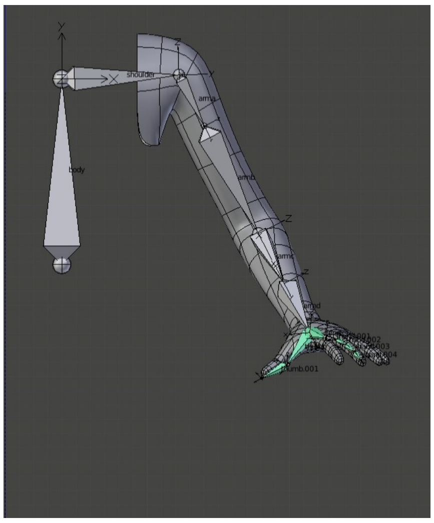

(f) robotic arm - Wrist Pronation


(g) Blender - Wrist Flexion and Extension


(h) robotic arm - Wrist Flexion and Extension


(i) Blender - Gripper (Closed)


(j) robotic arm - Thumb (Closed)


(k) Blender - Gripper (Wide Open)

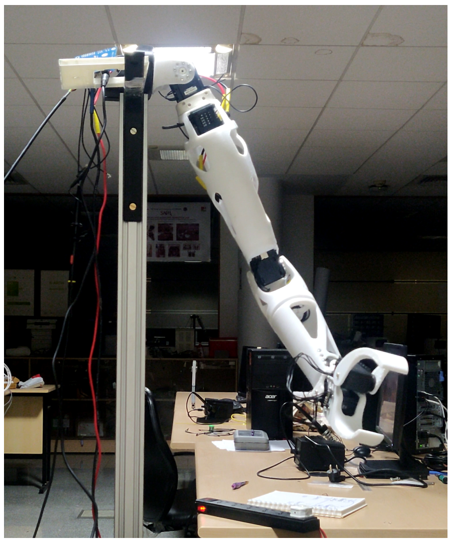
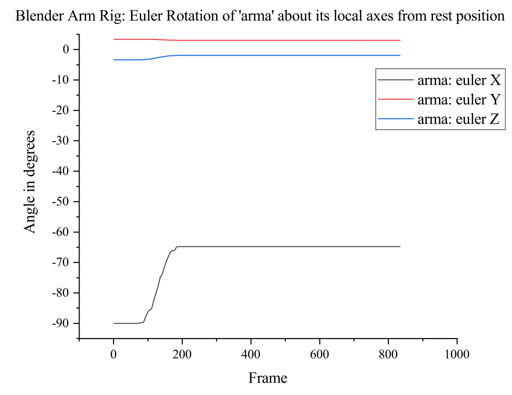
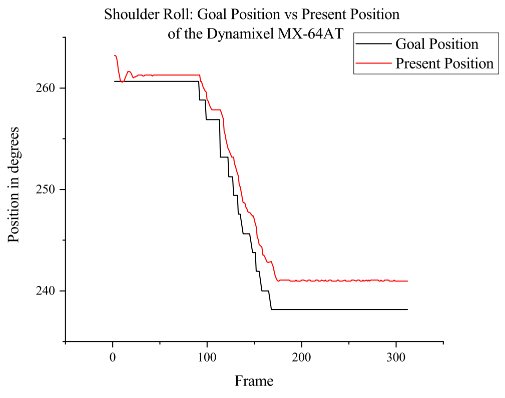

(l) robotic arm - Thumb (Wide Open)

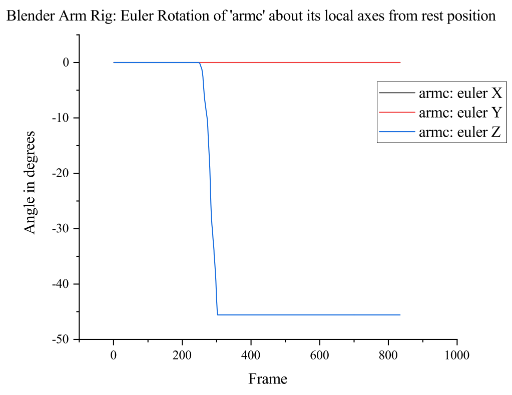
Figure FC4.6: Comparison of poses between the virtual arm in Blender and Robotic Arm.



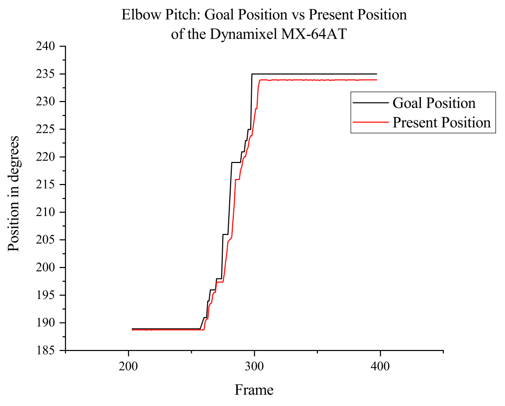
(a) *arma*



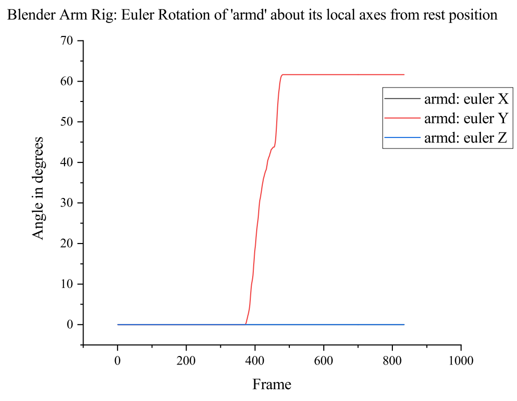
(b) Shoulder Roll of the arm against *arma* Euler X.



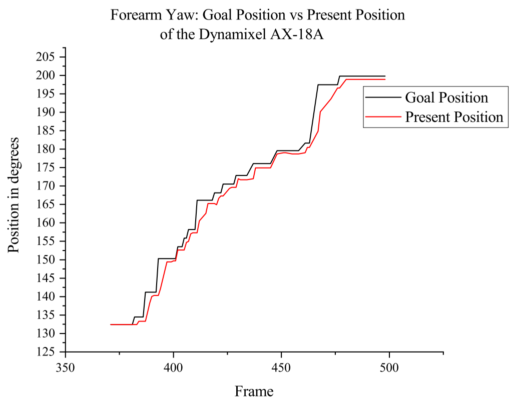
(c) *armc*



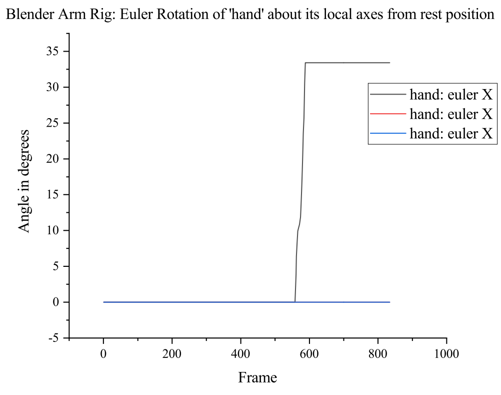
(d) Elbow Pitch of the arm against *armc* Euler Z.



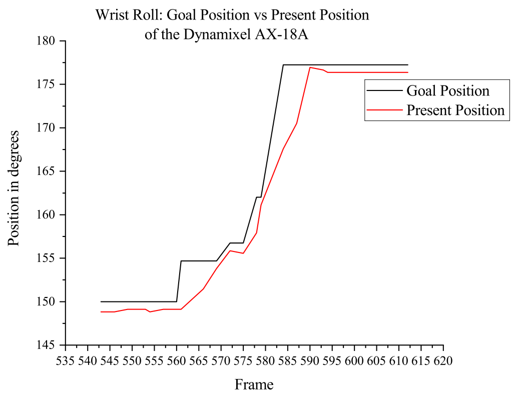
(e) *armd*



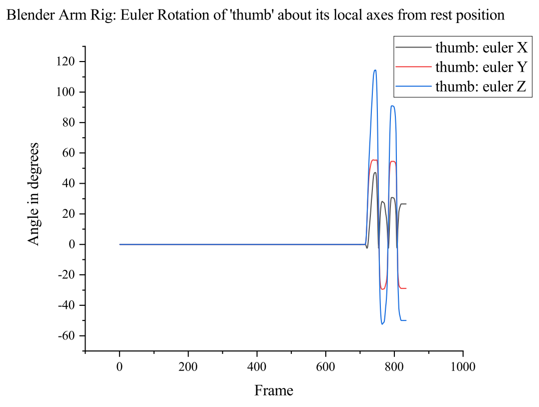
(f) Forearm Yaw of the arm against *armd* Euler Y.



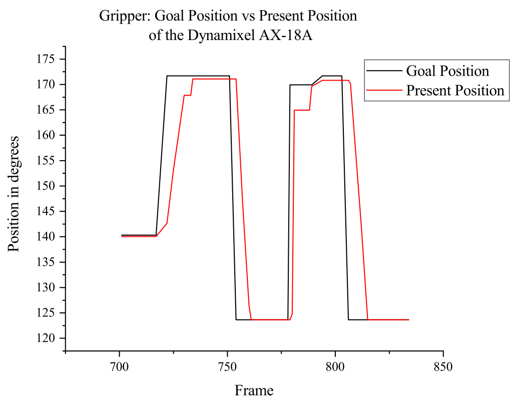
(g) *hand*



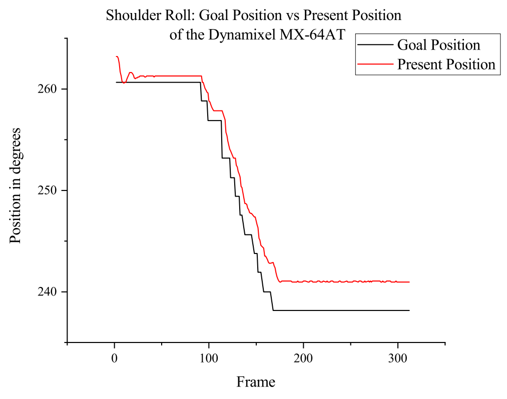
(h) Wrist Roll of the arm against *hand* Euler X.



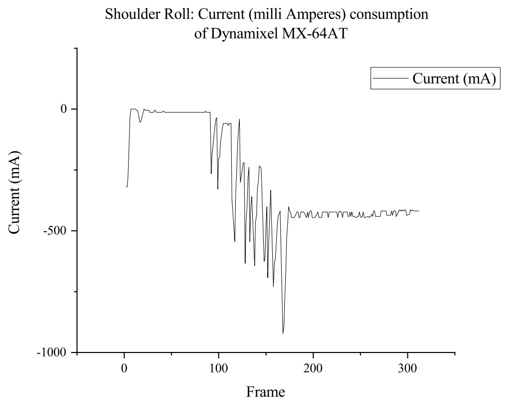
(i) *thumb*



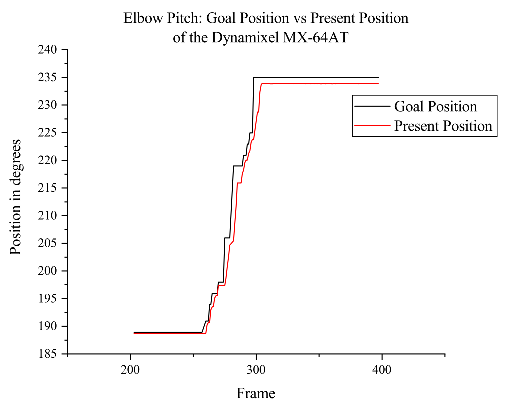
(j) Gripper of the arm against *thumb* Euler Z.



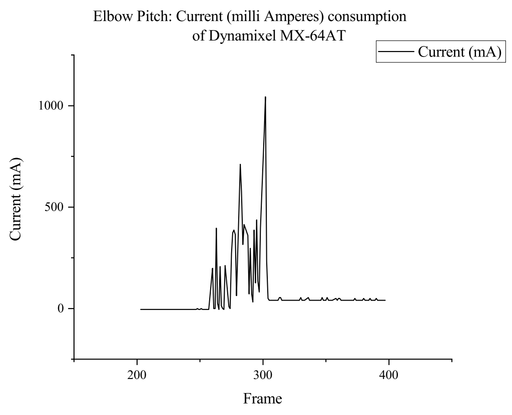
(k) Rotation of Shoulder Roll Dynamixel in degrees.



(l) Current consumption of Shoulder Roll Dynamixel in milli Amperes (mA).



(m) Rotation of Elbow Pitch Dynamixel in degrees.



(n) Current consumption of Elbow Pitch in milliamperes.

Figure FC4.7: Blender Arm Rig: Forward Kinematics graphs for individual bones with the positions of their respective actuators present on the robotic arm.

the arm which is shown in the Figure FC4.5. As all the bones are positioned down the hierarchy, the final end-effector position is achieved. In Figure FC4.7i, the highs and the lows are characterised by the flexion and extension of the *thumb* bone respectively in the Blender environment.

The Denavit-Hartenberg parameters for the current robot arm (till the wrist joint excluding elbow-forearm yaw link) are mentioned in Appendix D.

However, forward kinematics has its disadvantages albeit being easy for basic animation purposes. In order to animate the arm, the system follows a ”top-down” system of rotation making the user follow a hierarchy by first rotating the shoulder, upper arm, elbow, then the forearm and finally the hand itself until the end-effector is in the desired place.

In addition to this, if the end-effector is required to stay in place while rotating other joints in the arm, rotating the entire hierarchy of the kinematic chain would result in the position and orientation of the end-effector to change needlessly. Also, rotating the foremost joint in the hierarchy (the shoulder) while keeping other joint orientations intact would result in the hand’s (the end effector) orientation to change.

In order to overcome these challenges, the second control method describes the implementation of inverse kinematics in Blender.

4.2 Control Strategy 2: Inverse Kinematics

The inverse kinematic problem is defined as the problem of determining the joint angles that are required given the pose of the end effector. The inverse kinematic problem is solved through Blender's standard IK solver allowing the last bone to be positioned and the other bones in the kinematic chain will be positioned accordingly. Unlike the efforts taken in forward kinematics to sequentially position the joints to achieve a desired end effector position, through IK translating a target bone will activate inverse kinematics and rotate the target's parent bone, and the target's parent's parent, and so on, to follow the last bone in the hierarchy. One of the features of the human joints is that they have a limited range of motion that it can rotate. As not all the joints can rotate 360° around their local axes, such as the elbow joint, the bones can be deformed if proper constraints are not added. Along with angle constraints to limit rotation of the elbow joint, a secondary bone (called as Pole Target) is added which is responsible for where the chain bends to get to the target bone.

Given a kinematic chain of 7 bones as shown in the Figure FC3.10, the chain length is set to **4** starting from **armd** to **arma**. This mesh in Blender uses two bones to overcome the twist problem for the forearm: the **IK_Pole_Target** and the **IK_Target** bone. The position and the orientation of the pole target will determine the orientation of the elbow joint. The orientation of the **IK_Target** is mapped to the orientation of the *hand* bone as it is mapped to the robotic arm's end-effector.

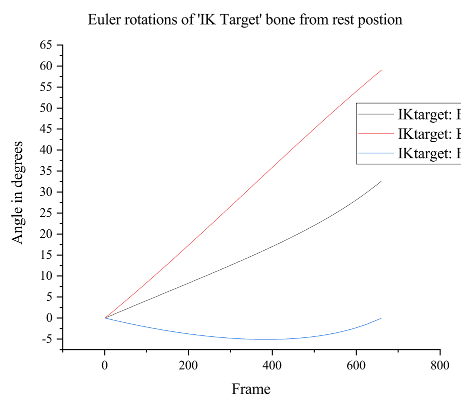
Therefore, inverse kinematics in Blender rotates the bones of the kinematic chain (or armature) into position according to the two control bones (target, pole). In other words, instead of computing the individual rotations required for the shoulder, elbow, forearm and the wrist joints to get the hand (end-effector) into position, the control of the hand joint within Blender would adjust the rotations of other joints in the body causing the rest of the arm to follow.

As the target bone is positioned, the position and the orientations of the joints in the arm are calculated through Blender's standard IK solver i.e, the Jacobian Inverse IK Method. Although this method is powerful, it may be computationally expensive as the Jacobian Matrix columns increases with the number of joints in the articulated body. The arm's yaw is controlled using the pole control bone in addition to controlling individual joints like in forward kinematics for higher degree of control and precision. An overview of the Jacobian IK is described in Appendix D.

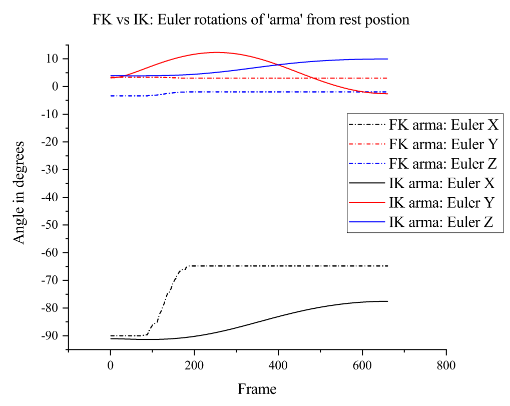
4.2.1 End-point control using Inverse Kinematics

In order to understand how inverse kinematics arranges the bones, from the bottom of the kinematic chain's hierarchy to the top i.e, from the hand to the shoulder bone as seen in Blender, the tail of the hand bone is taken as the target bone's location in the 3D space. Also, in the forward kinematics problem, the end-effector (the *hand* bone) reached the desired orientation in about 660 frames. This duration to reach the desired location can be controlled with inverse kinematics. We can either reach the same location in 660 frames or less. However, for comparison purposes, the IK target bone is made to move from the initial position to the final position of hand bone as seen in FK.

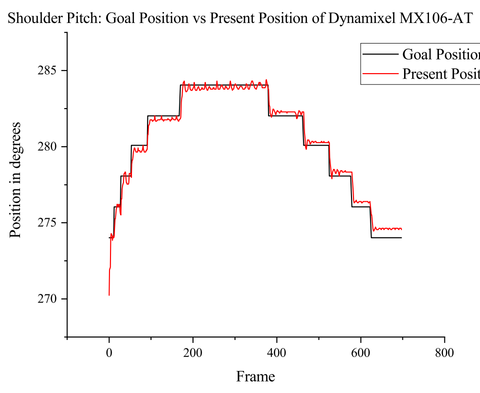
The process of animating the position of the target bone can be performed either in a linear interpolation (constant speed) or Bezier interpolation where inserted frames that comply with a Bezier curve. The differences between the two different interpolation sequences are shown in Figure FC4.9. The position and orientation of the IK target bone, during different methods of interpolation, varied as shown in the Figure FC4.9a



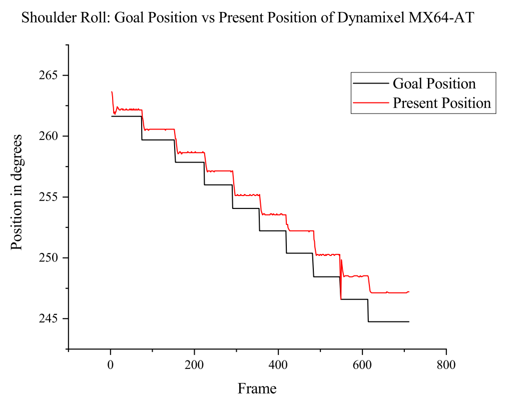
(a) IK Target bone rotations.



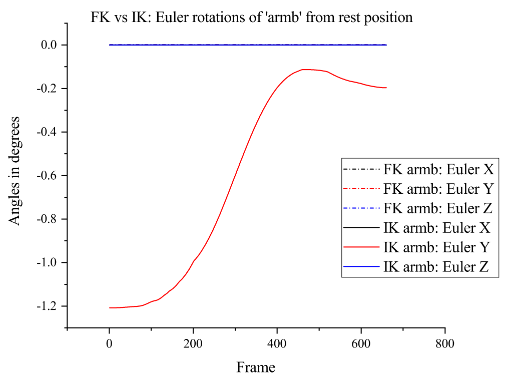
(b) FK vs IK: *arma* bone rotation comparison.



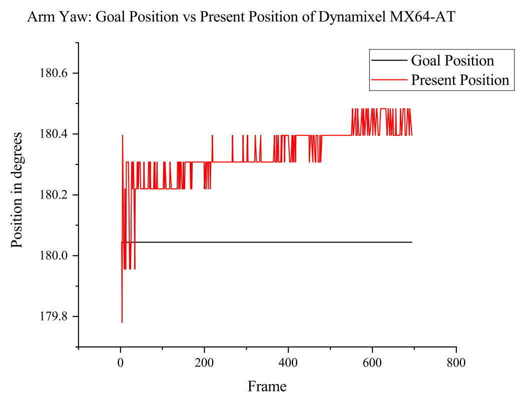
(c) Euler Y rotation versus Shoulder Pitch Dynamixel of the arm.



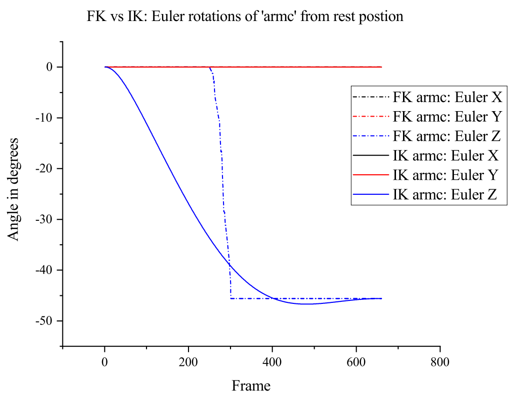
(d) Euler X rotation versus Shoulder Roll Dynamixel of the arm.



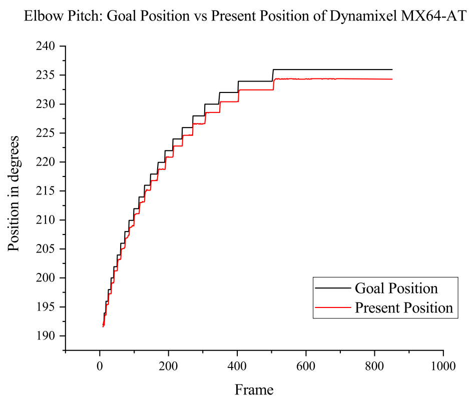
(e) FK vs IK: *armb* bone rotation comparison.



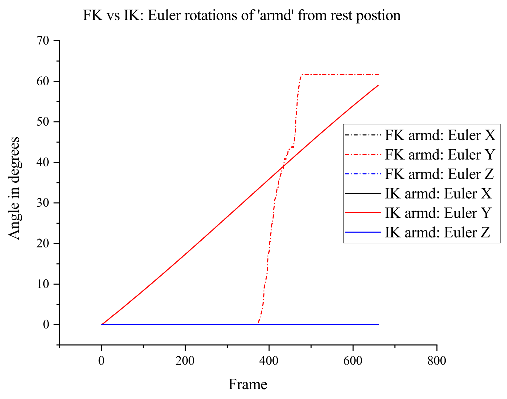
(f) Euler Y rotation versus Arm Yaw Dynamixel of the arm.



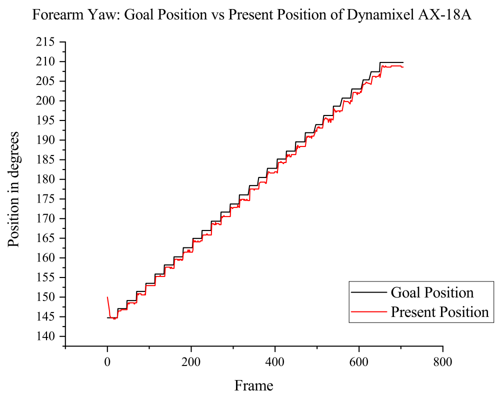
(g) FK vs IK: *armc* bone rotation comparison.



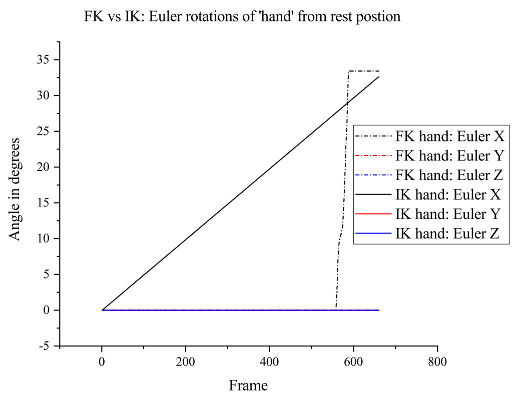
(h) Euler Z rotation versus Elbow Pitch Dynamixel of the arm.



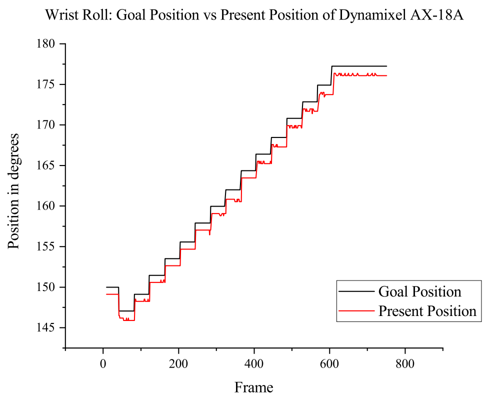
(i) FK vs IK: *armd* bone rotation comparison.



(j) IK: Euler Y rotation versus Forearm Yaw Dynamixel of the arm.

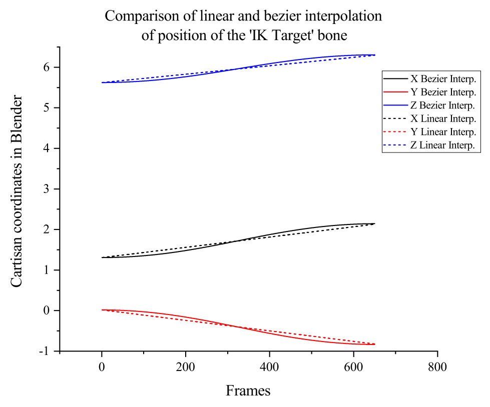


(k) FK vs IK: *hand* bone rotation comparison.

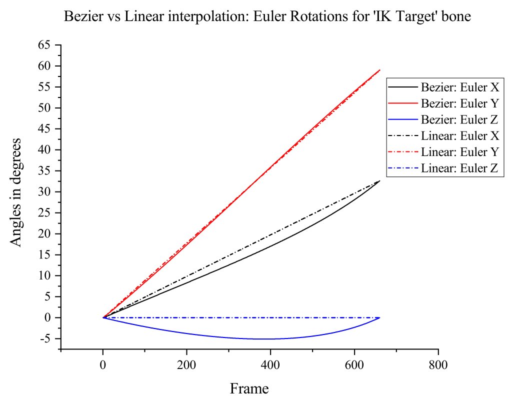


(l) IK: Euler Y rotation versus Wrist Roll Dynamixel of the arm.

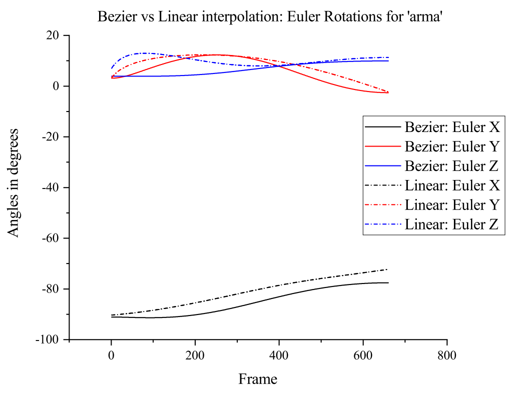
Figure FC4.8: Inverse Kinematics: Comparison of bone orientations between FK and IK.



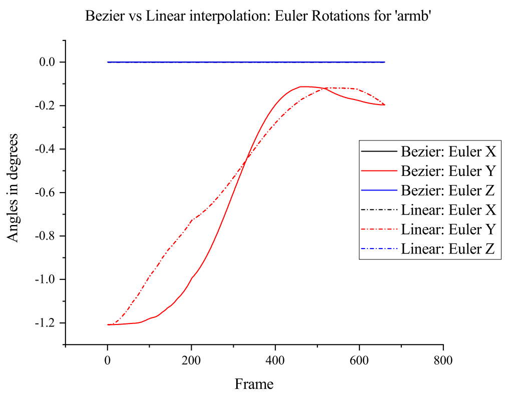
(a) IK Target bone positions.



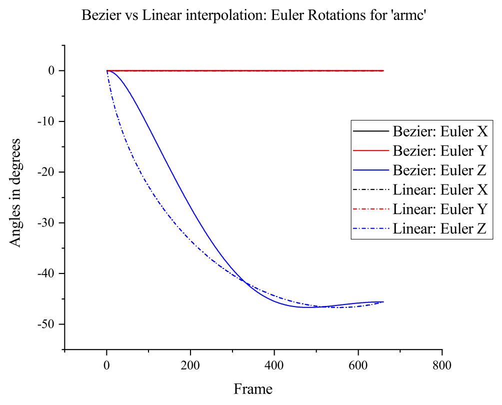
(b) IK Target bone orientations.



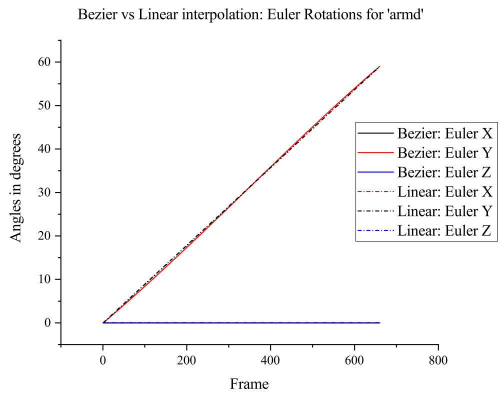
(c) *arma* orientation.



(d) *armb* orientation.

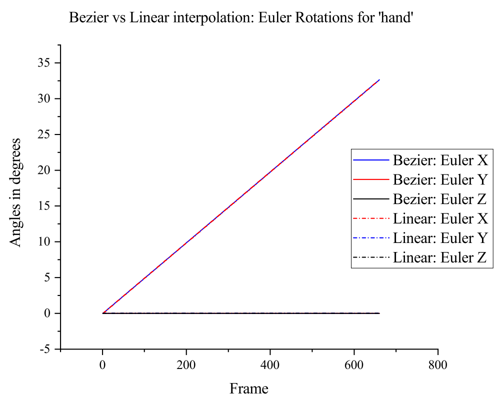


(e) *armc* orientation.



(f) *armd* orientation.

and Figure FC4.9b.



(g) *hand* orientation.

Figure FC4.9: Comparison of bone orientations during Linear and Bezier Interpolation through Inverse Kinematics.

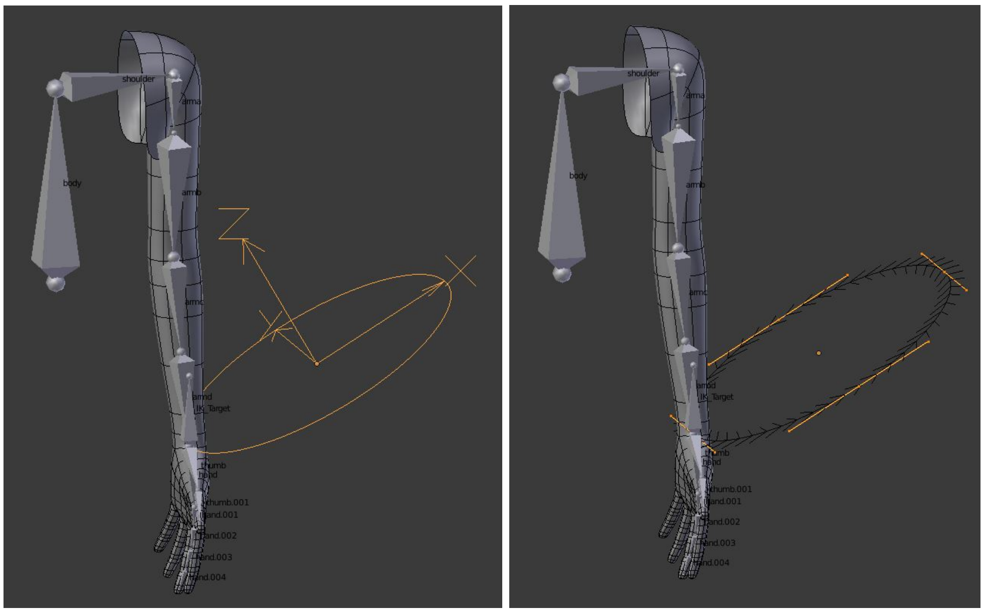
4.2.2 Path Following using Blender features

The positioning and orientation of the target bone can also be made to follow a path within Blender over a period of time by enabling the **Follow Path** constraint and inserting keyframes for the ***offset*** property in Blender for animation purposes.

The arm is made to follow a circular path rotated 30°about its Y axis as shown in the Figure FC4.10. The end-effector was made to go around the path once in 27.5 seconds (660 frames at 24 frames per second) in Experiment 1. In Experiment 2, the end-effector was made to go complete the path twice in 27.5 seconds and the feedback data from the motors was analyzed. The motion sequence from the experiment is seen in Figure FC4.12.

The motor feedback from the shoulder joints and the elbow are compared with the orientations as seen in Blender (see Figure FC4.11).

Another experiment was conducted to analyze tele-operation applications by performing a simple task of reaching to an object, using the gripper to grasp it, change the orientation of the object and then dropping it at another location. This task was per-



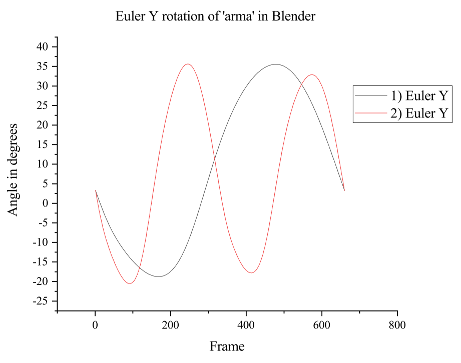
(a) Orientation of the path in Blender. (b) *Direction of the path.*

Figure FC4.10: Inverse Kinematics: Path Following - Circle

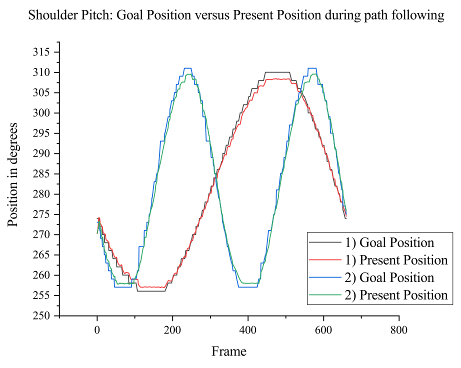
formed ten times consecutively to analyze the ability of the robot's end-effector reach the desired position and orientation during the each execution of the task. The sequence of motion of the arm and its representation in the Blender Game engine is shown in Figure FC4.13. The feedback from the motors is collected and plotted in Figure FC4.14.

For analyzing purposes, certain sections of the proposed motion are investigated to determine the pose deviation from the desired position.

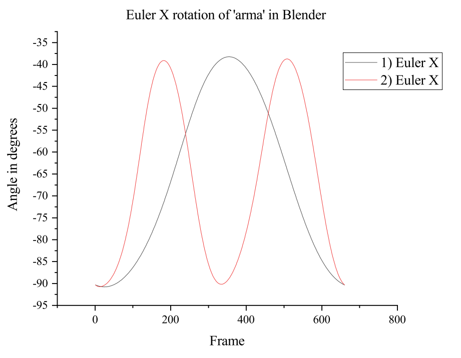
Therefore, through IK the end effector position can easily be achieved as the joint angles are computed by performing vector algebra within Blender before communicating the angle values to the micro-controller using serial communication.



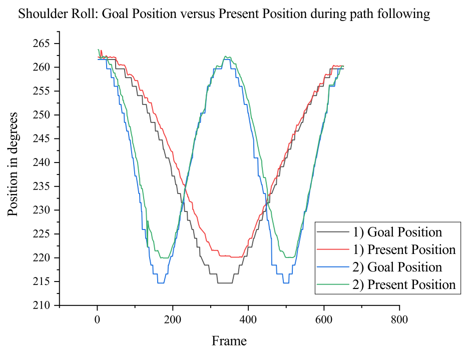
(a) Euler Y rotation of ’arma’.



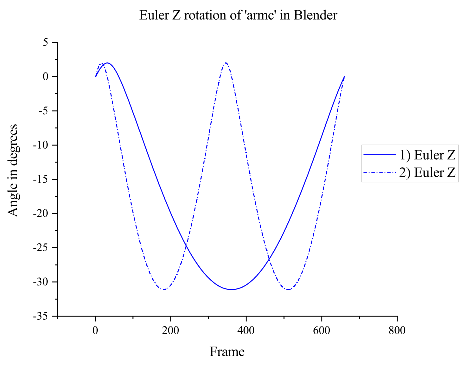
(b) Shoulder Pitch Dynamixel.



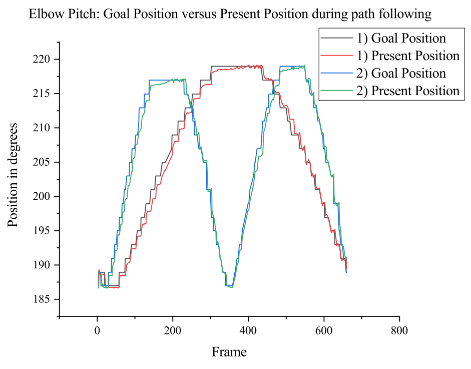
(c) Euler X rotation of ’arma’.



(d) Shoulder Roll Dynamixel.

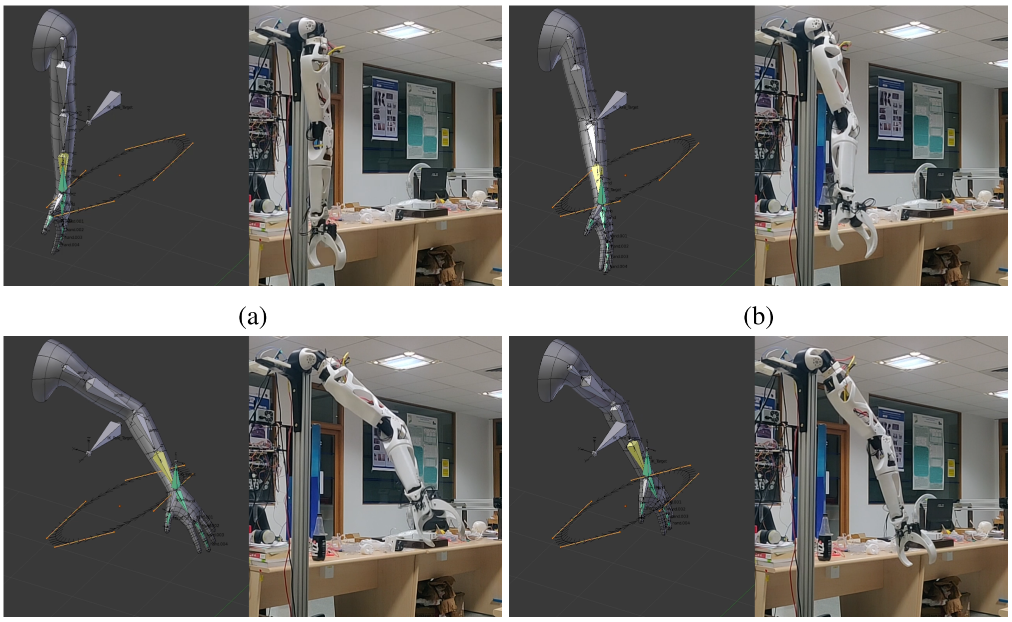


(e) Euler Y rotation of ’armc’.



(f) Elbow Pitch Dynamixel.

Figure FC4.11: Inverse Kinematics: Path Following in Cirlce - Goal position versus Present position during motion of the arm.

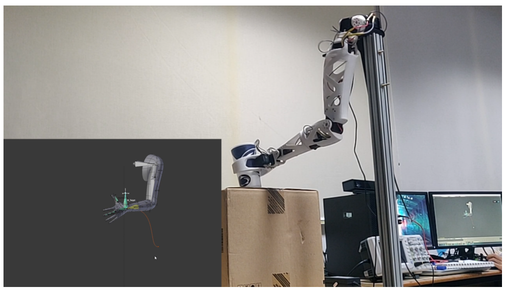


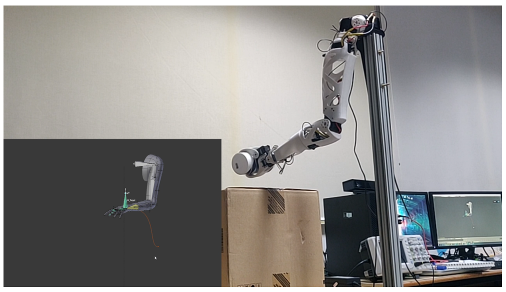
(a) (b) (c) (d)

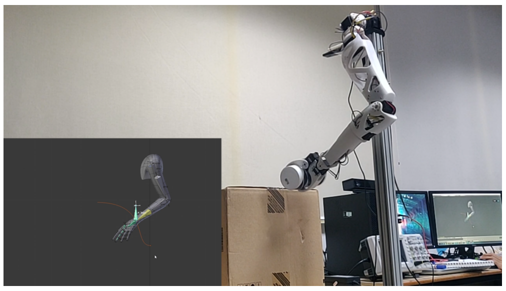
Figure FC4.12: Inverse Kinematics - Path Following.

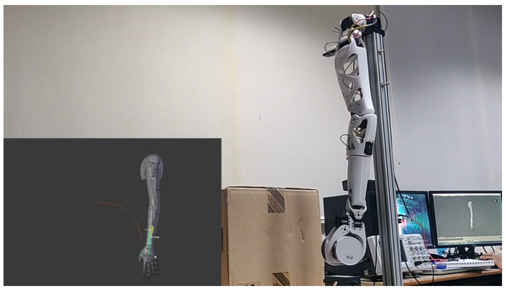
4.3 Control Strategy 3: MoCap

Using the Delicode NI-Mate software, the joint orientations and postions are communicated to the Blender environment in OSC format where the joint data is decoded and then applied onto the armature of a human-meta rig. Since we are only focusing on the left arm, the joint orientations are re-mapped to the arm rig and then the motion of the upper arm is recorded. This recorded motion is applied onto the robotic arm by computing the joint angles and the bone orientations as captured in Blender. The motion capture experiments were performed at a distance of 3 meters away from the Kinect sensor.


(a) Reaching.


(b) Grasping and changing orientation.


(c) Carrying the object.


(d) Holding the object.

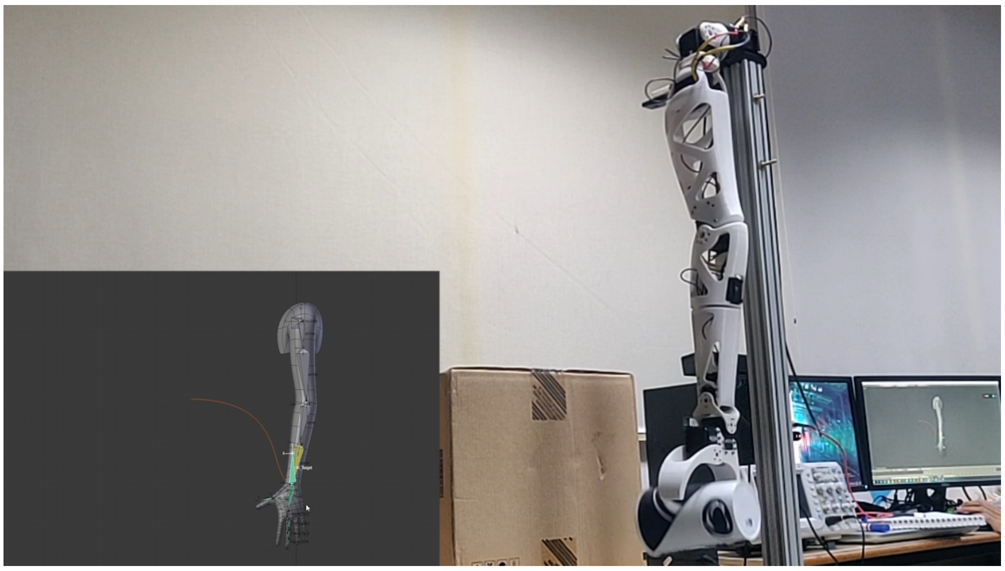
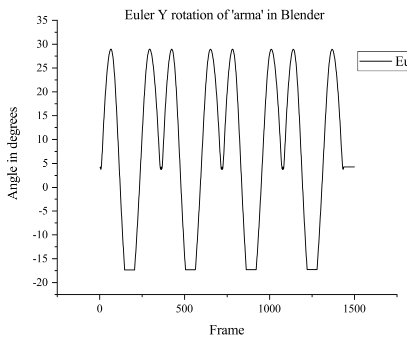
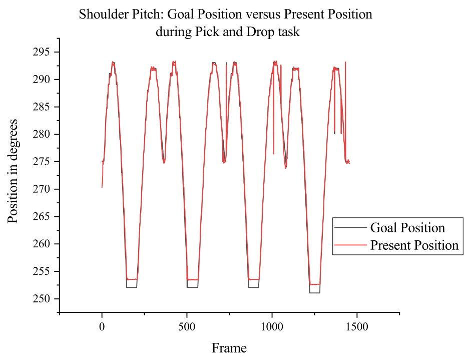

(e) Dropping the grasped object.

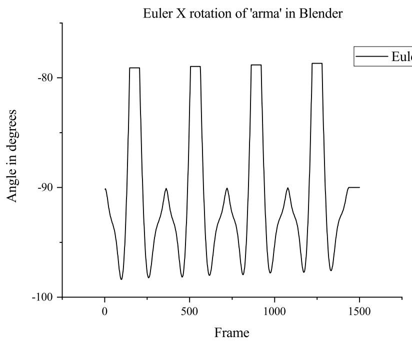
Figure FC4.13: Inverse Kinematics - Teleoperation.



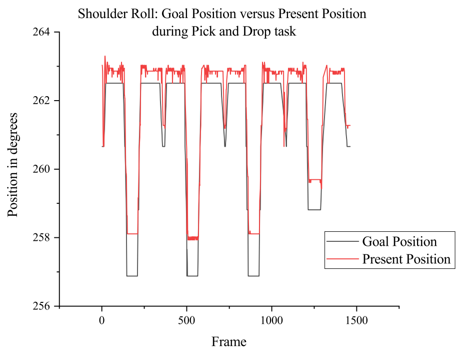
(a) Euler Y rotation of ’arma’.



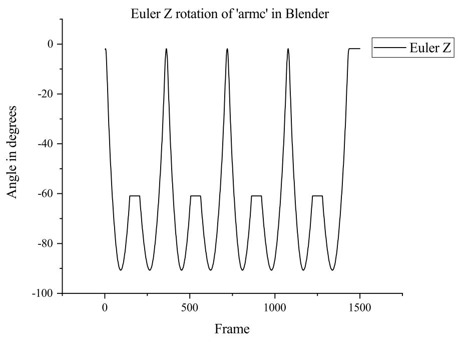
(b) Shoulder Pitch Dynamixel.



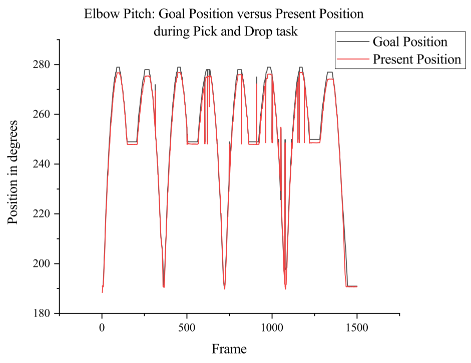
(c) Euler X rotation of ’arma’.



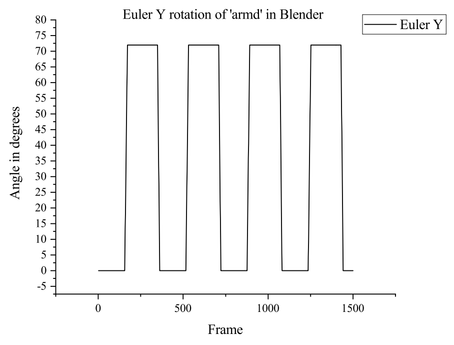
(d) Shoulder Roll Dynamixel.



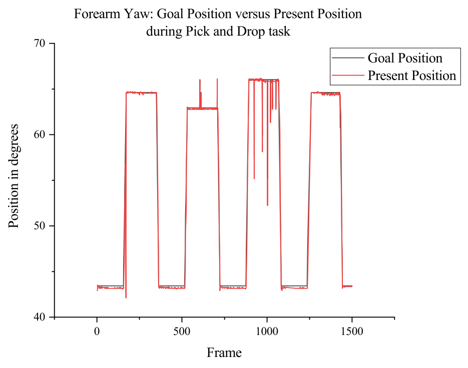
(e) Euler Y rotation of ’armc’.



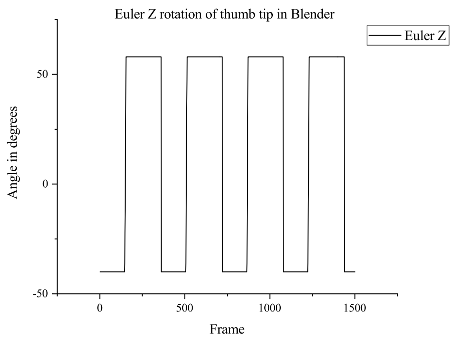
(f) Elbow Pitch Dynamixel.



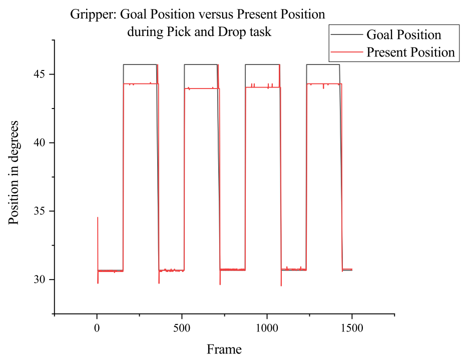
(g) Euler Y rotation of 'armd'.



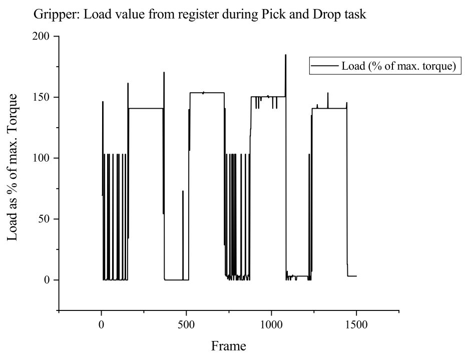
(h) Forearm Yaw Dynamixel.



(i) Euler Y rotation of 'armd'.



(j) Forearm Yaw Dynamixel.



(k) Forearm Yaw Dynamixel Load value.

Figure FC4.14: Inverse Kinematics: Path Following to grasp an object - Goal position versus Present position during motion of the arm.

4.3.1 MoCap: Wrist Joint

The Kinect can capture the wrist's flexion and extension, and the radial[1] (abduction) and ulnar[2] (adduction) deviation of the wrist although it is subjected to variation due to the ToF artifacts in human pose estimation. Also, the radial and ulnar deviation of the wrist as well as its extension and flexion as detected by the kinect are constrained to a specific range.

In order to determine the maximum deviation captured by the Kinect when performing the wrist's extension and flexion, a subject's motion was recorded by the Kinect and the graphs from the captured motion are analysed. Multiple recordings of such motions are performed to determine the variations and limitations of Kinect's motion capture. The design of the wrist joint of the robotic arm limits the wrist extension to the value of 35°. The Kinect is unable to measure forearm pronation and supination. However, with the help of NI-mate, the orientation of the wrist can be applied to the forearm yaw in order to perform pronation and supination of the wrist on the robotic arm.

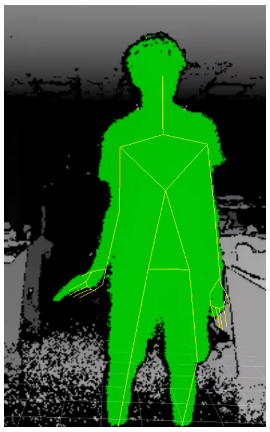
The wrist's range of motion i.e, its flexion, extension, abduction and adduction is measured after recording the data in Blender using the Kinect. The typical Range of Motion of a human wrist joint, its values determined by the Kinect and the maximum range of motion of the robotic arm are tabulated and presented in Table TC4.3.

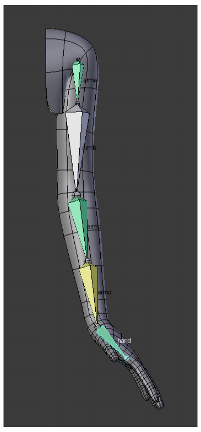
Table TC4.3: Comparison of Wrist's Range of Motion. The typical range of motion values are taken from [9].

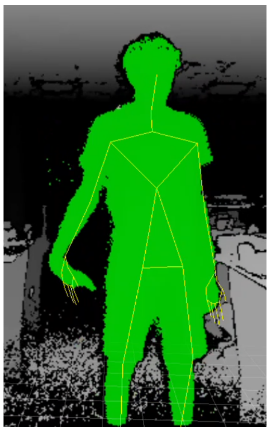
	Typical Range of Motion	Kinect V2	robotic arm
Wrist Extension	70°	45.58°	35°
Wrist Flexion	75°	44.34°	75°
Wrist Abduction	20°	36.47°	70°
Wrist Adduction	30°	30.86°	60°

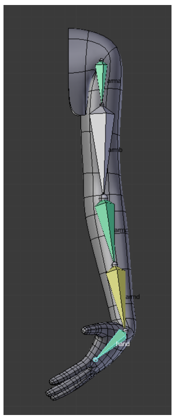
[1]Radial deviation, otherwise known as radial flexion, is the movement of bending the wrist to the thumb, or radial bone side.

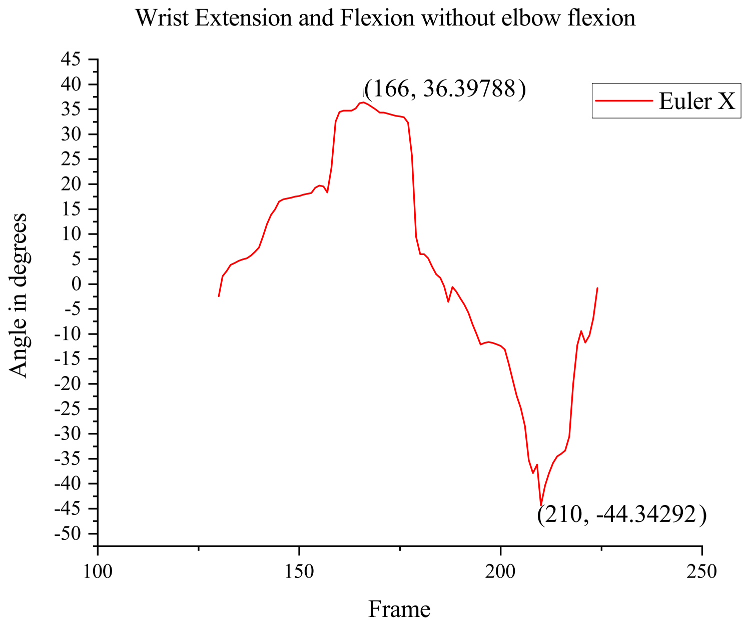
[2]Ulnar deviation is the movement of bending the wrist to the little finger, or ulnar bone side.


(a) Wrist Extension - Kinect.


(b) Wrist Extension - Blender (frame 166 in the graph below).

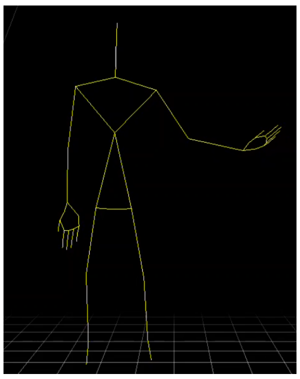

(c) Wrist Flexion - Blender at frame 166.

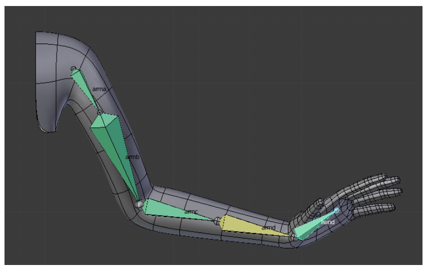

(d) Wrist Flexion - Blender (frame 210 in the graph below).

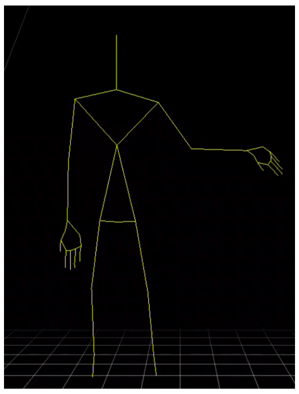


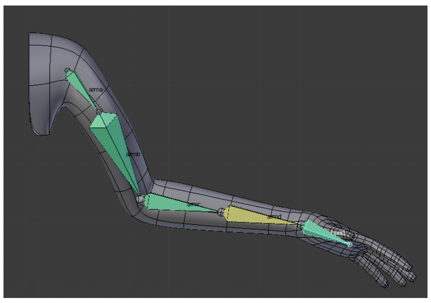
(e) Wrist.

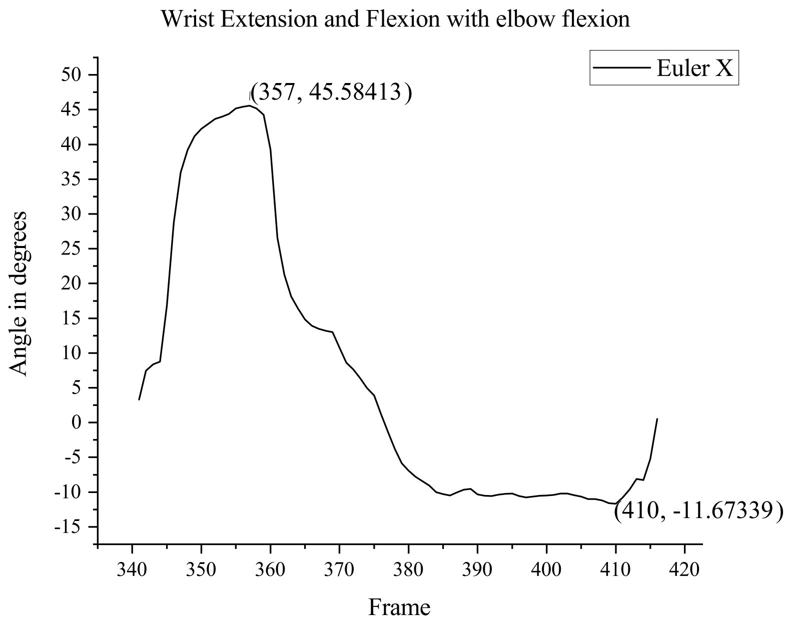
Figure FC4.15: Wrist Extension and Flexion.


(a) Wrist Extension - Kinect.


(b) Wrist Extension - Blender (frame 357 in in the graph below).

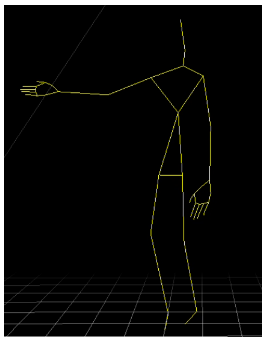

(c) Wrist Flexion - Blender at frame 166.

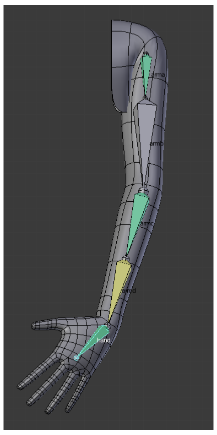

(d) Wrist Flexion - Blender (frame 410 in the graph below.

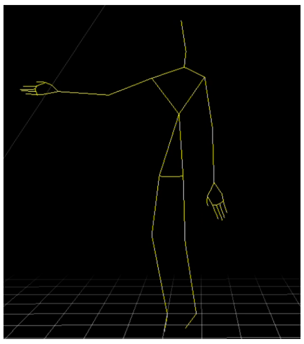


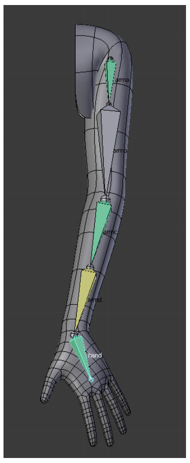
(e) Wrist.

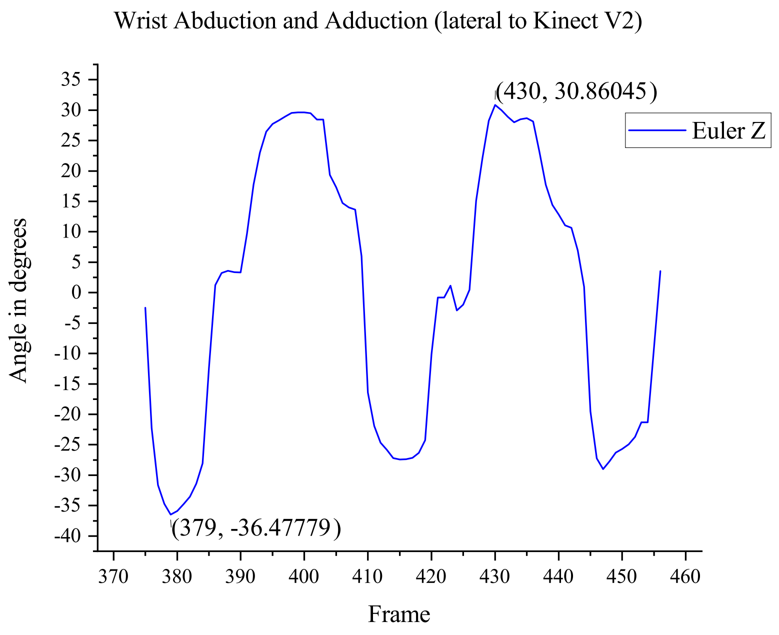
Figure FC4.16: Wrist Extension and Flexion with elbow flexion.


(a) Wrist Extension - Kinect.


(b) Wrist Extension - Blender (frame 379 in in the graph below).


(c) Wrist Flexion - Blender at frame 166.


(d) Wrist Flexion - Blender (frame 430 in the graph below.



(e) Wrist.

Figure FC4.17: Wrist Abduction and Adduction.

Moreover, Kinect captures the orientations of the joints and not the orientation of the link between the joints. Therefore, after viewing the change in orientation of the empties by viewing the motion capture data within Blender, the orientation of the appropriate empties which correspond to a particular joint are projected onto the corresponding connected link in the armature.

4.3.2 MoCap: Gripper Control on Robotic Arm

To ensure that the *armb* is facing according to the pose the person is making within the FoV of the Kinect, **add rotations** and **Basic + Orientation** is enabled within Blender and NI-Mate respectively. In addition to this, the *armb* bone is constrained to copy the rotation of the **Elbow** as seen by the Kinect through NI-Mate. This is through manual observation of the orientation of the elbow joint empty received by the Blender that differs only in the Z of the quaternion value.

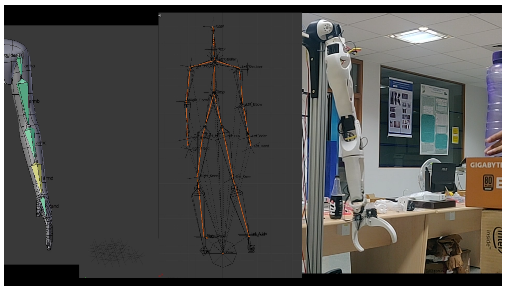
Each of the fingers has three joints namely:

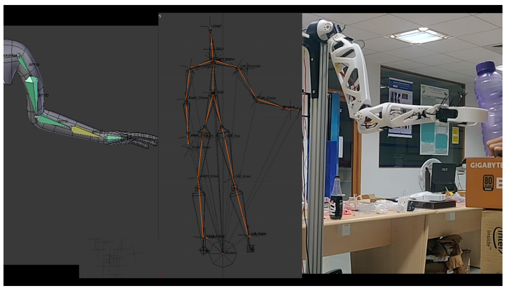
- Metacarpophalangeal joint (MCP) – the joint at the base of the finger.
- Proximal interphalangeal joint (PIP) – the joint in the middle of the finger.
- Distal interphalangeal joint (DIP) – the joint closest to the fingertip.

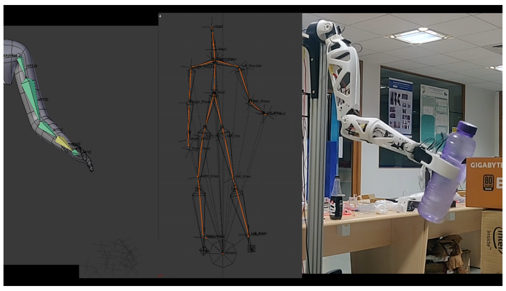
Although the Kinect is able to track these joints through the NI-Mate software, since the robotic arm does not include a hand like end-effector, the gripper is controlled through sensing whether the hand is open or closed as detected by the Kinect camera. The opening and closing of the hand is either determined by computing the distance between the thumb tip and the hand tip joint or by determining the proximity of the above mentioned joints of the finger with that of the thumb joints. In this study, we have taken the proximity between the two joints (the Left Middle finger's MCP and

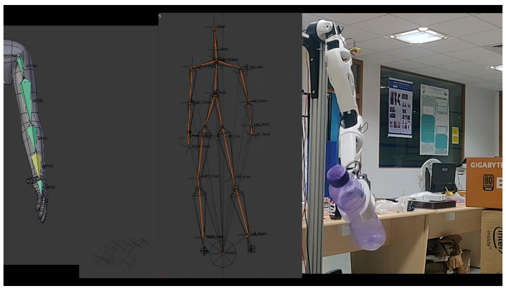
Left Middle finger's DIP joints) to determine whether the hand is open or closed. The distance factor is mapped to range of motion of the gripper motor in order to achieve the action of opening and closing of the fist on the robotic arm.

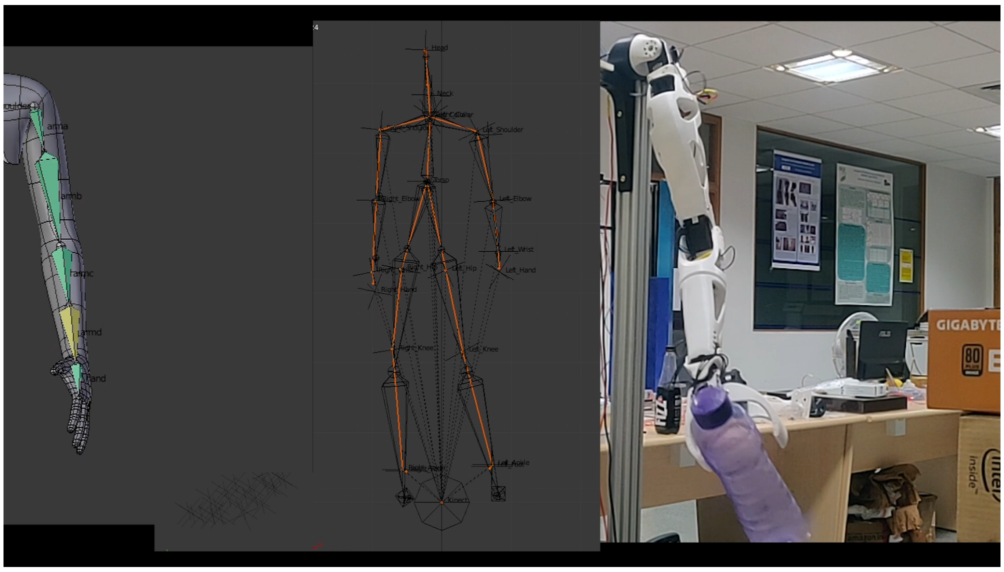
The motion sequence (see Figure FC4.18) was recorded and applied on the biomimetic arm to analyze and understand the end-effector's abilities as well as to determine if the end-effector actions can be executed using recorded motion. These actions can be replicated in order to perform simple operations such as pick and drop an object. The action is performed in a manner where the joints detected by the NI-Mate is not occluded to avoid any errors during the measurement of the position and orientation of the joint.


(a) Starting pose using MoCap.


(b) Reaching pose using MoCap.

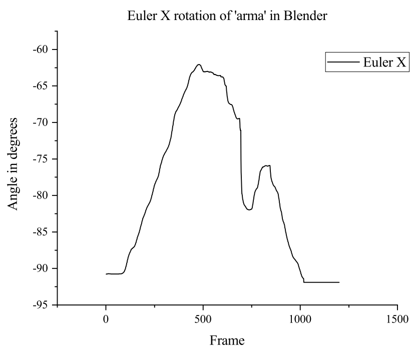

(c) Grasping detected by the MoCap.


(d) Carrying the object.

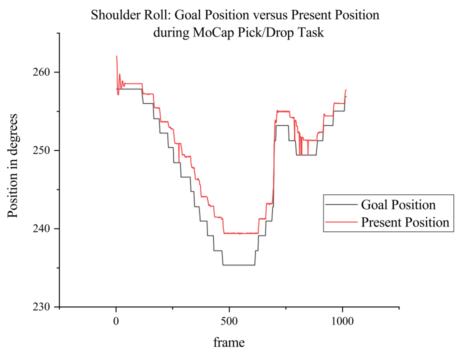


(e) Dropping the grasped object.

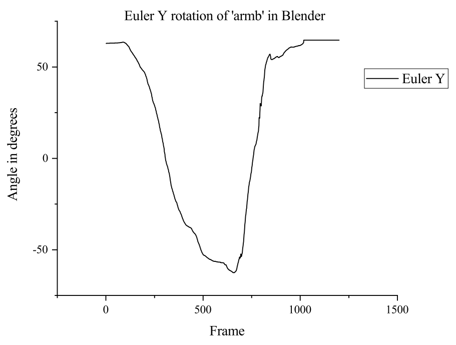
Figure FC4.18: Motion Capture - Applying a recorded motion onto the robotic arm. The object was placed to ensure that there is some load applied on the robotic arm's end effector.



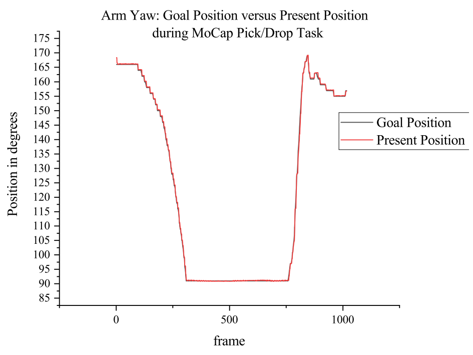
(a) Euler X rotation of 'arma'.



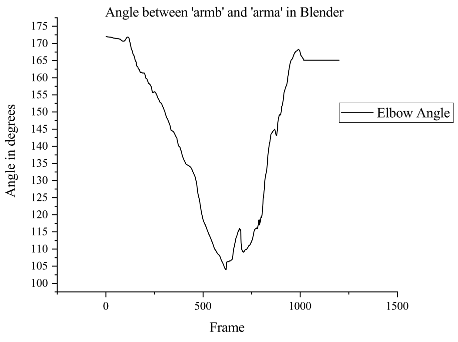
(b) Shoulder Roll Dynamixel.



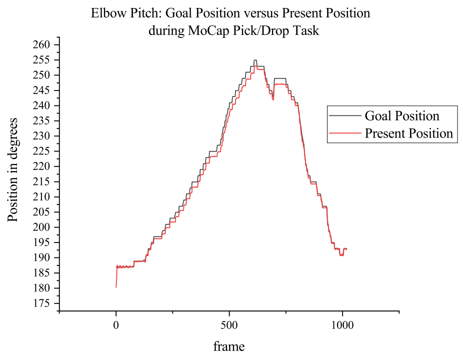
(c) Euler Y rotation of 'armb'.



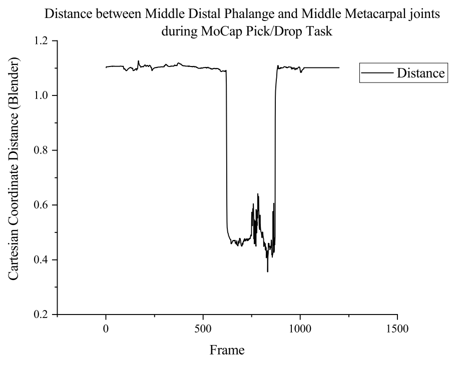
(d) Arm Yaw Dynamixel.



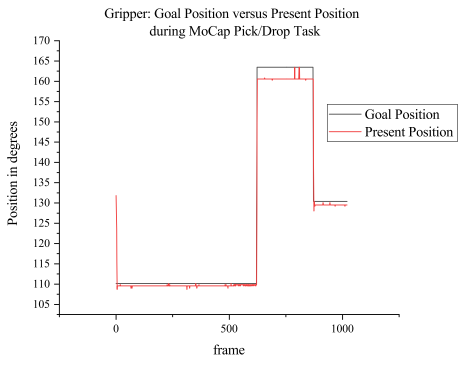
(e) Angle between *armb* and *armc* in Blender.



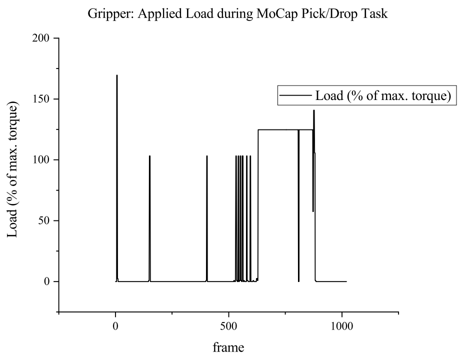
(f) Elbow Pitch Dynamixel.



(g) Middle Metacarpal and Distal joint distance.



(h) Gripper Dynamixel.



(i) Applied load on the Gripper Dynamixel.

Figure FC4.19: Motion Capture: Applying a recorded motion onto the arm for pick/drop tasks.

4.4 Default Characteristics of the System

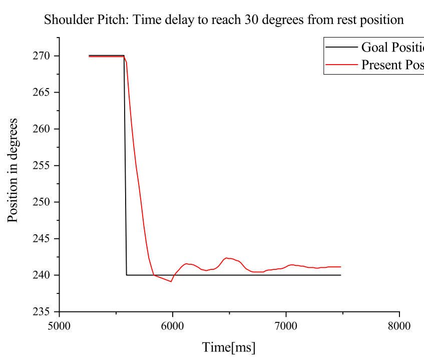
The robot upper arm is composed of a total of 8 motors, each of which are responsible for one degree of freedom and their role is tabulated in the Table TC4.4.

Table TC4.4: Dynamixel configuration for the robotic arm.

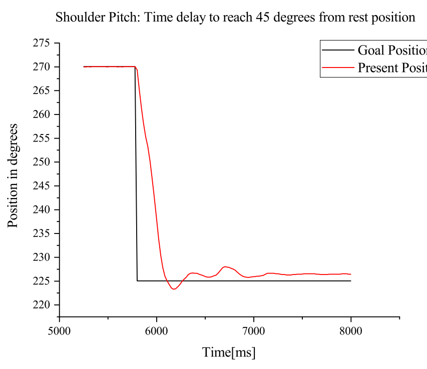
Dynamixel Model	ID of motor	DoF controlled	Range of Motion (current)
MX-106 AT	10	Shoulder Pitch	180°
MX-64 AT	11	Shoulder Roll	90°
MX-64 AT	12	Arm Yaw	180°
MX-64 AT	13	Elbow Pitch	100°
AX-18 A	14	Forearm Yaw	300°
AX-18 A	15	Wrist Pitch	124.8
AX-18 A	16	Wrist Roll	101.95°
AX-18 A	17	Gripper	74.707°

The Dynamixel MX-series actuators provide 360 degrees position control with a resolution of 0.088°and the AX-series actuators provide 300 degrees position control with a resolution of 0.29°. However, in this study the motors are actuated only after the change in the position from the Blender virtual environment is greater than 1°. The transient response for the Dynamixels present on the robotic arm to reach a certain steady state is plotted and tabulated below.

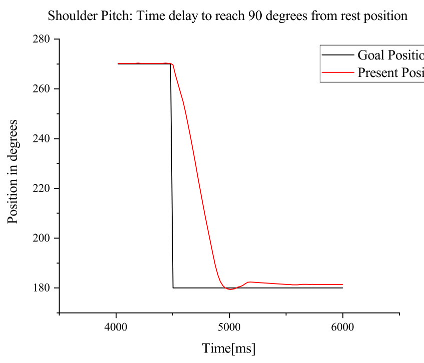
- Shoulder Pitch - (Figure FC4.20 and Table TC4.5)
- Arm Yaw - (Figure FC4.21 and Table TC4.6)
- Elbow Pitch - (Figure FC4.22 and Table TC4.7)
- Forearm Yaw - (Figure FC4.23 and Table TC4.8)
- Wrist Roll - (Figure FC4.25 and Table TC4.10)
- Wrist Pitch - (Figure FC4.24 and Table TC4.9)
- Shoulder Roll and Gripper - (Figure FC4.26 and Table TC4.11)



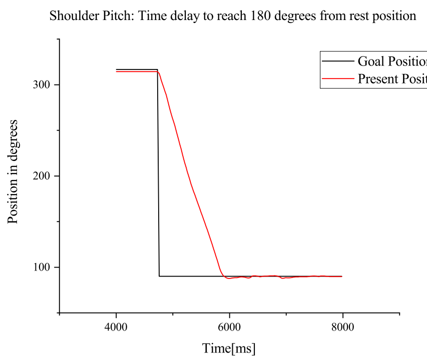
(a) Delay for 30°.



(b) Delay for 45°.



(c) Delay for 90°.

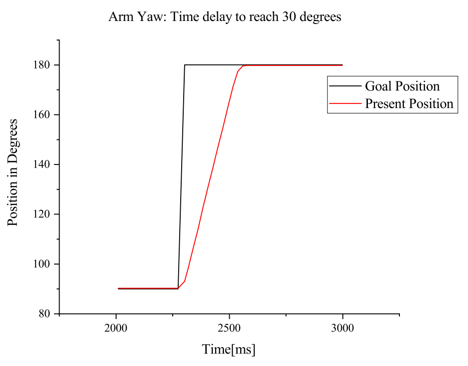


(d) Delay for 180°.

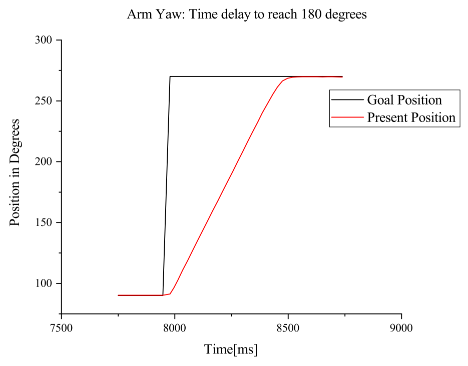
Figure FC4.20: Delay for Shoulder Pitch Dynamixel.

Table TC4.5: Time delay to reach goal position by Shoulder Pitch Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Variation(degree)
10	270°	240°	1.143	0.4396
	270°	225°	1.099	0.879
	270°	180°	1.296	1.319
	270°	90°	1.770	0.439



(a) Delay for 90°.

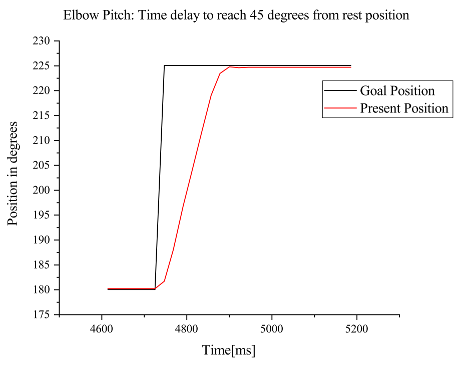


(b) Delay for 180°.

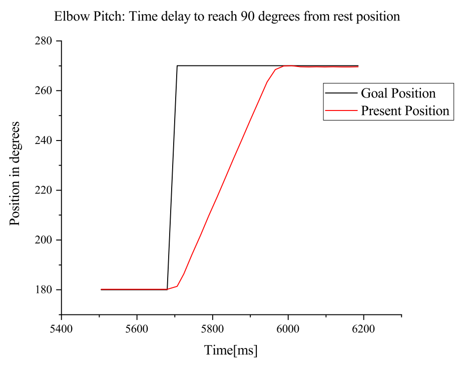
Figure FC4.21: Delay for Arm Yaw Dynamixel.

Table TC4.6: Time delay to reach goal position by Arm Yaw Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Error (degrees)
12	90°	180°	0.278	0.2637
	90°	270°	0.584	0.2637



(a) Delay for 45°.

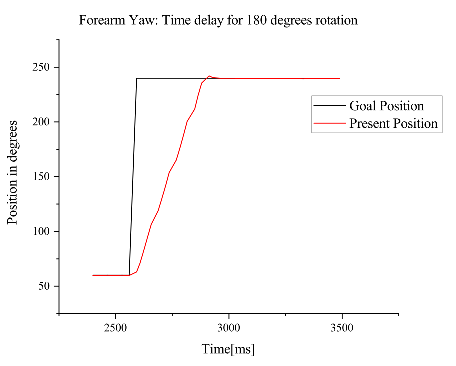


(b) Delay for 90°.

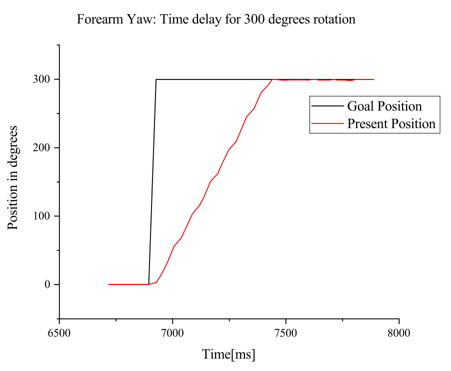
Figure FC4.22: Delay for Elbow Pitch Dynamixel.

Table TC4.7: Time delay to reach goal position by Elbow Pitch Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Error (degrees)
13	180°	225°	0.198	0.3156
	180°	270°	0.348	0.5274



(a) EDelay for 180°.

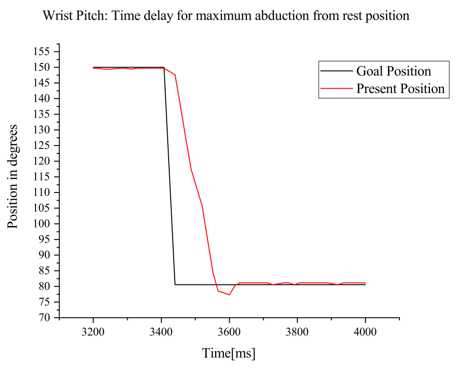


(b) Delay for 300°.

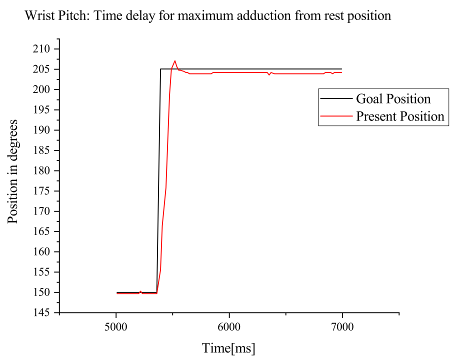
Figure FC4.23: Delay for Forearm Yaw Dynamixel.

Table TC4.8: Time delay to reach goal position by Forearm Yaw Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)
14	0°	300°	0.88
	60°	240°	0.368



(a) Delay for Maximum Abduction.

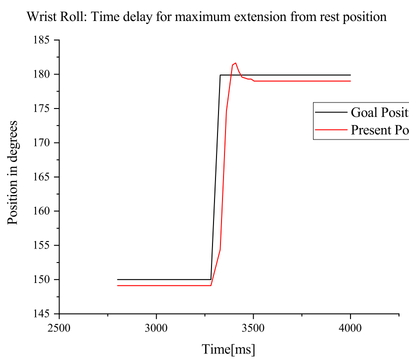


(b) Delay for Maximum Adduction.

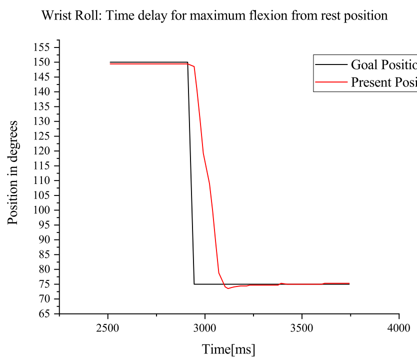
Figure FC4.24: Delay for Wrist Pitch Dynamixel.

Table TC4.9: Time delay to reach goal position by Wrist Pitch Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Error (degrees)
15	150°	80°	0.191	0.5859
	150°	205°	0.464	0.8789



(a) Delay for Maximum Extension.

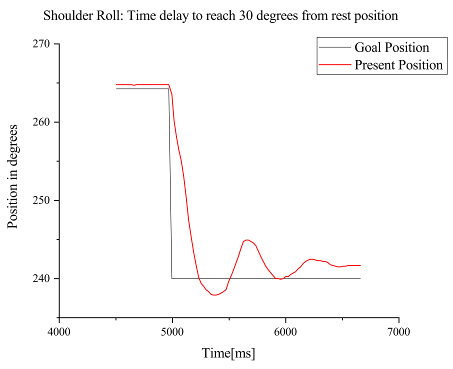


(b) Delay for Maximum Flexion.

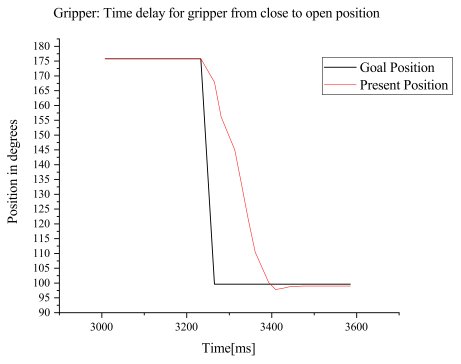
Figure FC4.25: Delay for Wrist Roll Dynamixel.

Table TC4.10: Time delay to reach goal position by Wrist Roll Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Error (degrees)
16	150°	180°	0.175	0.8789
	150°	75°	0.287	0.2929



(a) Delay for 30°.



(b) Delay for the Gripper to reach open position.

Figure FC4.26: Delay for Shoulder Roll and Gripper Dynamixel.

Table TC4.11: Time delay to reach goal position by Shoulder Roll and Gripper Dynamixel.

ID of motor	From (degrees)	To (degrees)	Delay (seconds)	Error (degrees)
11	270°	240°	1.422	1.6703
17	99.71°	175.95°	0.208	0.5859

4.5 Results and Observations

The observed results show correlation between the joint orientations performed in virtual space and the joint orientations performed in the actuator space with a variation of 2-5 degrees at the maximum. The cause for the variations are described later in this section under the limitations of the system. The respective code files for the experiments are documented in the Appendix E.

4.5.1 Forward Kinematics

In Forward Kinematic control of the robotic arm, the final pose (steady state) reached by the robotic arm differed from the actual position of the arm as viewed in Blender by a certain margin (see Table TC4.12). In other words, the position reached by a certain Dynamixel motor differs in value from the desired position value as seen in the Blender environment.

Table TC4.12: Forward Kinematics final orientation of end-effector.

ID of motor	Goal Position (Final Pose)	Present Position (Final Pose)	Error (degrees)
11	238.153	240.967	2.813
13	234.989	233.934	1.055
14	199.804	198.925	0.879
16	177.246	176.367	0.879

4.5.2 Inverse Kinematics (Same Pose as FK)

Similarly, in Inverse Kinematic control of the robotic arm, the final pose (steady state) reached by the robotic arm differed from the actual position of the arm as viewed in Blender by a certain margin (see Table TC4.13).

Table TC4.13: Inverse Kinematics final orientation of end-effector.

ID of motor	Goal Position (Final Pose)	Present Position (Final Pose)	Error (degrees)
10	274.022	274.637	0.615
11	244.747	247.12	2.373
12	180.44	180.3956	0.35
13	235.956	234.285	1.67
14	209.765	208.88	0.879
16	177.246	176.074	1.171

4.5.3 Inverse Kinematics (Path Follow: Circle)

In this particular experiment, the arm was made to go around a circular path (see Figure FC4.10) once and twice respectively within 27.5 seconds. The positions reached by the Dynamixels were recorded from the motors appeared to differ, although by a small margin, during the execution of the experiment. As the robotic arm completed following the path twice within the given interval, the values reached by the Dynamixels differed at similar points on the circular path. For instance, in Figure FC4.11f, the values reached by the Elbow Pitch Dynamixel during the first and second run are 217.1429 and 219.1648 respectively. These variations are most likely due to actuating the Dynamixels after the change in the value as seen from within Blender is greater than 1°or due to the load of the distal end of the robotic arm i.e, the forearm and the end-effector part.

4.5.4 Inverse Kinematics (Pick up/Drop Task)

In this particular experiment, the arm was made to follow a path to reach an object,change its orientation, and drop it at the starting position. The task was executed ten times. However, for analysis purposes, only a certain section of the time duration is considered. Although the positions reached by the Dynamixels were recorded from the motors appeared to differ, although by a small margin, during the execution of the experiment, the robot's end effector successfully reached the object and was able to grasp it.

It is observed that during the length of the experiment, the positions reached by the robotic arm are mostly consistent. However, in Figure FC4.14, the values reached by the Dynamixels during every run when grasping the object are tabulated below:

Table TC4.14: Variation of Dynamixel values during multiple runs. The value units are in degrees(°).

	1st		2nd		3rd		4th	
Motor ID	Goal	Present	Goal	Present	Goal	Present	Goal	Present
10	252.044	253.538	252.044	253.450	252.044	253.538	251.077	252.659
11	256.879	258.110	256.879	257.934	256.879	258.110	258.813	259.692
13	248.967	247.912	248.967	248.0	248.967	247.912	249.934	248.615
14	64.615	64.527	62.945	62.763	66.021	66.109	64.615	64.527
17	45.714	44.307	45.714	43.956	45.714	44.043	45.714	44.307

In the Table TC4.14, the motor having its ID as 17 is the Gripper motor. The reason for its variation in the position value is due to the outward thrust by the deformation caused during grasping of the object. The variation of the goal position values during multiple runs is of the order of 1°. This is likely due to actuating the motors from values computed within Blender between two consecutive frames do not meet the required required difference of 1°to actuate the motors.

4.5.5 MoCap (Pick up/Drop Task)

To increase the intuitiveness of kinematic control, motion imitation allows the kinematics of a redundant robot to be controlled simply by human motion [6,7]. In this experiment, the human motion was recorded using the Kinect and then applied onto the robotic arm to perform a Pick up and Drop task. The object was positioned to the left side of the robotic arm. The opening and closing of the gripper was achieved by computing the distance between the Left Middle Finger's Metacarpal and Distal joints as observed by the NI-Mate add-on using the Kinect camera. The distance between these two joints during opening and closing of the fist was viewed within Blender and then a threshold value was applied to determined whether to actuate the gripper to open or close respectively. As seen in Figure FC4.19g, when the distance between the two joints is below 0.5 (in Blender's Coordinate System), the gripper was made to close accordingly in Figure FC4.19h. The load applied on the Dynamixel during this task is seen in Figure FC4.19i which shows that the object has been grasped.

In the Figure FC4.19c, the rotation of the arm joint as seen in Blender was twice as that of the actual motion. The arm's yaw when performing the task was changed to nearly 90°, but the NI-Mate viewed the joint rotation to be twice that of the actual value. However, due to constraints placed on the robotic arm, the arm yaw was limited to 90°.

To ensure that the rotation value of the arm yaw as computed by NI-Mate is same, the *Copy Rotation* constraint in the bone properties panel of the Blender has an influence factor. This value cab be changed to a factor of 0.5 to ensure that values incoming from the NI-Mate get multiplied by this factor before applied onto the armature bones.

However, this method cannot precisely control a robot's end-effector on a given trajectory for two reasons. One reason is the mechanical discrepancy between the human arm and the robotic arm, and the other reason is the inaccuracy of the recorded human motion.

4.6 Study Limitations and Improvements

The findings of this study have to be seen in light of some limitations namely:

- The primary limitation to the generalization of these results is the fact that the Dynamixels are actuated only after there is a change of 1°in the orientation of any bone as viewed in Blender. Though the Dynamixel MX and AX series provide a resolution of 0.088°and 0.29°respectively, the motion of the robotic arm is drastically changed. The primary reason for choosing the resolution of 1°is that repetitive calls to set the position of the Dynamixel using the Arbotix-M microcontroller causes the motor to shutdown by throwing an Overload error and the fact that the Arbotix-M has a lower clock frequency than compared to that of the Dynamixels.

- The current baud rate is set to 115200 which can also be increased as the Dynamixel MX series and AX series have a 4.5 Mbps and 1 Mbps maximum baud rate specification.

- Measurements using Kinect involve the placement of Kinect is a limiting factor. The Kinect is designed to track the front side of the user and motion tracking suffers from occlusions (e.g. self-occlusion by other body parts and due to surrounding objects being detected as body parts.)

- Real-time Motion Capture is Limited to the control of two Dynamixels i.e., the shoulder pitch and the shoulder roll. During real-time tracking, the voltage across the data line along which the values are sent to the Dynamixels is at 5V. However, as more Dynamixels are daisy-chained, the voltage across the data pins on the Arbotix-M board diminishes to a value lesser than 3V. As the serial communication at a TTL level always remains between the limits of 0V and Vcc, which is often 5V or 3.3V, any value lower than this would result in no actuation of the motors.

- Stability of the support off of which the arm is hanging may induce errors during operation as the Dynamixels, despite their compact size, can produce high torques which may result in a reaction force causing a change in course of the arm trajectory.

The system's performance can be increased to achieve better accuracy and precision by:

- The margin of error in reaching the goal position by the Dynamixel modules can be fine tuned by either setting the complaince margins or by applying Proportional, Integral, Derivative (PID) gains to the motor.
- The resolution of the actuation can be improved by mapping the values from 0°-360°to the range of the Dynamixels (0-4095 for MX and 0-1024 for AX) and setting an appropriate value as the required threshold.
- A stable support can improve the results of the experiments performed.

Further improvements to the system include:

- Applying PID gains values to the Dynamixel servos to reduce the error margin of the system.
- The Arbotix-M controller incorporates an XBee socket which can be used to add wireless communications to the board by plugging in an XBee module. This will create a wireless serial connection, allowing the Arbotix-M present on the slave device to another micro-controller or a computer through a second XBee module.
- The system can be interfaced with a camera to provide visual feedback for teleoperation purposes as well as to accomplish tasks such as object detection and manipulation using computer vision techniques.
- The integration of a third party end-effector can make the system suitable for manipulation tasks on top of reaching and grasping actions.
- With the introduction of a better micro-controller or a microprocessor, real-time motion capture with the Kinect may be achieved.
- Long-term testing with the higher payloads and potential impacts with the environment can help in determining the durability of the system.

CHAPTER 5

CONCLUSIONS

Telerobotics is one of the essential research topics, which is widely applied into assistive and medical rehabilitation, and even in the manufacturing processes in the industry. This study implements a telerobotic system with 8-DoF that mimic the human arm motion using a Game Engine as an interface for simulation and emulation purposes. We believe that the current robot platform and interface is highly suitable for telerobotic purposes, and can be of use to researchers at the intersection of cognitive science and human motor control with robotics.

In this study, two control strategies to drive an 8-DoF robot arm are presented. With the use of motion capture technology such as the Kinect, the robot arm was able to reproduce the actions of a human operator to perform simple tasks such as reaching and grasping objects. A visual representation of the human arm is modeled in Blender Game Engine which formed the basis of understanding the kinematics for the human upper limb and use of Blender as a visualization, simulation and emulation tool to gain qualitative information about the robot's behavior and effectiveness. Through inverse kinematics using Blender Game Engine, new human-like motions can be modeled to perform various telerobotic tasks. This study demonstrated the abilities of Blender as an interface to control and program the robot arm, as well as providing support for motion capture using NI-Mate to capture human motions in real-time. The characteristics of the robot arm are highlighted and the mapping of the physical arm design to that of the virtual arm model is discussed. In addition to this, different control strategies were applied onto the robot arm and compared to quantify its effectiveness, accuracy, and precision in the execution of reaching and grasping tasks applicable to telerobotic scenarios. Finally, the results and limitations of the current system are discussed. In the future, the entire system can be upgraded to achieve a some degree of autonomy with the integration of a camera to enable human-robot interaction with the environment.

Bibliography

[1] Sébastien Mick, Mattieu Lapeyre, Pierre Rouanet, Christophe Halgand, Jenny Benois-Pineau, Florent Paclet, Daniel Cattaert, Pierre-Yves Oudeyer, and Aymar de Rugy. Reachy, a 3d-printed human-like robotic arm as a testbed for human-robot control strategies. *Frontiers in Neurorobotics*, 13:65, 2019.

[2] Martin F. Stoelen, Fabio Bonsignorio, and Angelo Cangelosi. Co-exploring actuator antagonism and bio-inspired control in a printable robot arm. In Elio Tuci, Alexandros Giagkos, Myra Wilson, and John Hallam, editors, *From Animals to Animats 14*, pages 244–255, Cham, 2016. Springer International Publishing.

[3] J. Shotton, A. Fitzgibbon, M. Cook, T. Sharp, M. Finocchio, R. Moore, A. Kipman, and A. Blake. Real-time human pose recognition in parts from single depth images. In *CVPR 2011*, pages 1297–1304, June 2011.

[4] Vangos Pterneas. Finger tracking using kinect v2. January 2016.

[5] Diana Pagliari and Livio Pinto. Calibration of kinect for xbox one and comparison between the two generations of microsoft sensors. *Sensors*, 15:27569–27589, 10 2015.

[6] Musculoskeletal key - structure and function of the elbow and forearm-complex. retreived from `https://musculoskeletalkey.com/structure-and-function-of-the-elbow-and-forearm-complex/`. 2016, December 5.

[7] Wanbin Song, Le Anh Vu, Seokmin Yun, Seung-Won Jung, and Chee Won. Hole filling for kinect v2 depth images. 12 2014.

[8] Ziyun Cai, Jungong Han, Li Liu, and Ling Shao. Rgb-d datasets using microsoft kinect or similar sensors: a survey. *Multimedia Tools and Applications*, 76, 03 2016.

[9] Physiopedia. Wrist and hand — physiopedia,, 2020. [Online; accessed 24-September-2020].

[10] National Research Council, Nathaniel I. Durlach, and Anne S. Mavor. *Virtual Reality: Scientific and Technological Challenges*. National Academy Press, USA, 1994.

[11] Haiying Hu, Jiawei Li, Zongwu Xie, Bin Wang, Hong Liu, and G. Hirzinger. A robot arm/hand teleoperation system with telepresence and shared control. volume 2, pages 1312 – 1317, 08 2005.

[12] Anthony Maciejewski and Charles Klein. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The International Journal of Robotics Research*, 4, 09 1985.

[13] Hsien-I Lin, Yu-Cheng Liu, and Yu-Hsiang Lin. Intuitive kinematic control of a robot arm via human motion. *Procedia Engineering*, 79, 12 2014.

[14] H. Seraji. Configuration control of redundant manipulators: theory and implementation. *IEEE Transactions on Robotics and Automation*, 5(4):472–490, 1989.

[15] Zhe Xu and E. Todorov. Design of a highly biomimetic anthropomorphic robotic hand towards artificial limb regeneration. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3485–3492, 2016.

[16] V. Patidar and R. Tiwari. Survey of robotic arm and parameters. In *2016 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–6, 2016.

[17] C. Bartneck, M. Soucy, K. Fleuret, and E. B. Sandoval. The robot engine — making the unity 3d game engine work for hri. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 431–437, 2015.

[18] T.B. Sheridan. Teleoperation, telerobotics and telepresence: A progress report. *Control Engineering Practice*, 3(2):205 – 214, 1995.

[19] M. A. Diftler, J. S. Mehling, M. E. Abdallah, N. A. Radford, L. B. Bridgwater, A. M. Sanders, R. S. Askew, D. M. Linn, J. D. Yamokoski, F. A. Permenter, B. K. Hargrave, R. Platt, R. T. Savely, and R. O. Ambrose. Robonaut 2 - the first humanoid robot in space. In *2011 IEEE International Conference on Robotics and Automation*, pages 2178–2183, 2011.

[20] Juxi Leitner, M. Luciw, Alexander Förster, and J. Schmidhuber. Teleoperation of a 7 dof humanoid robot arm using human arm accelerations and emg signals. 06 2014.

[21] T. Wei, B. Lee, Y. Qiao, A. Kitsikidis, K. Dimitropoulos, and N. Grammalidis. Experimental study of skeleton tracking abilities from microsoft kinect non-frontal views. In *2015 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, pages 1–4, 2015.

[22] Douglas Robertson, Graham Caldwell, Joseph Hamill, Gary Kamen, and Saunders Whittlesey. *Research Methods in Biomechanics: Second edition (eBook)*. 11 2013.

[23] C. Zerpa, Chelsey Lees, P. Patel, and Eryk Pryzsucha. The use of microsoft kinect for human movement analysis. *International Journal of Sports Science*, 5:120–127, 2015.

[24] Lars Mündermann, Stefano Corazza, and Thomas Andriacchi. The evolution of methods for the capture of human movement leading to markerless motion capture for biomechanical applications. *Journal of neuroengineering and rehabilitation*, 3:6, 02 2006.

[25] Lacramioara Dranca, Lopez Mendarozketa, Alfredo Goni, Arantza Illarramendi, Irene Navalpotro, Manuel Delgado-Alvarado, and María Rodríguez-Oroz. Using kinect to classify parkinson's disease stages related to severity of gait impairment. *BMC Bioinformatics*, 19:471, 12 2018.

[26] M. Gabel, R. Gilad-Bachrach, E. Renshaw, and A. Schuster. Full body gait analysis with kinect. In *2012 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 1964–1967, Aug 2012.

[27] Aaron Staranowicz, Garrett Brown, and Gian-Luca Mariottini. Evaluating the accuracy of a mobile kinect-based gait-monitoring system for fall prediction. *ACM International Conference Proceeding Series*, 05 2013.

[28] Chien-Yen Chang, B. Lange, Mi Zhang, S. Koenig, P. Requejo, N. Somboon, A. A. Sawchuk, and A. A. Rizzo. Towards pervasive physical rehabilitation using microsoft kinect. In *2012 6th International Conference on Pervasive Computing Technologies for Healthcare (PervasiveHealth) and Workshops*, pages 159–162, May 2012.

[29] D. Leightley, M. H. Yap, J. Coulson, Y. Barnouin, and J. S. McPhee. Benchmarking human motion analysis using kinect one: An open source dataset. In *2015 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA)*, pages 1–7, Dec 2015.

[30] Ruizhe Wang, Gérard Medioni, Carolee Winstein, and Cesar Blanco. Home monitoring musculo-skeletal disorders with a single 3d sensor. pages 521–528, 06 2013.

[31] H. Alabbasi, A. Gradinaru, F. Moldoveanu, and A. Moldoveanu. Human motion tracking evaluation using kinect v2 sensor. In *2015 E-Health and Bioengineering Conference (EHB)*, pages 1–4, Nov 2015.

[32] W. Cao, J. Zhong, G. Cao, and Z. He. Physiological function assessment based on kinect v2. *IEEE Access*, 7:105638–105651, 2019.

[33] Ahmed Ali, Fathy Elmisery, Ramadan Mostafa, and Mohammed Hussein. Motion control of robot by using kinect sensor. *Research Journal of Applied Sciences, Engineering and Technology*, 8:1384–1388, 09 2014.

[34] Jinxiao Gao, Yinan Chen, and Fuhao Li. Kinect-based motion recognition tracking robotic arm platform. *Intelligent Control and Automation*, 10:79–89, 01 2019.

[35] J. Rosado, F. Silva, V. Santos, and Z. Lu. Reproduction of human arm movements using kinect-based motion capture data. In *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 885–890, Dec 2013.

[36] Brook Galna, Gillian Barry, Daniel Jackson, Dadirayi Mhiripiri, Patrick Olivier, and Lynn Rochester. Accuracy of the microsoft kinect sensor for measuring movement in people with parkinson's disease. *Gait and Posture*, 39, 04 2014.

[37] Hamed Sarbolandi, Damien Lefloch, and Andreas Kolb. Kinect range sensing: Structured-light versus time-of-flight kinect. *Computer Vision and Image Understanding*, 05 2015.

[38] Alireza Bilesan, Mohammadhasan Owlia, Saeed Behzadipour, Shuhei Ogawa, Teppei Tsujita, Shunsuke Komizunai, and Atsushi Konno. Marker-based motion tracking using microsoft kinect. *IFAC-PapersOnLine*, 51(22):399 – 404, 2018. 12th IFAC Symposium on Robot Control SYROCO 2018.

[39] Silvio Giancola, Andrea Corti, Franco Molteni, and Remo Sala. Motion capture: An evaluation of kinect v2 body tracking for upper limb motion analysis. pages 302–309, 06 2017.

[40] Anna Lekova, D. Ryan, and Reggie Davidrajuh. Fingers and gesture recognition with kinect v2 sensor. *Information Technologies and Control*, 14, 09 2016.

[41] Xiong Lu, Beibei Qi, Huang Qian, Yongqiang Gao, Junbin Sun, and Jia Liu. Kinect-based human finger tracking method for natural haptic rendering. *Entertainment Computing*, 33:100335, 2020.

[42] A. Bilesan, S. Behzadipour, T. Tsujita, S. Komizunai, and A. Konno. Markerless human motion tracking using microsoft kinect sdk and inverse kinematics. In *2019 12th Asian Control Conference (ASCC)*, pages 504–509, June 2019.

[43] Z. Jamali and S. Behzadipour. Quantitative evaluation of parameters affecting the accuracy of microsoft kinect in gait analysis. In *2016 23rd Iranian Conference on Biomedical Engineering and 2016 1st International Iranian Conference on Biomedical Engineering (ICBME)*, pages 306–311, Nov 2016.

[44] Alexandra Pfister, Alexandre M. West, Shaw Bronner, and Jack Adam Noah. Comparative abilities of microsoft kinect and vicon 3d motion capture for gait analysis. *Journal of Medical Engineering and Technology*, 38(5):274–280, 2014.

[45] M. R. Kharazi, A. H. Memari, A. Shahrokhi, H. Nabavi, S. Khorami, A. H. Rasooli, H. R. Barnamei, A. R. Jamshidian, and M. M. Mirbagheri. Validity of microsoft kinect for measuring gait parameters. In *2015 22nd Iranian Conference on Biomedical Engineering (ICBME)*, pages 375–379, 2015.

[46] Moataz Eltoukhy, Christopher Kuenze, Jeonghoon Oh, Savannah Wooten, and Joseph Signorile. Kinect-based assessment of lower limb kinematics and dynamic postural control during the star excursion balance test. *Gait and Posture*, 58:421–427, 09 2017.

[47] Moataz Eltoukhy, Jeonghoon Oh, Christopher Kuenze, and Joseph Signorile. Improved kinect-based spatiotemporal and kinematic treadmill gait assessment. *Gait and Posture*, 51, 01 2017.

[48] Kourosh Khoshelham and Sander Oude Elberink. Accuracy and resolution of kinect depth data for indoor mapping applications. *Sensors*, 12(2):1437–1454, 2012.

[49] Andreas Aristidou and Joan Lasenby. Inverse kinematics: a review of existing techniques and introduction of a new fast iterative solver. 09 2009.

[50] S. C. Abdullah, M. A. M. Jusoh, N. M. Nawi, and M. D. Amari. Robot arm simulation using 3d software application with 3d modeling, programming and simulation support. In *2016 International Symposium on Micro-NanoMechatronics and Human Science (MHS)*, pages 1–3, 2016.

[51] James Meacham. Blender for robotics. October 2012.

[52] Camilo Cortes, Ana de los Reyes-Guzmán, Davide Scorza, Álvaro Bertelsen, Eduardo Carrasco, Ángel Gil-Agudo, Oscar Ruiz, and Julián Esnal. Inverse kinematics for upper limb compound movement estimation in exoskeleton-assisted rehabilitation. *BioMed Research International*, 2016:1–, 05 2016.

[53] Dwight Goins. Kinect v2 - microsoft forum, retrieved from `https://social.msdn.microsoft.com/Forums/en-US/c95d3e40-6ed6-47a1-a206-5ff26c889c29/kinect-v2-maximum-range?forum=kinectv2sdk`. September 2014.

[54] Ruuskanen Antti. Inverse kinematics in game character animation. pages 6–7, 2018.

[55] Chad Jenkins. Inverse kinematics: one-to-many mapping of workspace endeffector pose to robot configuration. 2018.

[56] Oscar Liang. Inverse kinematics basics. August 2015.

[57] Koen Buys, Tinne De Laet, Ruben Smits, and Herman Bruyninckx. Blender for robotics: Integration into the leuven paradigm for robot task specification and human motion estimation. pages 15–25, November 2010.

APPENDIX A

INSTALLING DRIVERS AND SOFTWARES FORMOTION CAPTURE USING KINECT

A.1 Getting started with Microsoft Kinect for Windows 2 (V2)

A.1.1 Setting up MoCap using Kinect V2

This section aims at implementing a motion capture system using Microsoft Kinect, developing the interface using Blender to capture motion as well as analysing measurements from captured or recorded motion data. The section is divided into several stages which include experimental setup, data acquisition and analysis of the skeletal tracking data obtained from the Kinect.

To setup a motion capture system, refer to FA1.1 to connect the Kinect V2 to your system.

Although the NI Mate installation also installs the required drivers for the Kinect V2, it is important to know what these drivers are capable to make the sensor technology perform.

Note: On first plugin after installation of required drivers, the firmware on the device will be updated. This may result in the device enumeration happening several times in

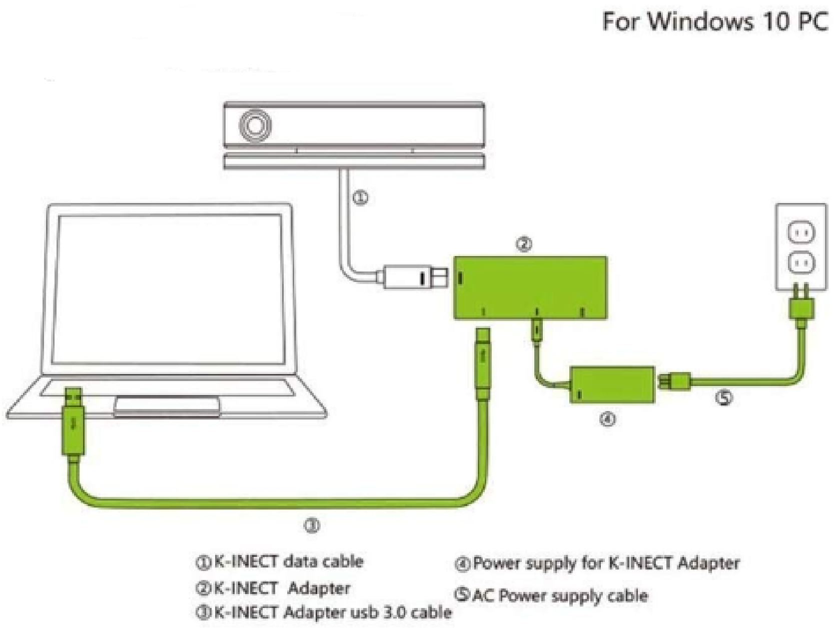


Figure FA1.1: Kinect to Windows 10 PC

the first minute.

A.1.2 Kinect for Windows Runtime 2.2.1811

To install the Kinect for Windows Runtime:

1. Make sure the Kinect sensor is not plugged into any of the USB ports on the computer.
2. Download and extract the contents of KinectRuntime-v2.2_1811.zip to a location on your PC.
3. Right click kinectsensor.inf and click Install. Do not run KinectRuntime-x64.msi directly.
4. Once the Kinect for Windows Runtime driver has completed installing successfully, ensure the Kinect sensor is connected to the Kinect for Windows power hub

and the power hub is plugged into an outlet. Plug the USB cable from the power hub into a USB 3.0 port on your computer. The drivers will load automatically.

A.1.3 Kinect for Windows SDK 2.0

To install the Kinect for Windows SDK 2.0:

1. Make sure the Kinect sensor is not plugged into any of the USB ports on the computer.
2. From the download location, double-click on KinectSDK-v2.0_1409-Setup.exe
3. Once the Kinect for Windows Runtime driver has completed installing successfully, ensure the Kinect sensor is connected to the Kinect for Windows power hub and the power hub is plugged into an outlet. Plug the USB cable from the power hub into a USB 3.0 port on your computer. The drivers will load automatically. Once the Kinect for Windows SDK has completed installing successfully, you can verify that installation has completed by launching Device Manager and verifying that ”KinectSensor Device” exists in the device list.

A.1.4 Kinect V2 Configuration Verifier

Kinect Configuration Verifier is a tool designed to verify your PC if it can run the Kinect V2 sensor. If any part of your system is determined to be incompatible with the sensor, it will be flagged. The program will display system information about CPU, OS, RAM, USB Controller or GPU.

A.1.5 Bug Fixes

First, visit ”Microphone privacy settings” and turn on the switch for ”Allow desktop apps to access your microphone”. In Device Manager, you should have ”WDF Kinect-Sensor Interface 0” with driver 2.2.1811.10000 (provided automatically). You should also see the Kinect as an enabled audio recording device, both in Device Manager, and under the advanced ”Sound” settings menu. If it’s disabled, in either place, the kinect reconnects every 8-10 seconds and gets disconnected.

Also, Kinect for Windows V2 periodically disconnects from the Windows 10 version 1809 system. Though the RGB and depth channel may provide data in the Kinect Studio, the video stream stops running after 5-6 seconds and remains static for 1-2 seconds. This is because no new frame was received and the connection from the Kinect was actually lost as the **Kinect Service.exe** process keeps restarting when viewed form the Task Manager.

The Kinect service is probably designed to test the microphone, and so, if it’s not able to access the microphone, the Kinect repeatedly disconnects and reconnects until it has sound input. Moreover, in Device Manager, the *Xbox NUI Sensor* constantly disappears and reappears when the interruptions occur. The Microsoft Forum suggested either to disable Kinect audio or to enable the audio device. However, disabling and enabling the microphone from the system, and not the kinect, fixes the problem.

A.2 Delicode NI Mate 2.14

Open Delicode NI Mate. On the left, all currently connected sensors are listed. The status of each sensor as well as some controllers are also listed. If you click on > beside the sensor, you can choose to transmit the kind of data you want to transmit over OSC: skeletal tracking; controller tracking; face analysis; face shapes; and triggers.

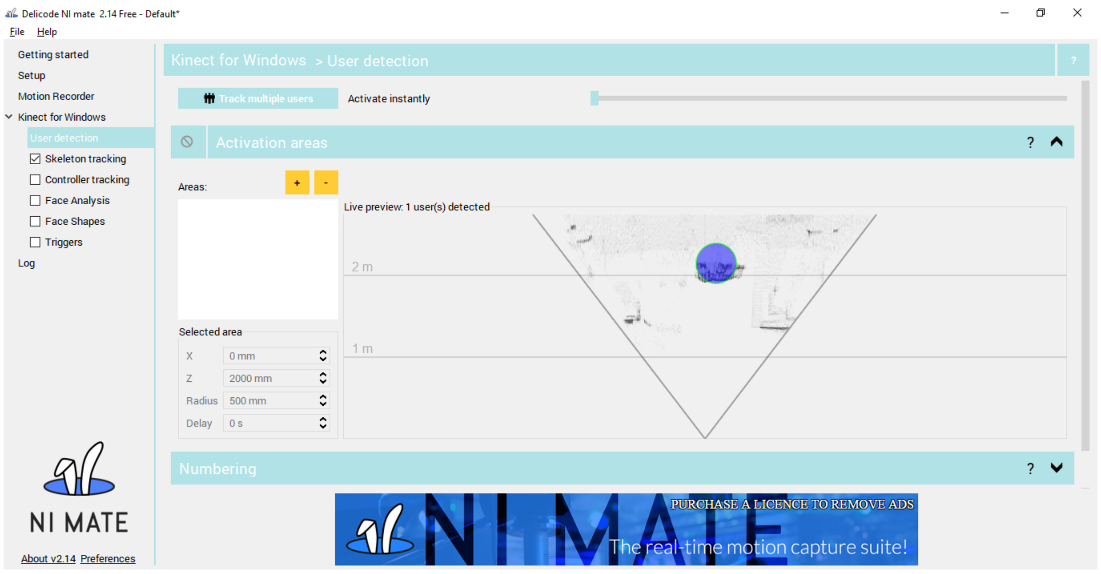


Figure FA1.2: Detecting a User in NI mate

Click on *Skeleton Tracking* → *Coordinates* and choose from the options given. It is easier to choose the Sensor as origin. Click on the checkbox for Skeleton Tracking from the menu for transporting the coordinates of the joints from kinect to blender for animation purposes as shown in FA1.3. This component is used for finding a human skeleton out of the live feed and computing its position and orientation.

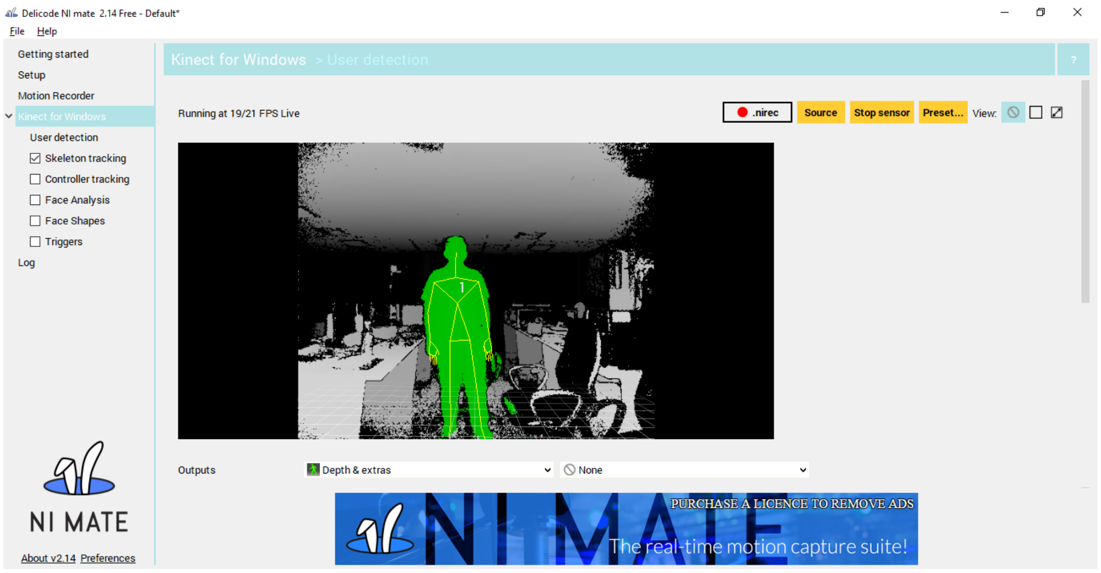
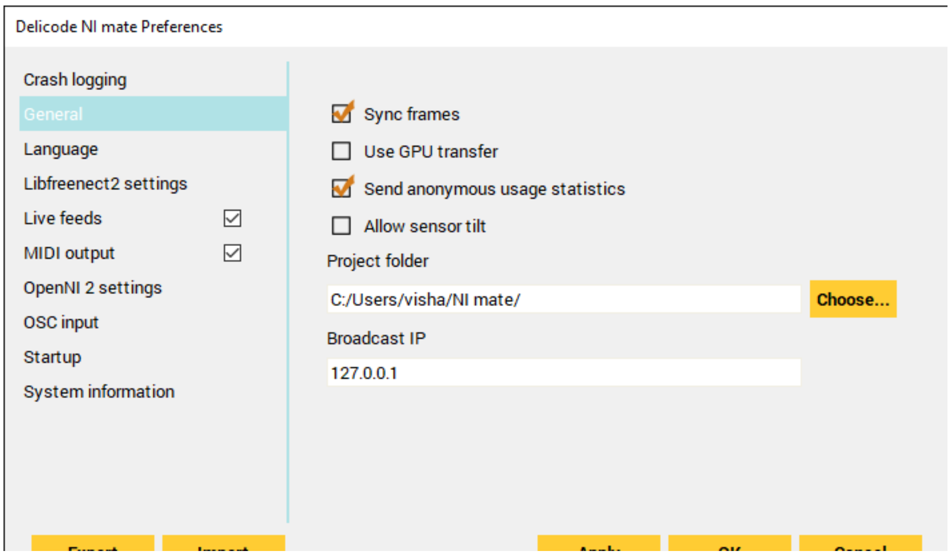


Figure FA1.3: Skeleton Tracking in NI mate to restart your device or sensor, use the *Start Sensor* or *Stop Sensor* at the top right.

NI mate uses a feature called GPU texture transfer which speeds up the process of receiving the live feed from a sensor by using the graphics card for transferring the data. Although this promotes significant speedup, unfortunately it does not work on all systems. If any crashes occur in NI mate, this setting should be the first one to disable in *Preferences* → *Use GPU Texture Transfer*. Refer to FA1.4 to enable or disable the GPU texture Transfer option.


Delicode NI mate Preferences

- Crash logging
- General
- Language
- Libfreenect2 settings
- Live feeds ☑
- MIDI output ☑
- OpenNI 2 settings
- OSC input
- Startup
- System information

- ☑ Sync frames
- ☐ Use GPU transfer
- ☑ Send anonymous usage statistics
- ☐ Allow sensor tilt

Project folder: C:/Users/visha/NI mate/

Broadcast IP: 127.0.0.1

Figure FA1.4: Disable GPU Texture Transfer

Once all the above mentioned steps are completed, the system is ready to track human motion, i.e, all the 25 joints of the human body are ready to be tracked by the NI mate software. The IP address and port specify where all out-going OSC traffic is sent to. In most cases, leaving 127.0.0.1 (localhost, the computer currently being used) with the default port of 7000 are the desired settings.

The data being transmitted follows the OSC format by specifying an address and its content. The address refers to the value the OSC message should control in the receiving software. In NI mate, a good example is the name of some skeleton joint. The content is a list of values that can be strings or numbers. For instance, an OSC message look as follows:

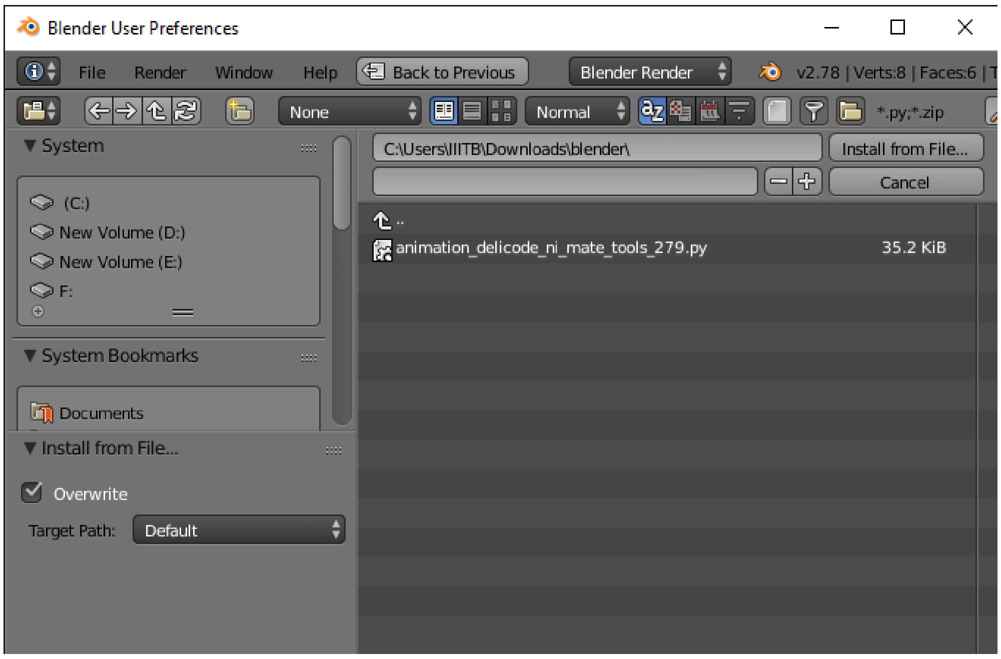
```
/skeleton_left_hand 5.2 2.1 −20.7 0.97 0.2 0.5810 0.062
```

The message contains the address, beginning with the protocol compliant forward slash symbol, with the content being 3 position floats and 4 orientation floats.

This concludes the NI mate section.

A.3 Blender 2.78

In order to interface Blender with NI mate, the first step is importing the NI mate Blender add-on. Open up Blender and go to *File* $\rightarrow$ *UserPreferences* or press *Ctrl* + *Alt* + *U*. Click the Add-on tab and then Install Add-on. Locate the **animation_delicode_ni_mate.py** and click Install From File as shown in FA1.5. The file is available at `https://ni-mate.com/download/`.



C:\Users\IITB\Downloads\blender\

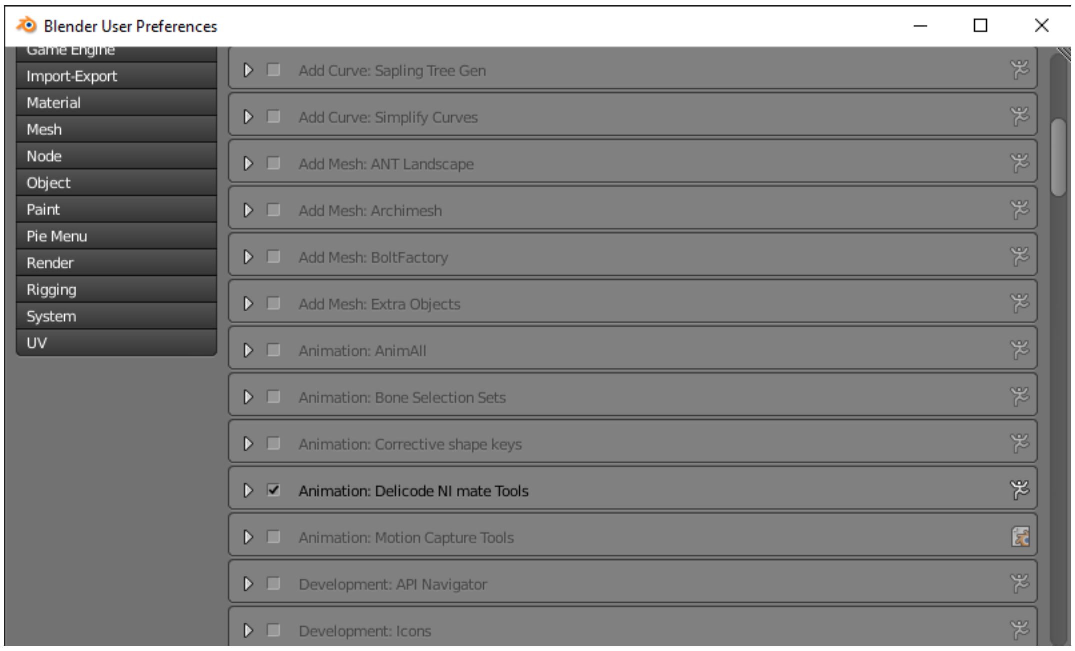
animation_delicode_ni_mate_tools_279.py 35.2 KiB

Install from File...

Overwrite ✓

Target Path: Default

Figure FA1.5: NI Mate Blender Add-on



Blender User Preferences

Game Engine
Import-Export
Material
Mesh
Node
Object
Paint
Pie Menu
Render
Rigging
System
UV

- [ ] Add Curve: Sapling Tree Gen
- [ ] Add Curve: Simplify Curves
- [ ] Add Mesh: ANT Landscape
- [ ] Add Mesh: Archimesh
- [ ] Add Mesh: BoltFactory
- [ ] Add Mesh: Extra Objects
- [ ] Animation: AnimAll
- [ ] Animation: Bone Selection Sets
- [ ] Animation: Corrective shape keys
- [x] Animation: Delicode NI mate Tools
- [ ] Animation: Motion Capture Tools
- [ ] Development: API Navigator
- [ ] Development: Icons

Figure FA1.6: Enabling Add-on

The Animation Delicode NI mate add-on should now be visible, as shown in FA1.6, under the $UserPreferences \rightarrow Add-ons$ tab. Enable it by clicking the checkbox to the right of the plug-in.

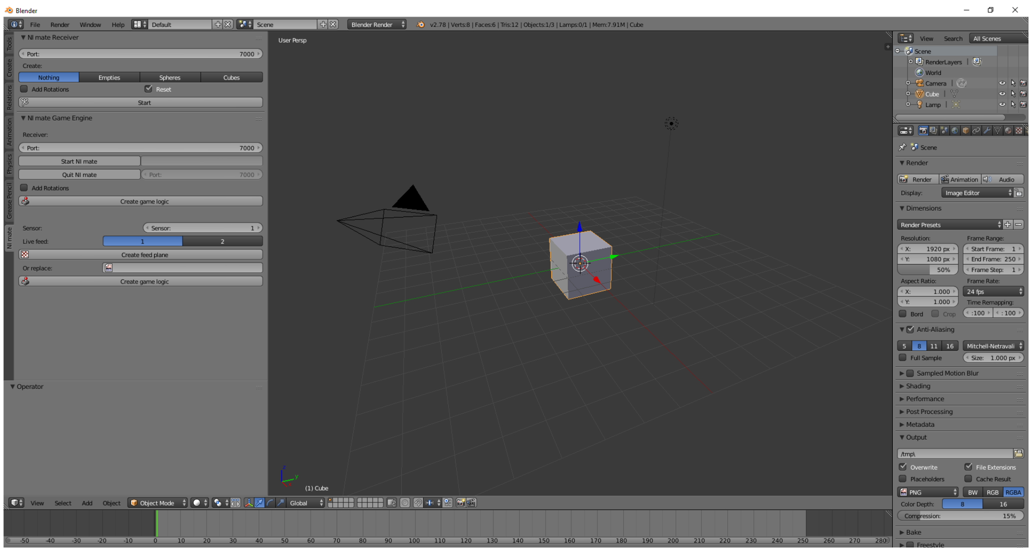


Figure FA1.7: NI Mate Panel in Blender

If the User preferences window is closed, you should notice the Delicode NI mate panel to the left as shown in FA1.7.

In order to check if NI mate is able to transmit skeletal data, shift over to the Delicode NI mate software, and check the box beside Skeleton Tracking. In Blender, click on the NI mate tab in the tool menu. Click on the Empties which will create entities that will represent the joint data being tracked from the Kinect once the Start button is clicked in the plug-in menu.

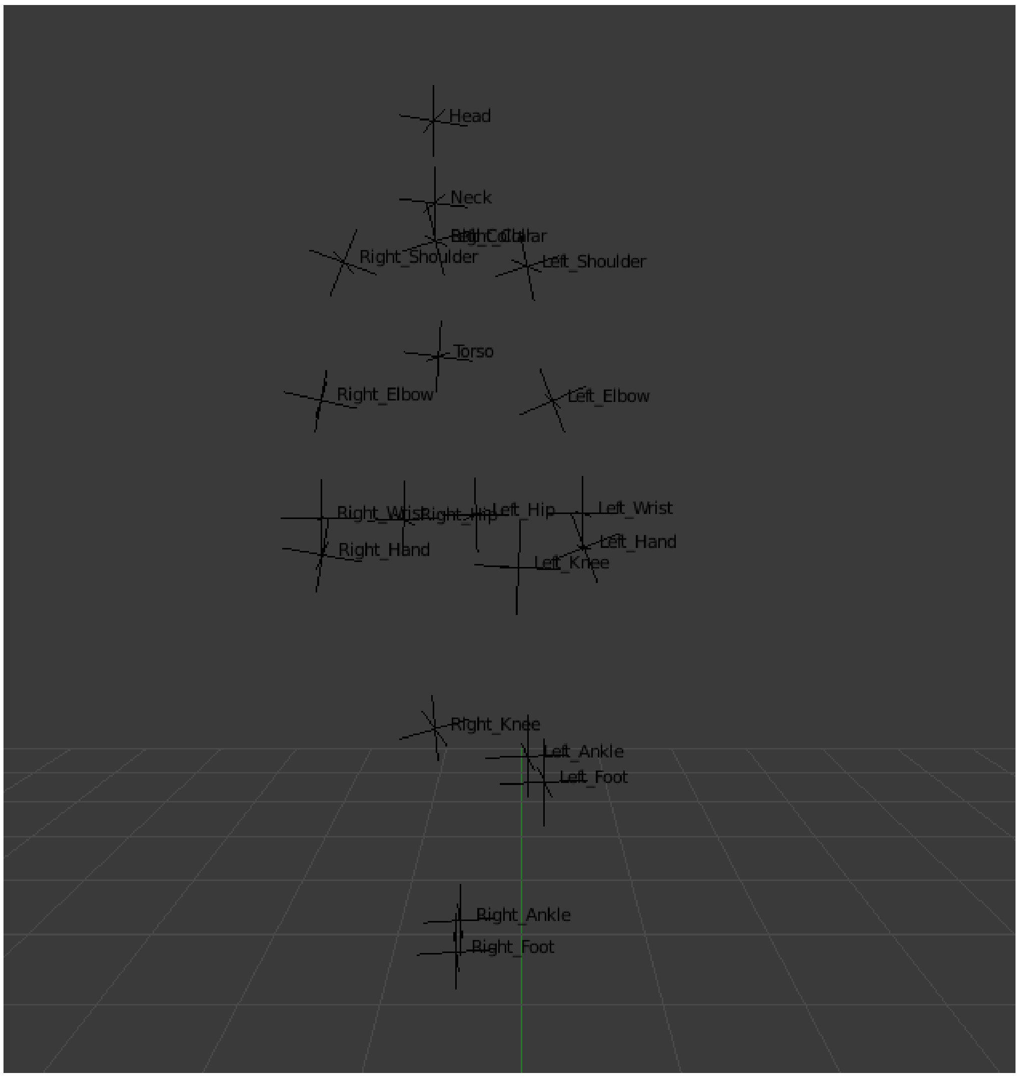
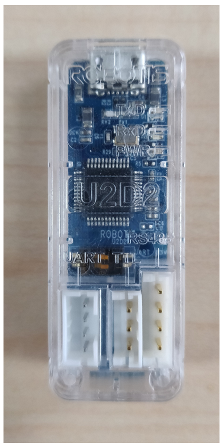


Figure FA1.8: Skeleton Tracking represented by Empties

APPENDIX B

SETTING UP DYNAMIXEL

For the robot to function correctly, each servo needs to be set to a unique ID before they can be programmed to drive the arm. Correspondingly, **U2D2**(FA2.1) is a USB communication converter that enables to control and to operate the Dynamixel with the PC. U2D2 does not supply power to the Dynamixel, therefore, an external power supply should be used to provide power to the Dynamixel. U2D2 coupled with **SMPS2Dynamixel**(FA2.1), which also allows connections to an external power supply, allows to manage Dynamixel's firmware and to check it's status using the Roboplus software as shown in figure FA2.2.


(a) U2D2

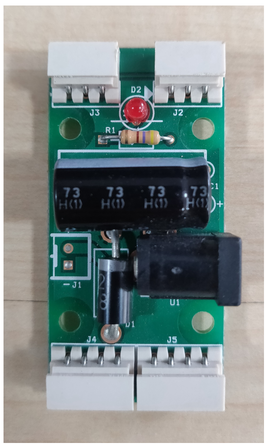

(b) SMPS2Dynamixel

Figure FA2.1: Setting Dynamixel IDs with U2D2 and SMPS2Dynamixel components.

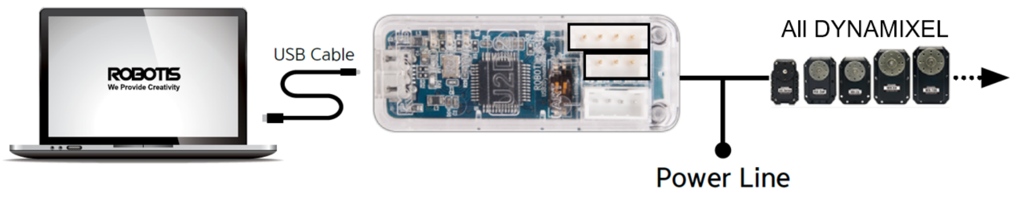


Figure FA2.2: PC to Dynamixel where power line represents SMPS2Dynamixel component in figure FA2.1b

B.1 Connecting Dynamixel

1. Connect Dynamixel to PC through U2D2. (see FA2.2)
2. Connect the external power supply of 12V-5A to SMPS2Dynamixel and couple it to the other connection port in the Dynamixel to supply the power. Notice the **red** LED on the Dynamixel switch on and off. If it does so, then the servo is powered properly.
3. Select the communication port.
4. Search Dynamixel. The search range can be set, if necessary.
5. Any Dynamixels detected can be checked in the list on the left.

B.2 Firmware

Firmware is a program installed in Dynamixel which controls it. There are two protocols that differentiate the firmware. Previous Dynamixels such as the AX model servos work with their standard libraries designed for programming using Arduino and Arbotix microcontrollers. According to Trossen Robotics Support, the MX-106 will also work with these libraries, however the protocol version has to be reverted to protocol 1. As the new MX servos are coming with firmware / protocol 2, it is required to download an older version of Roboplus Software that comprises of Dynamixel wizard that has protocol 1.0 firmware for the 106 servo and manually reset the servo using that software. For instance, the MX servos with firmware v42 or above will comprise of protocol 2.0. It is required to do so as the protocols differ in the functionality of some of the registers. In this study, the firmware used is v41.

Another software that can be used is the Dynamixel Wizard 2.0 that can allow for plotting graphs to track the Dynamixel's voltage, current, load, present position and goal position.

B.2.1 Recovery

When Dynamixel detection fails, ensure that it is properly wired. However, if problems persists, the Dynamixel firmware needs to be restored. After firmware restoration, the ID and baud rate values have to be set again.

B.3 Testing

Once the search is complete, the searched Dynamixel is appeared on the left of the list. The servo ID can be changed and later, signals can be sent to control and move particular Dynamixel using the set ID through programming the Arbotix-M.

Following this, programming the ArbotiX-M requires an FTDI-to-USB device which also may be used to power the board without using an external power supply for testing purposes. Here, the power jumper is moved so that it connects the middle pin and the **USB** pin as shown in figure FA2.3. However, since the Dynamixel servomotors require a nominal voltage of 10-12 volts, an external power source is required to power the board to eventually drive the robotic arm. Therefore, the power jumper is moved so that it connects the middle pin and the **VIN** pin as shown in figure FA2.3.

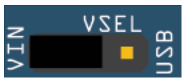


Figure FA2.3: Power Jumper for ArbotiX-M.

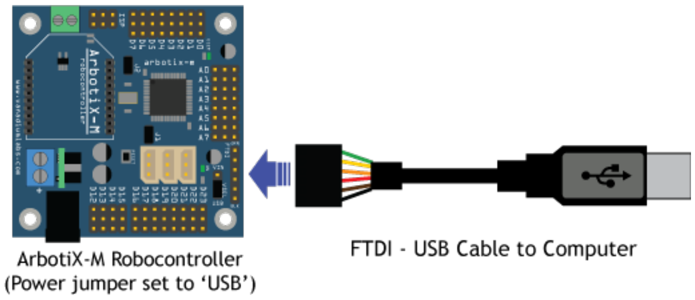


Figure FA2.4: ArbotiX-M to FTDI-USB Connection.

The ArbotiX-M is connected to FTDI-USB connection is shown in figure FA2.4. As the Dynamixel servomotors are designed to be modular and capable of daisy-chaining, the design of the robot arm allows the servos to be connected freeing up space as its open, low-density structure also improves motor heat dissipation thanks to freer air circulation according to [1]. Therefore, it is easier to incorporate motors into the design of the robot as they are connected with each other in a series using the three-pin connectors.

The FTDI port is a dual programming and serial port. By connecting an FTDI device (like a FTDI Cable or a UARTSBee you can program the ArbotiX and relay serial communications. The FTDI port and the XBee socket share a serial port, so only one can be used at a time. To program the ArbotiX while an XBee is connected, you must use an ISP programmer.

The ArbotiX M does not have a dedicated I2C port, but it still supports the I2C protocol on pins 16 and 17 as tabulated in TA2.1.

I2C Pin	Arbotix-M Pin
SCL	16
SDA	17

Table TA2.1: I2C Pins on Arbotix-M.

APPENDIX C

FINGER RIGGING AND WEIGHT PAINTING USING BLENDER

C.1 Finger Rigging using Blender

As previously discussed in chapter 2, the NI-Mate add-on provides functionality to track all the fingers. This functionality allows for the end-effector (gripper) control of virtual arm in Blender using skeletal tracking data from Kinect V2 with regards to MoCap.

Additionally, the individual fingers of the human hand can be rigged in blender to without the need for Inverse Kinematics constraints. Using a few simple constraints and through the process of weight painting, a human hand model is rigged to obtain a rig that allows us to have finer control by being able to control the fingers, proving to be very feasible for animation purposes.

The technique is implemented using the armature or bones with rotation constraints to act as the joints in the finger of a human hand. On top of these constraints, weight paint is a brush type found in Blender which allows the user to create a heatmap which corresponds to the influence of the bone on an mesh object with vertices. Weight paint is used in many situations, and here are a few instances where it is used to simplify the process of rigging and applying modifiers to an object:

- Character Rigging: Through the process of weight painting, the user can determine how much of an effect moving a bone on the model will have on a set of vertices that form the model.
- Physics Simulation: This process also allows the user to define which parts of an object will be affected more or less by the modifiers.

In order to get a better visual appearance and also to improve the process of animation, the bone appearance of Blender is replaced with other shapes to help in understanding the bone movements.

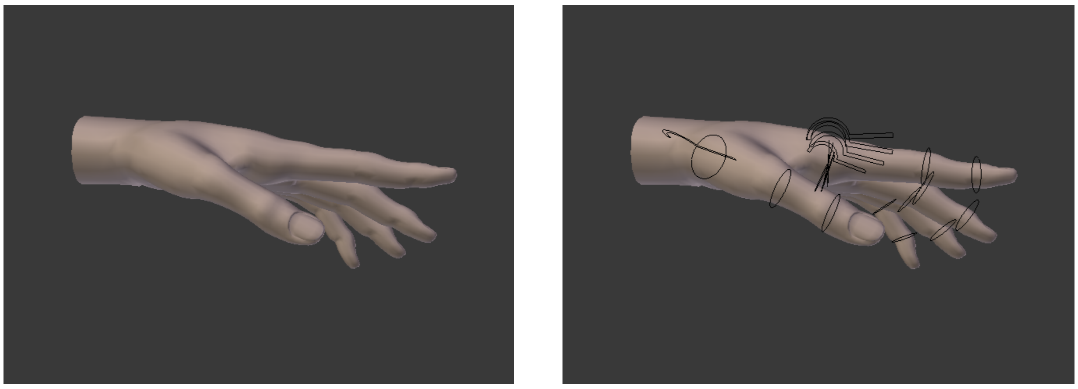

(a) Human Hand model. (b) Rigged Human Hand.

Figure FA3.1: Rigging Fingers in Blender.

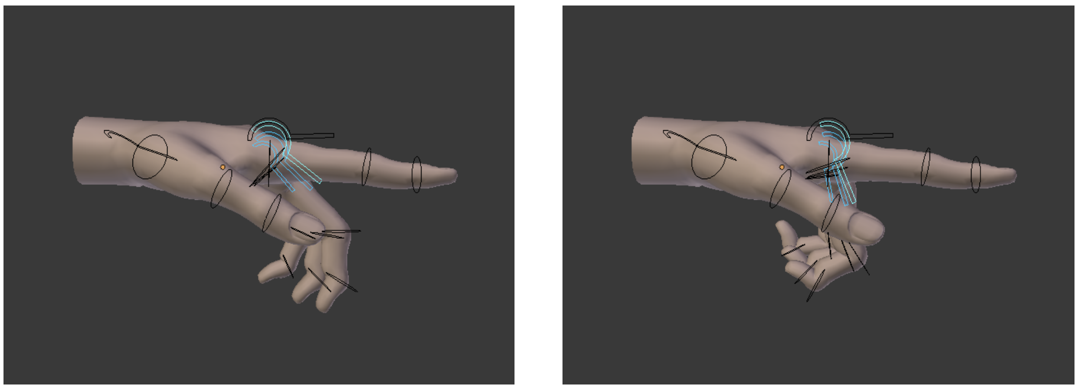
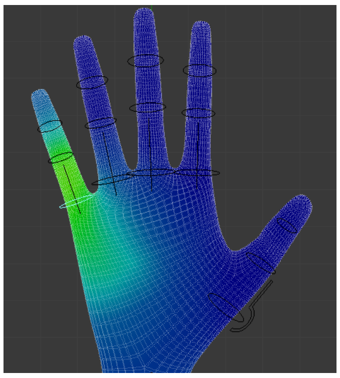
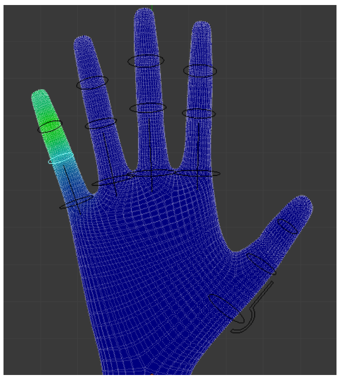


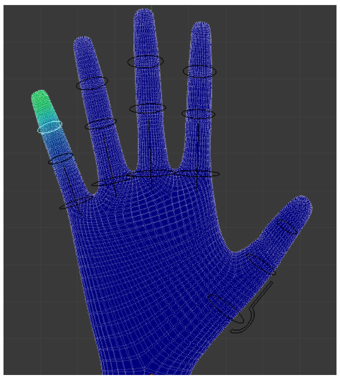
Figure FA3.2: Controlling the finger curl using bone constraints.



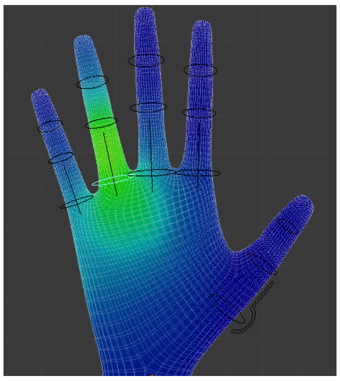
(a) Pinky Metacarpal Influence



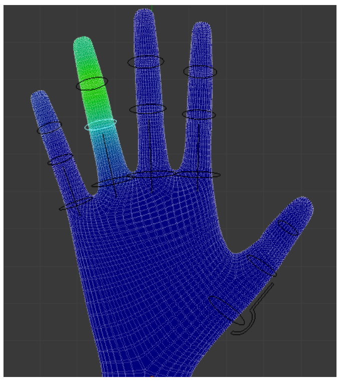
(b) Pinky Proximal Influence



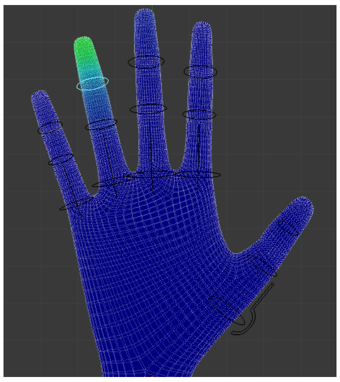
(c) Pinky Distal Influence



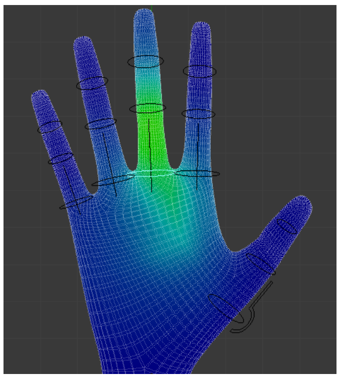
(d) Ring Metacarpal Influence



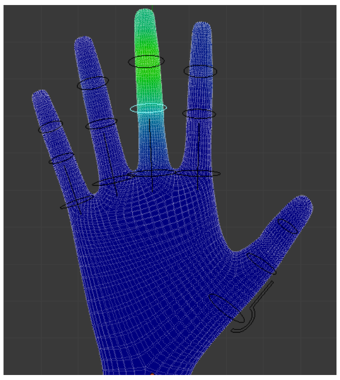
(e) Ring Proximal Influence



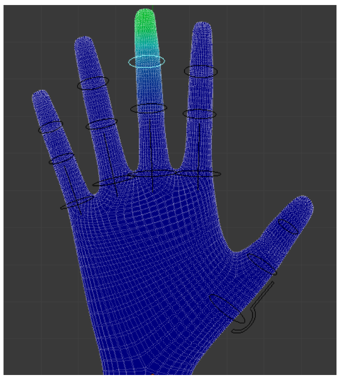
(f) Ring Distal Influence



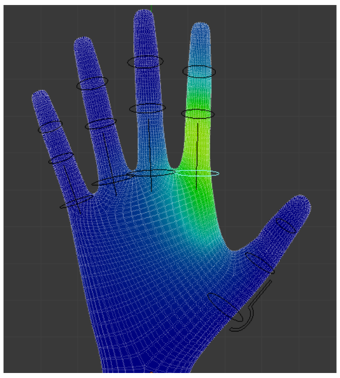
(g) Middle Metacarpal Influence



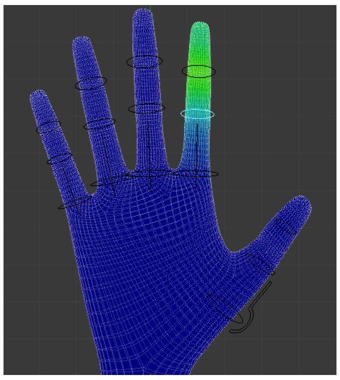
(h) Middle Proximal Influence



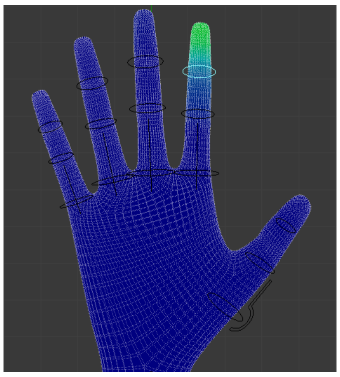
(i) Middle Distal Influence



(j) Index Metacarpal Influence



(k) Index Proximal Influence



(l) Index Distal Influence

Figure FA3.3: Adding weights to the bones in the armature using Weight Painting in Blender.

APPENDIX D

FORWARD AND INVERSE KINEMATICS

D.1 FK: Denavit-Hartenberg Convention

In order to determine the end-effector position in real-world scenario, the following four transformation parameters known as D–H parameters are needed to be determined:

- d - offset along previous z to the common normal.
- θ - angle about previous z, from old x to new x.
- a - offset along previous z to the common normal.
- α - offset along previous z to the common normal.

The reference frames for the joints are laid out as follows:

- the z-axis is in the direction of the joint axis.
- x-axis is parallel to the common normal: $x_n = z_n \times z_{n-1}$ (or away from z_{n-1}).
- If there is no unique common normal (parallel z axes), then d is a free parameter. The direction of x_n is from z_{n-1} to z_n.

- y-axis follows from the x and z-axis by choosing it to be a right-handed coordinate system.

The Denavit-Hartenberg parameters for the current robot arm (till the wrist joint excluding elbow-forearm yaw link) are tabulated in the Table TA4.1.

Link	θ	d(cm)	a(cm)	α
1	θ_1	6.6*	0	90°
2	θ_2	0	0	90°
3	θ_3	0	0	90°
4	θ_4	28*	0	90°
5	θ_5	0	25*	90°

Table TA4.1: Dynamixel configuration for the robotic arm. *The distance between the links are approximate as they are distances between the centers of two Dynamixel Actuators present in the structure.

The position of the end-effector (in this case, till the wrist) can be computed given the individual joint angles (here θ_1, θ_2, θ_3, θ_4, θ_5) about their rotation axis.

D.2 Overview of Jacobian IK

The Jacobian Solver in Blender is an iterative approach to solve the inverse kinematics problem to determine the change in orientations of every joint present in the chain to reach the desired position.

Jacobian methods have three main steps from a top-down perspective:

1. Find the joint configurations: T
2. Compute the change in rotations: dO
3. Compute the Jacobian: J

D.2.1 Joint Configurations

As all the joints are revolute, O is a pose vector which represents the initial orientation of every joint, T is the pose vector which represents the final orientation of every joint, such that the end effector reaches its target position and dO is the vector which represents the change in orientation for each joint, such that the kinematic chain reaches T from O.

$$T = O + dO \times h$$

where h is just a simulation step that can be tuned.

D.2.2 Compute Change in Orientation

The change in orientation is calculated using:

$$V = J \times dO$$

where J is the Jacobian and V is the change in spatial location i.e., V = T-E. The Jacobian is a matrix which represents the relationship between the position of the end effector and the rotation of each joint.

$$J^{-1}V = J^{-1}JdO$$

$$J^{-1}V = dO$$

$$dO = J^{T}V$$

A rough approximation to the Jacobian Inverse that works in many simple cases is replacing the Jacobian Inverse with the Jacobian Transpose. The Jacobian Transpose always exists, as opposed to the Jacobian Inverse and it is computationally less expensive.

D.2.3 Compute Jacobian

Each term in the Jacobian matrix represents how a change in the specified joint angle effects the spatial location of end effector. For instance, the first column of the matrix shows how much the end effector position would change in X-Y-Z coordinate space, if Joint A's angle is changed by a differential amount.

$$J = \begin{bmatrix} (R_A(E-A))_X & (R_B(E-B))_X & (R_C(E-C))_X \\ (R_A(E-A))_Y & (R_B(E-B))_Y & (R_C(E-C))_Y \\ (R_A(E-A))_Z & (R_B(E-B))_Z & (R_C(E-C))_Z \end{bmatrix}$$

where R_A is the axis of rotation of joint A. E is the position of the end effector, as before and A is the position of joint A.

Then number of joints in the articulated body determines the number of columns in the Jacobian.

APPENDIX E

CODE

E.1 Blender: Computing Bone Angles

```
import bpy
import math
import time
import sys
import serial
import glob
import os
from math import degrees
import numpy as np
from mathutils import Vector
os.system('cls')

#assigning COM port and Baud rate for Serial Communication
port=''.join(glob.glob("/dev/ttyUSB*"))
ser = serial.Serial('COM4',115200)
print("connected to: " + ser.portstr)

#setting the armature as the active object in Blender
ob = bpy.data.objects['Armature']
bpy.context.scene.objects.active = ob

#setting pose mode for controlling the arm
bpy.ops.object.mode_set(mode='POSE')

#variables for computing bone orientations
shoulder = ob.pose.bones.get("shoulder")
arma = ob.pose.bones.get("arma")
armb = ob.pose.bones.get("armb")
armc = ob.pose.bones.get("armd")
armd = ob.pose.bones.get("armd")
hand = ob.pose.bones.get("hand")
thumb = ob.pose.bones.get("thumb.001")

bonesl = [shoulder,arma,armb,armc,armd,hand,thumb]
```

```
#check for bones present in Blender
for bones in bpy.context.scene.objects.active.pose.bones:
    # use the decompose method to get matrices
    loc, rot, sca = bones.matrix_basis.decompose()
    # or use the to_quaternion method to get matrices
    rot = bones.matrix_basis.to_quaternion()
    # print the bones available
    print(bones)

def dotproduct(v1, v2):
    """This function computes dot product of two vectors."""
    return sum((a*b) for a, b in zip(v1, v2))

def mag(v):
    """This function computes magnitude of two vectors."""
    return math.sqrt(dotproduct(v, v))

def angle(v1, v2):
    """This function computes angle between two vectors."""
    return (np.arccos(dotproduct(v1, v2) / (mag(v1) * mag(v2))))*(180/math.pi)

def getRoll(bone):
    mat = bone.matrix.to_3x3()
    quat = mat.to_quaternion()
    if abs(quat.w) < 1e-4:
        roll = math.pi
    else:
        roll = 2*math.atan(quat.y/quat.w)
    return roll

def sendAngles():
  #insert visual loc,rot,scale if needed
  for b in bonesl:
        b.keyframe_insert('rotation_euler',options = {"INSERTKEY_VISUAL"})

    # orientation of arma with respect to Global Axis in Blender
    arma_px = math.degrees(Vector((1,0,0)).angle(arma.tail - arma.head)) # +x
    arma_py = math.degrees(Vector((0,1,0)).angle(arma.tail - arma.head)) # +y
    arma_pz = math.degrees(Vector((0,0,1)).angle(arma.tail - arma.head)) # +z
    arma_nx = math.degrees(Vector((-1,0,0)).angle(arma.tail - arma.head)) # -x
    arma_ny = math.degrees(Vector((0,-1,0)).angle(arma.tail - arma.head)) # -y
    arma_nz = math.degrees(Vector((0,0,-1)).angle(arma.tail - arma.head)) # -z

    ### REACHY ARM ANGLES CALCULATIONS

    #shoulderPitch
    shoulderPitch = arma_py

    if(shoulderPitch > 180):
        shoulderPitch = 180

    #shoulderRoll
    if(arma_z_angle1 > 90):
        shoulderRoll = 185 - math.degrees((arma.tail - arma.head).angle(shoulder.tail - shoulder.head))
```

```
if(arma_z_angle1 < 90):
    shoulderRoll = 175 + math.degrees((arma.tail - arma.head).angle(shoulder.tail - shoulder.head))
if(shoulderRoll > 177):
    shoulderRoll = 177

#armYaw
rot = (np.degrees(list(armb.matrix_basis.to_quaternion().to_euler())))
armYaw = int(rot[1])+180

#elbowPitch
elbowPitch = 180 + math.degrees((armb.tail - armb.head).angle(armc.tail - armc.head))
if(elbowPitch > 280):
    elbowPitch = 280

#forearmYaw
armcx = armc.x_axis
armdx = armd.x_axis
rot = (np.degrees(list(armd.matrix_basis.to_quaternion().to_euler())))
forearmYaw = int(75+int(rot[1]))

#wrist roll and wrist pitch
extension=int(math.degrees(armd.z_axis.angle(hand.z_axis)))
flexion=int(math.degrees(armd.z_axis.angle(hand.z_axis)))
abduction=int(math.degrees(armd.x_axis.angle(hand.x_axis)))
adduction=int(math.degrees(armd.x_axis.angle(hand.x_axis)))

up_down = int(math.degrees(armd.z_axis.angle(hand.y_axis)))
left_right = int(math.degrees(armd.x_axis.angle(hand.y_axis)))

#wrist roll
if(up_down > 0 and up_down < 90): #up
    extension=int(math.degrees(armd.z_axis.angle(hand.z_axis)))
    flexion = 0
if(up_down > 90 and up_down < 180): #down
    flexion=int(math.degrees(armd.z_axis.angle(hand.z_axis)))
    extension = 0

#wrist pitch
if(left_right > 0 and left_right < 90): #left
    abduction = 0
    adduction = int(math.degrees(armd.x_axis.angle(hand.x_axis)))
if(left_right > 90 and left_right < 180): #left
    abduction = int(math.degrees(armd.x_axis.angle(hand.x_axis)))
    adduction = 0

Dyna_extension = 150+extension
Dyna_flexion = 150-flexion
Dyna_abduction = 150+abduction
Dyna_adduction = 150-adduction

if(Dyna_extension > 180): #195
    Dyna_extension = 180
if(Dyna_flexion < 85): #76/77
    Dyna_flexion = 85
if(Dyna_abduction > 190): #205
```

```
        Dyna_abduction = 190
    if(Dyna_adduction < 105): #95
        Dyna_adduction = 105

    wrist_pitch = 150
    if(abduction>0): #if orientation is towards thumb
        wrist_pitch = Dyna_abduction
    elif(adduction>0): #if orientation is away from thumb
        wrist_pitch = Dyna_adduction

    wrist_roll = 150
    if(extension>0):
        wrist_roll = Dyna_extension #bends the hand backward and up
    elif(flexion>0):
        wrist_roll = Dyna_flexion #bends the hand forward and up

    #gripper
    thumbrotation = thumb.matrix.to_euler()
    rot = (np.degrees(list(thumb.matrix_basis.to_quaternion().to_euler())))
    thumbopen = int(rot[2])+45
    if(thumbopen < 0 ):
        thumbopen = 0
    if(thumbopen > 105 ):
        thumbopen = 105

    armYaw = str(armYaw)
    shoulderPitch = str(int(shoulderPitch))
    shoulderRoll = str(180-int(shoulderRoll))
    elbowPitch = str(int(elbowPitch))
    forearmYaw = str(int(forearmYaw))
    wrist_pitch = str(int(wrist_pitch))
    wrist_roll = str(int(wrist_roll))
    thumbopen = str(thumbopen)

    currentFrame = int(bpy.context.scene.frame_current)

    val = str(shoulderPitch+","+
      shoulderRoll+","+armYaw+
      ","+elbowPitch+","+forearmYaw+
      ","+wrist_pitch+","+wrist_roll+
      ","+thumbopen+","+str(currentFrame))
    print(val)
    #serially send values
    ser.write((val).encode('UTF-8'))

def frameChange(passedScene):
    sendAngles()

bpy.app.handlers.frame_change_pre.append(frameChange)
```

E.2 Blender: Exporting Graphs from Graph Editor

```
import bpy
import pandas as pd
import math
from math import degrees
import numpy as np
from mathutils import Vector
import os
os.system('cls')

def findAnglesBetweenTwoVectors1(v1s, v2s):
    dot = np.einsum('ijk,ijk->ij',[v1s,v1s,v2s],[v2s,v1s,v2s])
    return np.degrees(np.arccos(dot[0,:]/(np.sqrt(dot[1,:])*np.sqrt(dot[2,:]))))

context = bpy.context
ob = context.object

sce = bpy.context.scene
ob = bpy.context.object

animation_data = []

#from start of the animation till end
for f in range(sce.frame_start, sce.frame_end+1):

    #jump to keyframe
    sce.frame_set(f)

    shoulder = ob.pose.bones.get("shoulder")
    arma = ob.pose.bones.get("arma")
    armb = ob.pose.bones.get("armb")
    armc = ob.pose.bones.get("armd")
    armd = ob.pose.bones.get("armd")
    hand = ob.pose.bones.get("hand")
    thumb = ob.pose.bones.get("thumb.001")

    shoulder_x_angle = math.degrees(Vector((1,0,0)).angle(shoulder.tail - shoulder.head))
    shoulder_y_angle = math.degrees(Vector((0,1,0)).angle(shoulder.tail - shoulder.head))
    shoulder_z_angle = math.degrees(Vector((0,0,1)).angle(shoulder.tail - shoulder.head))

    v1 = shoulder.tail - shoulder.head
    v2 = arma.head - arma.tail
    shouldertoarm = v1.angle(v2)

    v1 = armb.head - armb.tail
    v2 = armc.head - armc.tail
    elbow = v2.angle(v1)

    v1 = armd.tail - armd.head
    v2 = hand.head - hand.tail
    wrist = v1.angle(v2)

    #rotation data: [X,Y,Z]
    rotdata = []
    rotdata.append(f)

    for b in bpy.context.scene.objects.active.pose.bones:
        if b.name in ["hand"]:#,"armb","armc","armd","hand","IK_Target"] :
            # use the decompose method
            loc, rot, sca = b.matrix_basis.decompose()
            # or use the to_quaternion method
            rot = np.degrees(list(b.matrix_basis.to_quaternion().to_euler()))
            rotdata.append(rot[0])
            rotdata.append(rot[1])
            rotdata.append(rot[2])

    animation_data.append(rotdata)


df = pd.DataFrame(animation_data,columns=['Frame', 'arma: euler X','arma: euler Y','arma: euler Z'
                                        ,'armb: euler X','armb: euler Y','armb: euler Z'
                                        ,'armc: euler X','armc: euler Y','armc: euler Z'
                                        ,'armd: euler X','armd: euler Y','armd: euler Z'
                                        ,'hand: euler X','hand: euler Y','hand: euler Z'
                                        ,'IK Target: euler X','IK Target: euler Y','IK Target: euler Z'])

print("making csv",os.getcwd())
print("shape of df ", df.shape)
df.to_csv('animationdata.csv',index=False)
```

E.3 Arbotix-M: Dynamixel Actuation and Motor Feedback

```
#include <ax12.h>
#include <BioloidController.h>
#include <Wire.h>

//  id
//  10 MX-106AT shoulder_pitch
//  11 MX-64AT  shoulder_roll
//  12 MX-64AT  arm_yaw
//  13 MX-64AT  elbow_pitch
//  14 Ax-18A   forearm_yaw
//  15 Ax-18A   wrist_pitch
//  16 Ax-18A   wrist_roll
//  17 Ax-18A   gripper

#define MX_GOAL_POSITION_L          30
#define MX_GOAL_POSITION_H          31
#define MX_GOAL_SPEED_L             32
#define MX_GOAL_SPEED_H             33
#define MX_TORQUE_LIMIT_L           34
#define MX_TORQUE_LIMIT_H           35
#define MX_PRESENT_POSITION_L       36
#define MX_PRESENT_POSITION_H       37
#define MX_PRESENT_SPEED_L          38
#define MX_PRESENT_SPEED_H          39
#define MX_PRESENT_LOAD_L           40
#define MX_PRESENT_LOAD_H           41
#define MX_PRESENT_VOLTAGE          42
#define MX_PRESENT_TEMPERATURE      43 //AX-18A servos do not have this register
#define MX_CURRENT                  68 //AX-18A servos do not have this register

int pos = 0;    // variable to store the servo position
int incomingByte = 0;   // for incoming serial data
String data = "";

String shoulder_pitch = "";
int shoulderPitchServo;
int shoulderPitchDegrees;



String shoulder_roll = "";
int shoulderRollServo;
int shoulderRollDegrees;

String arm_yaw = "";
int armYawServo;
int armYawDegrees;

String elbow_pitch = "";
int elbowPitchServo;
int elbowPitchDegrees;

String forearm_yaw = "";
```

```
int forearmYawServo;
int forearmYawDegrees;

String wrist_pitch = "";
int wristPitchServo;
int wristPitchDegrees;

String wrist_roll = "";
int wristRollServo;
int wristRollDegrees;

String thumb_tip = "";
int endGripperServo;
int endGripperDegrees;

String frame_num = "";
int framenum;

int tempshoulderPitchDegrees = 90;
int tempshoulderRollDegrees = 97; //set to 94 in rest pose
int temparmYawDegrees = 180;
int tempelbowPitchDegrees = 180;
int tempforearmYawDegrees = 150;
int tempwristPitchDegrees = 150;
int tempwristRollDegrees = 150;
int tempendGripperDegrees = 150;
int requiredDifference = 1;

float voltage;
int volt;
int goalposition;
int presentposition;
int current;
int load;
int speedreg;

void setup()
{
    Serial.begin(115200);
    Wire.begin();
    Wire.setClock(400000);

    //turning torque on for motors
    TorqueOn(10);
    TorqueOn(11);
    TorqueOn(12);
    TorqueOn(13);
    TorqueOn(14);
    TorqueOn(15);
    TorqueOn(16);
    TorqueOn(17);

    //setting rest postions for the arm
    SetPosition(10,3072);
    SetPosition(11,3008);
    SetPosition(12,2048);
    SetPosition(13,2048);
    SetPosition(14,512);
    SetPosition(15,512);
    SetPosition(16,512);
    SetPosition(17,512);

    delay(100);//wait for servo to move
}

void loop()
{

  if(Serial.available() > 0 ){

        //Serial.println("Serial available!");
        String incoming=readString();

        //substrings for assigning values to each Dynamixel
        shoulder_pitch = getValue(incoming,',',0);
        shoulder_roll = getValue(incoming,',',1);
        arm_yaw = getValue(incoming,',',2);
        elbow_pitch = getValue(incoming,',',3);
        forearm_yaw = getValue(incoming,',',4);
        wrist_pitch = getValue(incoming,',',5);
        wrist_roll = getValue(incoming,',',6);
        thumb_tip = getValue(incoming,',',7);
        frame_num = getValue(incoming,',',8);
        framenum = frame_num.toInt();

        //10
        shoulderPitchDegrees = shoulder_pitch.toInt();
        shoulderPitchServo = map(shoulderPitchDegrees, 0, 180, 4095, 2048);
        //11
        shoulderRollDegrees = shoulder_roll.toInt();
        shoulderRollServo = map(shoulderRollDegrees, 0, 90, 2048, 3008);
        //12
        armYawDegrees = arm_yaw.toInt();
        armYawServo = map(armYawDegrees, 0, 360, 0, 4095);
        //13
        elbowPitchDegrees = elbow_pitch.toInt();
        elbowPitchServo = map(elbowPitchDegrees, 0, 360, 0, 4095);
        //14
        forearmYawDegrees = forearm_yaw.toInt();
        forearmYawServo = map(forearmYawDegrees, 0, 160, 207, 820);
        //15
        wristPitchDegrees = wrist_pitch.toInt();
        wristPitchServo = map(wristPitchDegrees, 105, 190, 274, 700);
        //16
        wristRollDegrees = wrist_roll.toInt();
        wristRollServo = map(wristRollDegrees, 85, 180, 257, 605);
        //17
        endGripperDegrees = thumb_tip.toInt();
        endGripperServo = map(endGripperDegrees,0,105,340,595);
        if(abs(shoulderPitchDegrees - tempshoulderPitchDegrees) > requiredDifference && shoulderPitchDegrees < 177
              && shoulderPitchDegrees > 3){
           tempshoulderPitchDegrees = shoulderPitchDegrees;
           SetPosition(10,shoulderPitchServo);
        }
        if(abs(shoulderRollDegrees - tempshoulderRollDegrees) > requiredDifference && shoulderRollDegrees < 89 &&
              shoulderRollDegrees > 3){
           tempshoulderRollDegrees = shoulderRollDegrees;
           SetPosition(11,shoulderRollServo);
        }
        if(abs(armYawDegrees - temparmYawDegrees) > requiredDifference && armYawDegrees < 270 && armYawDegrees >
              90){
           temparmYawDegrees = armYawDegrees;
           SetPosition(12,armYawServo); //armYawServo
        }
        if(abs(elbowPitchDegrees - tempelbowPitchDegrees) > requiredDifference && elbowPitchDegrees > 180 &&
              elbowPitchDegrees < 280){
           tempelbowPitchDegrees = elbowPitchDegrees;
           SetPosition(13,elbowPitchServo);
        }
        if(abs(forearmYawDegrees - tempforearmYawDegrees) > requiredDifference){
           tempforearmYawDegrees = forearmYawDegrees;
           SetPosition(14,forearmYawServo);
        }
        if(abs(wristPitchDegrees - tempwristPitchDegrees) > requiredDifference && wristPitchDegrees > 100 &&
              wristPitchDegrees < 200){
           tempwristPitchDegrees = wristPitchDegrees;
           SetPosition(15,wristPitchServo);
        }

        if(abs(wristRollDegrees - tempwristRollDegrees) > requiredDifference && wristRollDegrees > 80 &&
              wristRollDegrees < 195){
           tempwristRollDegrees = wristRollDegrees;
           SetPosition(16,wristRollServo);
        }
        if(abs(endGripperDegrees - tempendGripperDegrees) > requiredDifference && endGripperDegrees > 0 &&
              endGripperDegrees < 105){
           tempendGripperDegrees = endGripperDegrees;
           SetPosition(17,endGripperServo);
        }

        computeMotorFeedback(10);
        computeMotorFeedback(11);
        computeMotorFeedback(12);
        computeMotorFeedback(13);
        computeMotorFeedback(14);
        computeMotorFeedback(15);
        computeMotorFeedback(16);
        computeMotorFeedback(17);
    }
}

void computeMotorFeedback(int id){
  voltage = ax12GetRegister (id, AX_PRESENT_VOLTAGE, 1)/1.0;
  volt = voltage;
  goalposition = ax12GetRegister (id, MX_GOAL_POSITION_L, 2);
  presentposition = ax12GetRegister (id, MX_PRESENT_POSITION_L, 2);
  current  = ax12GetRegister (id, MX_CURRENT, 2);
  load =  ax12GetRegister (id, MX_PRESENT_LOAD_L, 2);
  speedreg = ax12GetRegister (id, MX_PRESENT_SPEED_L, 2);
  sendMotorFeedback( id, volt, goalposition, presentposition, current, load, speedreg );
}

void sendMotorFeedback(int id,int volt,int goalposition,int presentposition,int current,int load,int speedreg ){
  Wire.beginTransmission(8);
  Wire.write(id);
  Wire.write(volt);
  Wire.write(highByte(goalposition));
  Wire.write(lowByte(goalposition));
  Wire.write(highByte(presentposition));
  Wire.write(lowByte(presentposition));
  Wire.write(highByte(current));
  Wire.write(lowByte(current));
  Wire.write(highByte(load));
  Wire.write(lowByte(load));
  Wire.write(highByte(speedreg));
  Wire.write(lowByte(speedreg));
  Wire.write(highByte(framenum));
  Wire.write(lowByte(framenum));
  Wire.endTransmission();
}

String getValue(String data, char separator, int index)
{
  int found = 0;
  int strIndex[] = {0, -1};
  int maxIndex = data.length()-1;

  for(int i=0; i<=maxIndex && found<=index; i++){
    if(data.charAt(i)==separator || i==maxIndex){
        found++;
        strIndex[0] = strIndex[1]+1;
        strIndex[1] = (i == maxIndex) ? i+1 : i;
    }
  }
  return found>index ? data.substring(strIndex[0], strIndex[1]) : "";
}

String readString(){
  String inString ="";
  char inChar;
  while(Serial.available()>0){
    inChar =(char) Serial.read();
    inString+=inChar;
    delay(1);
  }
  return inString;
}

int CheckCurrent(int id){
  int current = (ax12GetRegister (id, 68, 2));
  return current;
}

int CheckVoltage(int id){
  float voltage = (ax12GetRegister (id, AX_PRESENT_VOLTAGE, 1)) / 10.0;
  if (voltage < 10.0){
    Relax(id);
    while(1);
    return -1;
  }
  if (voltage >= 10.0){
    return voltage;
  }
}
```

E.4 Arduino: Receiving I2C communication (Motor Feedback)

```
#include<Wire.h>

byte receivedval;
void setup(){
  Wire.begin(8);
  Wire.setClock(400000);
  Wire.onReceive(receiveEvent);
  Serial.begin(115200);
}

void loop(){
  delay(100);
}

//14 Bytes of Data
void receiveEvent(int howMany){

  if(howMany == 14){
    int result;
    result = Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);
    Serial.print(",");

    result = Wire.read();
    result <<=8;
    result |= Wire.read();
    Serial.print(result);Serial.println(".");
  }
}
```