Palestine Polytechnic University

College of IT and Computer Engineering

Department of Computer Engineering

Kinect based Teleoperation of a Humanoid Robot

Supervisor

Dr. Hashem Tamimi

Team Members

Mohannad Albustanji

Eid Sonkrot

Huthaifa Rabae

2023-2024

Abstract

Teleoperation of robots has become increasingly important in a wide range of industries and applications, from manufacturing and logistics to medicine and disaster response. The ability to remotely control robots to perform complex tasks can significantly improve efficiency and safety, particularly in hazardous or hard-to-reach environments. The goal of this project is to develop a teleoperation system that can accurately replicate human movements using a Kinect sensor and a Nao robot. The system utilizes inverse kinematics to enable the robot to mimic human movements.

Specifically, the system will leverage the Microsoft Kinect V2 sensor to capture 3D skeletal joint data in real time, which will be processed and used to control the movements of the Nao robot. Choosing Kinect v2 over image processing to extract human skeleton offers the benefits of dedicated depth sensing, accuracy, and ease of integration.

We investigated two approaches to translate the human poses into the corresponding command on the robot joint. The first one is based on Deep Learnings and the second one is based on inverse kinematics. In the implementation, we decided to proceed with the second approach due to the time required to complete the graduation project. The Inverse Kinematics approach works on a chain level such that each Kinect skeleton chain is converted into Nao body chain. We work on 3 main chains in this project; head, arm and leg.

As the result, we were able to make the Nao robot imitate human in many poses and situations. Success was greater in individual chain movements than in complex movements and poses just as walking, mermaid and seated, with small errors in terms of centimeters,

Contents

List of Figures

List of Tables

Chapter 1: Introduction

The development of motion detection mechanisms enhances the development of new applications for human motion analysis and reproduction. Kinect sensor and humanoid robot are two different technologies with various trends and applications. We plan to create a new trend with this combination, developing an interface whose applications will have future implications.

1.1 Project Description

The proposed system is designed to capture human motion data using a Kinect sensor and reproduce the movements in real-time using NAO robots. The system consists of three main components: the Kinect sensor, the microcontroller, and the NAO robot.

The Kinect sensor is a device that uses depth sensing technology to capture detailed motion data from a human performer. Infrared wave is emitted and its reflection is detected by the Kinect sensor from the environment [7], creating a 3D model of the performer's body. The sensor can track the position and orientation of the performer's joints, providing information about their movements.

The motion data captured by the Kinect sensor is transmitted to a microcontroller using a wired USB 3.0 cable. The microcontroller processes the data and sends it to the NAO robot, which is equipped with software that can analyze and reproduce the movements in real-time. The NAO robot is a humanoid robot developed by Softbank Robotics, designed to mimic human movements and interact with humans in a natural way. The robot is equipped with sensors and cameras that allow it to perceive its environment and respond to external stimuli.

The system can be used to capture a wide range of human movements, from simple gestures to complex dance routines. The NAO robot can reproduce the movements in real-time, allowing for immediate feedback and analysis [8].

Overall, the proposed system offers a cost-effective and efficient solution for capturing and analyzing human motion data. The combination of the Kinect sensor and the NAO robot provides a flexible and versatile platform for real-time reproduction of motion data, with potential applications in a wide range of industries.

1.2 Benefits

1. To deliver an affordable and efficient method for capturing and analyzing human motion data utilizing the Kinect sensor and NAO robot.
2. To foster the development of innovative applications in areas like physical therapy, sports training, and entertainment, focusing on human motion analysis and replication.
3. To bolster research in human-computer interaction by offering a dynamic and adaptable platform to explore interactions between humans and robots.
4. To enhance the application of robotics and AI in educational settings, providing practical learning opportunities for students and researchers in robotics and computer science.

1.3 Report outline

Week 1 start from 1 Sep

Table 11-1 Project Schedule

Task/Week	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
QA plan															
Project plan															
Plan review															
Project analysis															
Requirements															
Project design & implementation															
Testing															
Documentation															

Chapter 2: Background

2.1 Overview

This chapter serves as an introductory overview of the essential components and theoretical foundations that shape our project. We will start by delving into the theoretical background, emphasizing recent advancements in trigonometric equations and mathematical modeling.

Our project heavily relies on these mathematical principles, and a comprehensive grasp of their fundamentals is imperative for its successful execution. Specifically, we will delve into the latest algorithms and techniques in mathematical modeling, essential for empowering our teleoperation system to function with precision and intricacy. This chapter aims to furnish readers with a solid understanding of the fundamental concepts and methodologies employed in our project, laying the groundwork for a more detailed exploration of our system in subsequent chapters

2.2 Hardware background

2.2.1 Humanoid robot (Nao)

- NAO is a versatile humanoid robot created by SoftBank Robotics [19].
- NAO has become a benchmark in the world of research and education.
- With its advanced sensors and interactive capabilities, NAO can adapt to any environment and interact with people in a natural way.
- it is fully programmable and customizable, making it possible to create application solutions that enable it to perform tasks in different areas based on all of his capabilities, including dialogue and motion [19].

Nao hardware specifications :

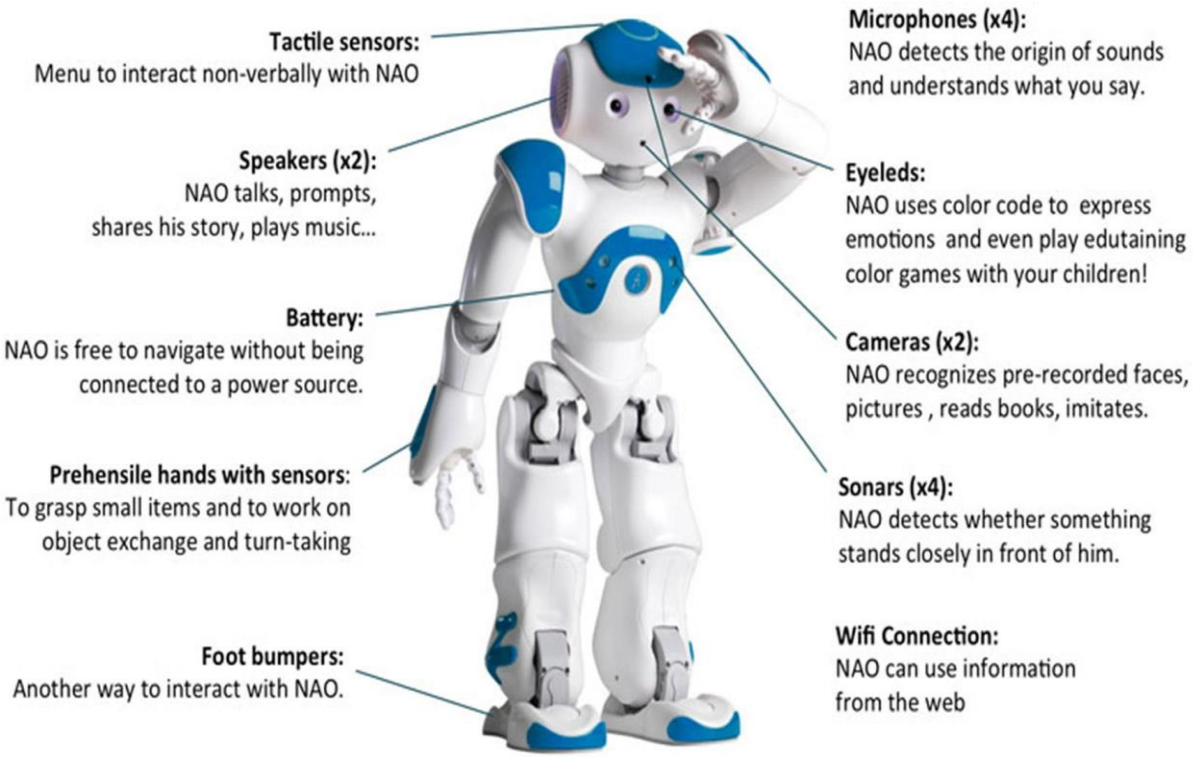
1. Dimensions:
    - Height: 58 cm (22.8 inches)
    - Width: 27 cm (10.6 inches)

- Depth: 22 cm (8.7 inches)
- Weight: Approximately 4.3 kg (9.5 lbs.)

2. Processing Unit:
    - Intel Atom Z530 Processor (1.6 GHz)
3. Memory:
    - 1 GB DDR2 RAM
4. Storage:
    - 2 GB Flash Memory
5. Sensors:
    - Two HD cameras (resolution: 1280x960 pixels)
    - Ultrasonic sensors for obstacle detection
    - Touch sensors on the head, hands, and feet
    - Inertial measurement unit (accelerometer and gyro-meter)
    - 4 Microphones for sound localization and voice recognition
6. Connectivity:
    - Wi-Fi (IEEE 802.11 b/g)
    - Ethernet port (100 Mbps)
    - USB 2.0 port
7. Power:
    - Battery: Lithium-ion battery
    - Battery capacity: 4800 mAh with approximately 60-90 minutes of continuous operation
    - Charging Time: Approximately 90 minutes
8. Operating System:
    - Aldebaran NAOqi OS (based on Linux)
9. Actuators:
    - 25 Degrees of Freedom (DOFs) for joint movement
    - Electric motors for actuation, including head, arms, hands, legs, and feet

10. Additional Features:
    - Text-to-speech synthesis
    - Voice and sound recognition
    - LED lights for visual feedback
    - Speaker for audio output

Figure 2.1, shows Nao Components:



The following image (Figure 2.2) explain Nao joints names, Field of motion and Direction

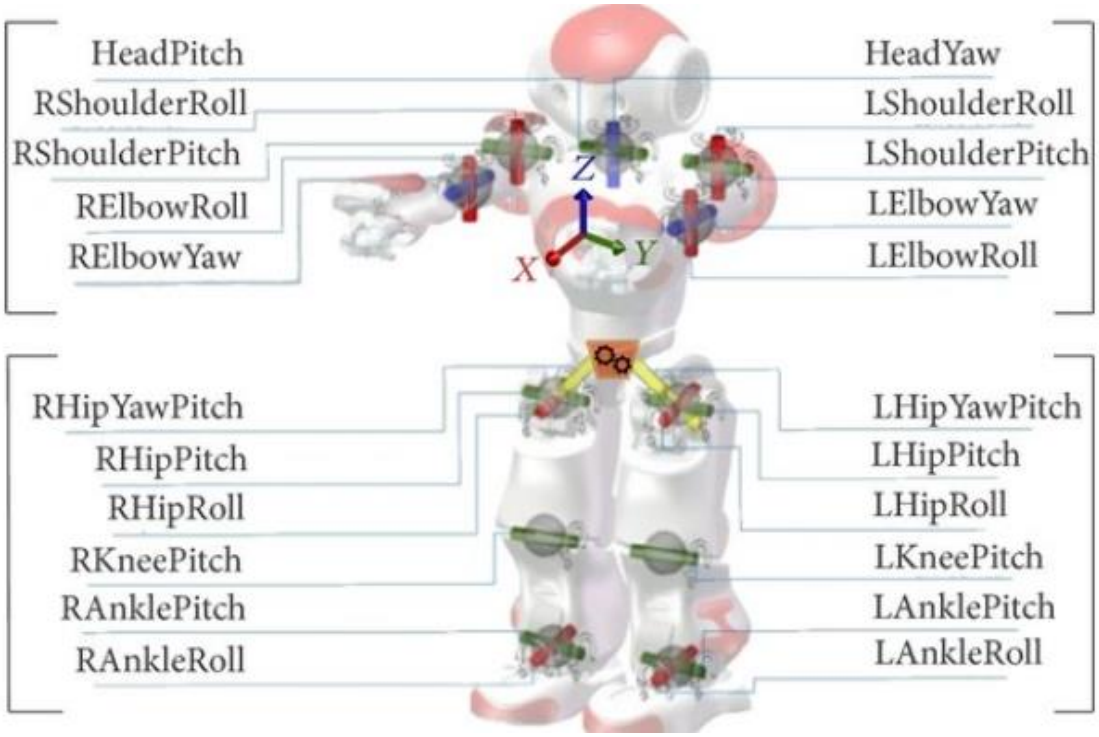


Figure 2-2 Nao Joints

2.2.2 Microsoft Kinect v2

Kinect v2 is a depth-sensing camera developed by Microsoft, primarily used for capturing skeletal joint data and depth information by a set of sensors shown in Figure 2.3. It offers an enhanced user experience in motion tracking, gesture recognition, and 3D depth sensing. The operation principle of Kinect v2 is ToF (Time-of-Flight) with modulation up to 130MHz, acquisition rate = 30Hz. In this section, we will explore the main components that Kinect works with [22][23].

1. **Depth Sensing:** Kinect v2 utilizes an infrared depth sensor that captures the depth information of the surrounding environment. It can measure the distance between the sensor and objects in its field of view. It provides a depth resolution of up to 512x424 pixels, Field of view (H=70.6º x V=60º), depth range (0.5-4.5) m, with ability to capture depth information with sub-millimeter precision [23].
2. **Color Camera:** In addition to depth sensing, Kinect v2 is equipped with a high-definition RGB color camera. It captures color images with a resolution of 1920x1080 pixels, FOV (H=84.1º x V=53.8º), enhancing the visual fidelity of the captured data [23].

3. **Infrared Camera:** Kinect v2 includes an infrared (IR) camera that works in tandem with the depth sensor. It measures the distance between the camera and objects by projecting IR patterns and analyzing the distortion caused by their interaction with the environment [22].
4. **Microphone Array:** The device incorporates a built-in microphone array that enables voice recognition and audio processing. This feature facilitates natural language interaction and voice commands [23].

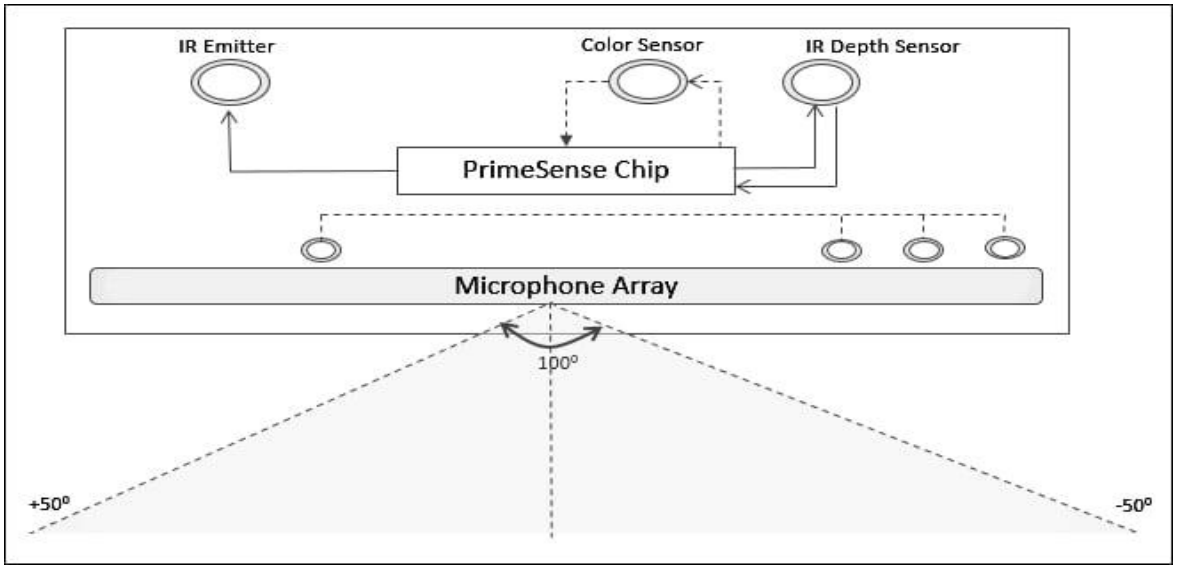


Figure 2-3 Microsoft Kinect

2.2.3 Microcontroller (Latte panda Intel Cherry Trail Z8350)

A Latte Panda is a complete Windows 10 single board computer. It has everything a regular PC has and can do anything that a regular PC does. It is compatible with almost every gadget like printers, joysticks, cameras and more. Any peripherals that work on PC will work on a Latte Panda. A Latte Panda comes pre-installed with a full edition of Windows 10 Home edition, so users can run powerful tools such as Visual Studio, NodeJS, Java and more.

Latte panda Specifications [24]

According to [24], the following is the microcontroller specification

1. Processor:
    - Intel® Cherry Trail Z8350 Quad Core
2. Co-processor Arduino integrated:
    - ATmega32u4
3. Operation System:
    - Windows 10
4. Memory:
    - (DFR0418, DFR0444 2GB DDR3L), (DFR0419 4GB DDR3L)
5. Storage Capability:
    - (DFR0418, DFR0444 32GB), (DFR0419 64GB)
6. GPU:
    - Intel HD Graphics, 12 EUs @ 200-500Mhz, single-channel memory
7. Connectivity:
    - USB 3.0 port and two USB 2.0 ports
    - Wi-Fi 802.11n 2.4G
    - Bluetooth 4.0
    - Supports 100Mbps Ethernet
8. Video output
    - HDMI and MIPI-DSI
9. Onboard touch panel overlay connector
10. GPIO
11. Power:
    - 5V/2A Power
12. Dimension of board
    - 88mm x 70mm/3.46 x 2.76"
13. Packing Size:
    - 110mm x 94mm x 30mm/4.33 x 3.70 x 1.18"
14. N.W.: 55g
15. G.W.: 100g

16. Included:
    - Latte Panda 4G/64GB
    - WiFi antenna
    - Safety precautions

The following Figure 2.4 Shows the Latte panda microcontroller components.

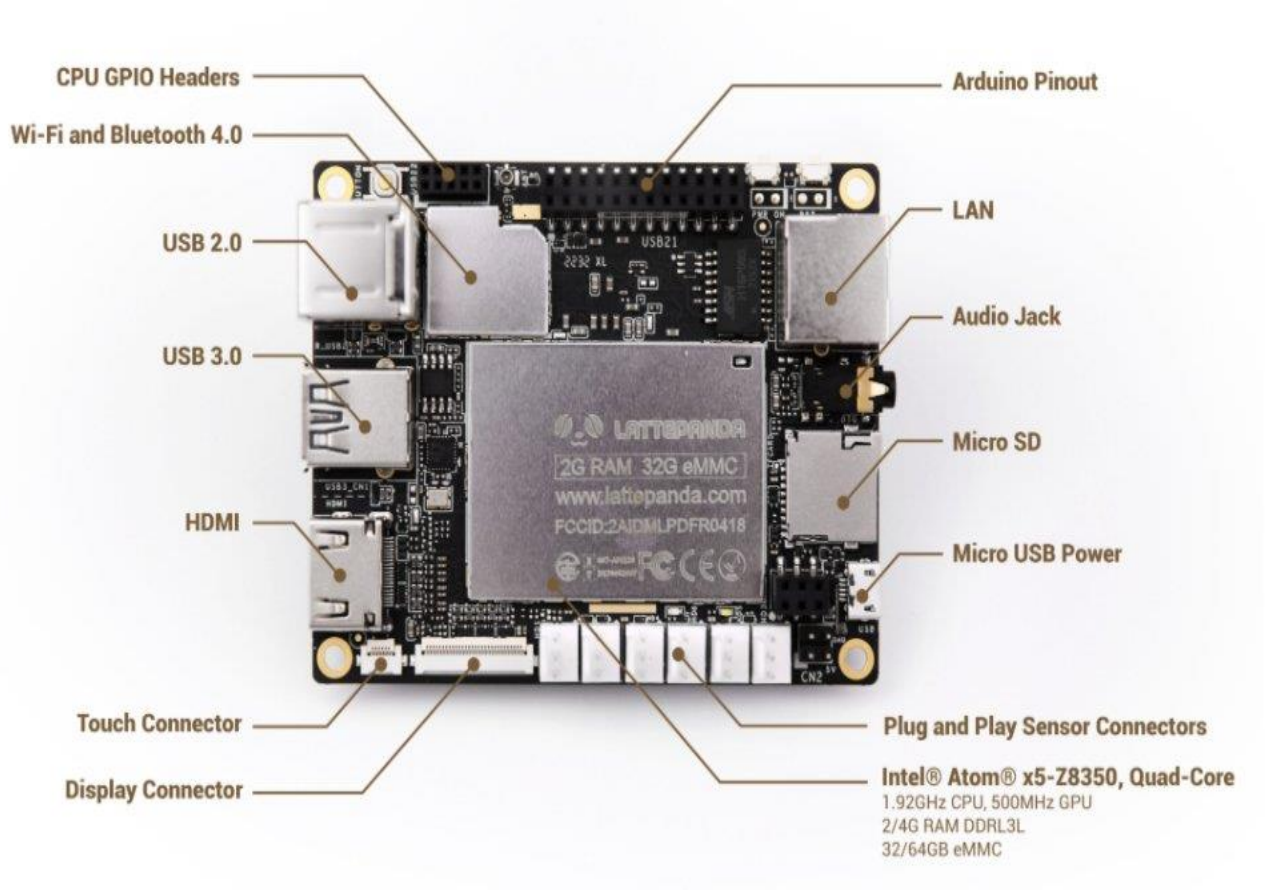


Figure 2-4 Latte Panda intel cherry trail z8350 Components [24]

2.3 Software Background

The software component of our project plays a crucial role in processing the data from the Kinect sensor and controlling the Nao robot's movements. This section provides an overview of the algorithms, programming languages and frameworks used in the development of our software.

2.3.1 Programming Languages

For our project, we have decided to use Python as the primary programming language (for Inverse Kinematics and deep learning). Python is a popular language for scientific computing, data analysis, and artificial intelligence due to its simplicity, readability, and vast library support. Also, we have decided to use C# to read depth data from Kinect sensor.

2.3.2 Simulator

Webots is a robotics simulation software widely used in both academia and industry for robotics and machine learning applications. It offers a virtual environment where developers can design, program, and test their robot models before implementing them in real life [10]. Key features of Webots include:

1. Realistic Physics Simulation: Webots provides a realistic simulation of robot mechanics and environmental interactions, essential for accurate testing of control algorithms.
2. Versatile Environment Customization: Users can design a variety of custom environments in Webots, allowing for extensive testing of robot performance under different conditions.
3. Compatibility with Machine Learning Frameworks: Webots supports integration with popular machine learning platforms like TensorFlow, enabling the development and testing of advanced robot control algorithms.
4. Open-Source Availability: As an open-source platform, Webots allows for code modifications and adaptations to fit specific project needs.

2.4 Algorithms and theoretical background

In this section we will explain the algorithms and theoretical background of Inverse Kinematics and Machine learning approaches.

2.4.1 Inverse Kinematics

Inverse kinematics refers to the mathematical process of determining the joint parameters that will position a humanoid robot's limbs in a desired configuration. In the context of both humans and humanoid

robots, inverse kinematics is crucial for planning and controlling movements. It involves calculating the joint angles or positions as shown in Figure 2.5 required to achieve a specific end-effector position or orientation. Θ is the angle calculated by Inverse Kinematics approach form joints positions in 3D space (X,Y,Z).

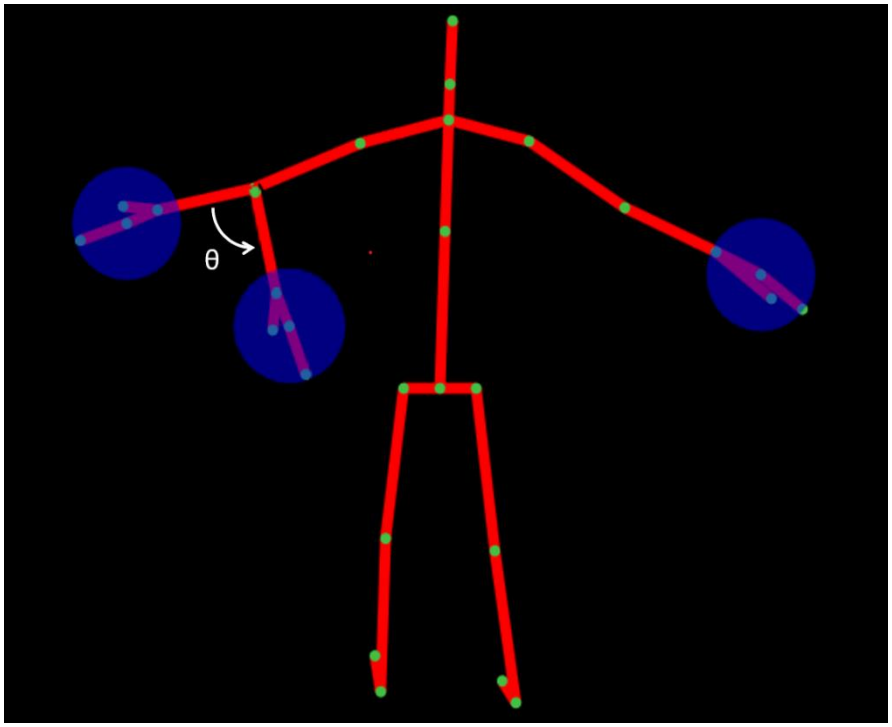


Figure 2-5 Kinect Skeleton

The given mathematical model comprises a series of computations aimed at explain the angles of different joints within a robotic framework. These mathematical calculations constitute fundamental elements in teleoperation systems, enabling management of robotic motion. Specifically, the model delineates the following functionalities:

The explanation provided in Figure 2.6 aims to enhance the clarity of our calculations. By focusing on the relationship between the three joints: shoulder, elbow, and wrist, we can observe two fixed distances: the Upper Arm and forearm lengths. Leveraging this fixed aspect, we can construct a triangle utilizing these lengths alongside other non-fixed distances. Employing inverse trigonometric functions, such as arctan, allows us to determine the angles at which different motors need to rotate. This approach facilitates a deeper understanding of the mechanics involved in coordinating the movements of these joints.

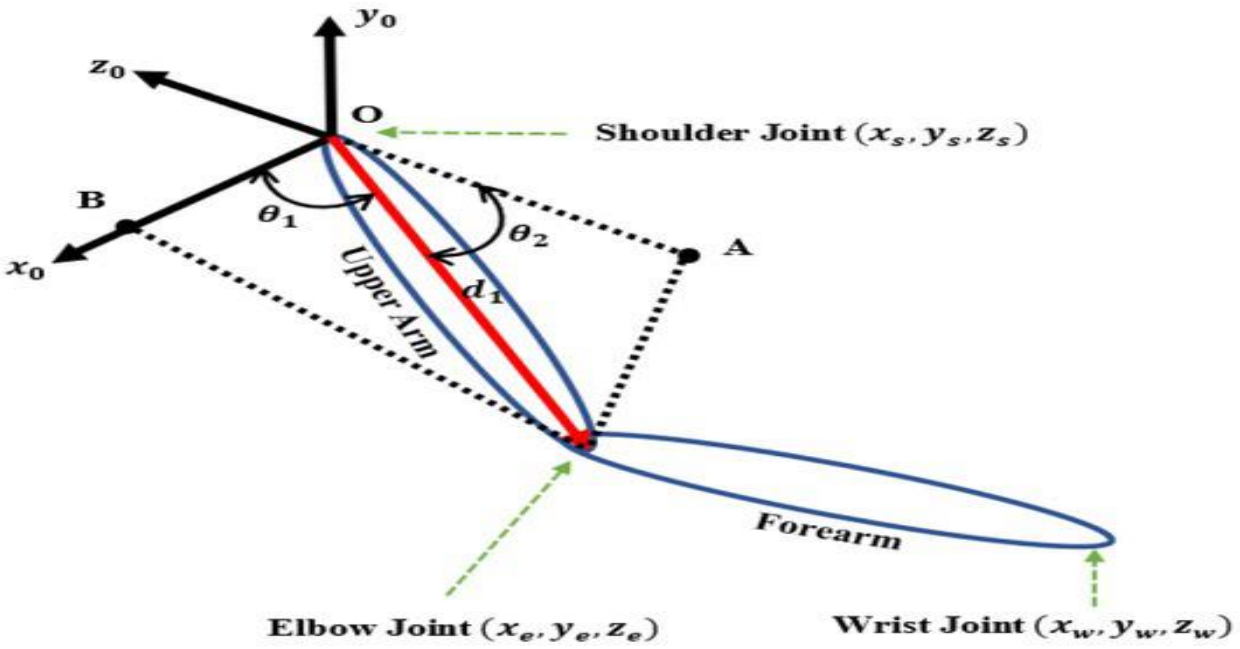


Figure 2-6 Joints with Angles

1. **Shoulder Pitch Computation (Right and Left):** These calculations illustrate the pitch angle of each shoulder, considering their spatial coordinates within a three-dimensional framework. We determine the pitch motors utilizing both the Elbow and Shoulder joints, as indicated in Equations **(2.1)** and **(2.2)**.

   - **Case ($Elbow_y < Shoulder_y$)**:

     $$ShoulderPitch(\theta) = \tan^{-1}\left(\frac{|Elbow_y - Shoulder_y|}{|Elbow_z - Shoulder_z|}\right) \quad (2.1)$$

   - **Case($Elbow_y >= Shoulder_y$)**:

     $$ShoulderPitch(\theta) = \frac{\pi}{2} - \tan^{-1}\left(\frac{|Elbow_z - Shoulder_z|}{|Elbow_y - Shoulder_y|}\right) \quad (2.2)$$

2. **Shoulder Roll Computation (Right and Left):** These computations elucidate the roll angle of each shoulder. Furthermore, in determining the shoulder roll, we employ both the Pitch of the Elbow and the Shoulder. These calculations entail a singular case, as demonstrated in Equation (**2.3**).

   $$ShoulderRoll(\theta) = \tan^{-1}\left(\frac{|Elbow_x - Shoulder_x|}{|Elbow_z - Shoulder_z|}\right) \quad (2.3)$$

3. **Elbow Yaw Computation (Right and Left)**: This facet of the model computes the yaw angle for each elbow, integrating factors such as the shoulder pitch and the spatial alignment of the elbow concerning the shoulder, as depicted in Equation (**2.4).**

$$ElbowYaw(\theta) = -\tan^{-1}\left(\frac{|Wrist_z - Elbow_z|}{|Wrist_y - Elbow_y|}\right) + ShoulderPitch(\theta) \quad (2.4)$$

4. **Elbow Roll Computation (Right and Left):** These calculations ascertain the roll angle for each elbow, taking into account the positions of the elbow joints alongside other pertinent factors. Furthermore, the wrist joint is factored into the analysis. We opt to utilize the Shoulder Pitch angle since the Yaw motors are contingent on Pitch. Although roll could serve as an alternative, Pitch is favored due to its computational simplicity, particularly considering the complexity of the calculations. Hence, four Equations (**2.5, 2.6, 2.7, and 2.8**) are necessary to address these complexities.

$$Upper\ Arm = \sqrt{\left(Sholder_y - Elbow_y\right)^2 + (Sholder_x - Elbow_x)^2 + (Sholder_z - Elbow_z)^2} \quad (2.5)$$

$$Fore\ Arm = \sqrt{\left(Elbow_y - Wrist_y\right)^2 + (Elbow_x - Wrist_x)^2 + (Elbow_z - Wrist_z)^2} \quad (2.6)$$

$$Shoulder\_Wirst = \sqrt{\left(Shoulder_y - Wrist_y\right)^2 + (Shoulder_x - Wrist_x)^2 + (Shoulder_z - Wrist_z)^2} \quad (2.7)$$

$$ElbowRoll(\theta) = \pi - \cos^{-1}\left(\frac{Upper\ Arm^2 + Fore\ Arm^2 - Shoulder\_Wirst^2}{2 \cdot Upper\ Arm \cdot Fore\ Arm}\right) \quad (2.8)$$

5. **Head Pitch Computation:** The pitch angle of the head concerning the neck joint is calculated as follows: Initially, the vertical distance (y) between the neck and head joints is determined by subtracting the y-coordinate of the head joint from that of the neck joint, and likewise for the depth distance (z). Subsequently, the pitch angle (Pitch) is computed using the arctangent function to ascertain the angle whose tangent corresponds to the ratio of z to y, multiplied by.

This derived angle signifies the pitch angle of the head concerning the neck joint, as demonstrated in Equation **(2.9)**.

$$HeadPitch(\theta) = \tan^{-1}\left(\frac{|Head_z - Neck_z|}{|Head_y - Neck_y|}\right) \quad (2.9)$$

6. **Hip Roll Computation:** The roll angle of the hip joint in relation to the knee joint is calculated as follows: Firstly, the horizontal distance (x) between the hip and knee joints is determined by subtracting the x-coordinate of the knee joint from that of the hip joint, and similarly for the vertical distance (y). Next, the angle (angle) is computed using the arctangent function to determine the angle corresponding to the ratio of x to y. This angle signifies the roll angle of the hip joint concerning the knee joint, as described in Equation (**2.10).**

$$HipRoll(\theta) = \tan^{-1}\left(\frac{|Hip_x - Knee_x|}{|Hip_y - Knee_y|}\right) \quad (2.10)$$

7. **Hip Pitch Computation:** We determine the Pitch angle for the Hip by measuring the distance between the Knee and Ankle along the X-axis and Y-axis, as illustrated in equation **(2.11).**

$$HipPitch(\theta) = -\tan^{-1}\left(\frac{|Knee_x - Ankle_x|}{|Knee_y - Ankle_y|}\right) \quad (2.11)$$

2.4.2 Deep Reinforcement learning

Deep reinforcement learning (DRL) is a type of machine learning technique that enables agents to learn how to make decisions in complex environments by trial and error. It combines deep learning and reinforcement learning, two powerful techniques that have been successful in solving a variety of machine learning problems [12].

Reinforcement learning involves training an agent to make decisions based on feedback received from its environment as shown in Figure 2.7. The agent learns to take actions that maximize a reward signal, which is a scalar value that indicates how well the agent is performing its task.

The goal of the agent is to learn a policy, which is mapping from states to actions that maximizes the expected cumulative reward.

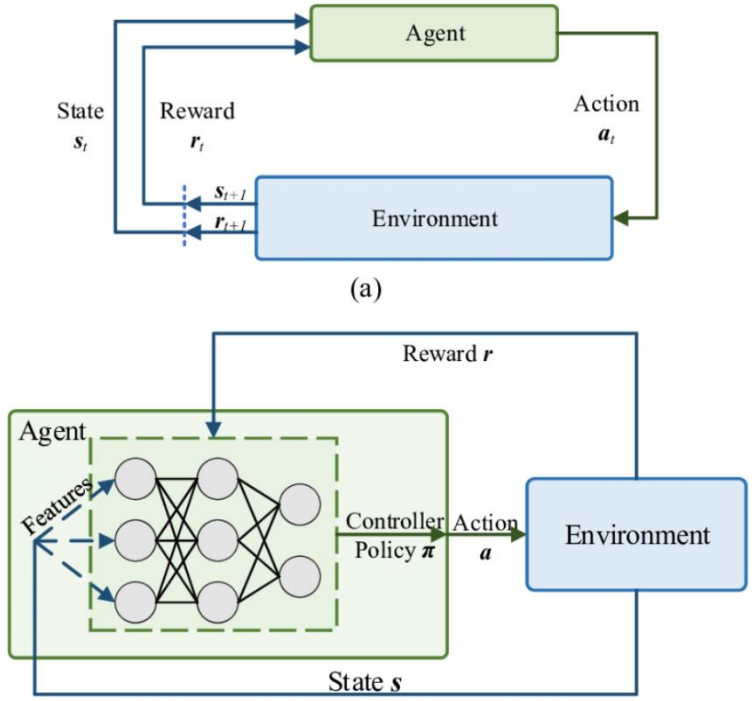


Figure 2-7 Deep Reinforcement Learning

DRL combines two techniques by using deep neural networks to approximate the value function or policy of the agent.

The deep neural networks in Figure 2.8 are used to represent the state-action value function, which is used to select the best action in each state, or the policy function, which is used to directly map states to actions.

This enables agents to learn from high-dimensional inputs, such as images and sensor readings, and make decisions in complex and dynamic environments [17].

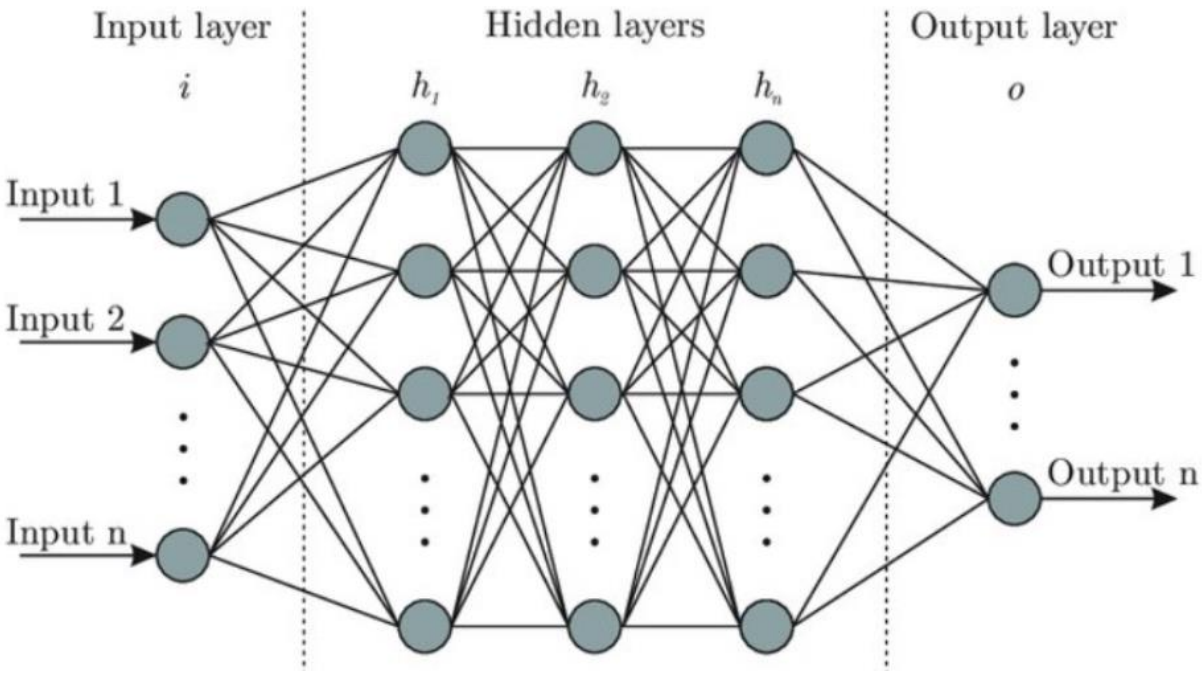


Figure 2-8 Deep Neural network

2.4.3 DDPG

Deep Deterministic Policy Gradient (DDPG) is a reinforcement learning algorithm that combines the ideas of Deep Learning and Policy Gradients with Q-Learning. The goal of the algorithm is to learn a deterministic policy, which maps states to actions directly, as opposed to a stochastic policy, which maps states to probability distributions over actions [14].

DDPG is based on the Actor-Critic method, which uses two neural networks: an Actor network and a Critic network.

The Actor network is responsible for learning the policy, i.e., mapping the state to the action, while the Critic network is responsible for learning the Q-value of the state-action pairs. The Q-value represents the expected discounted future reward of taking a specific action from the current state.

DDPG has been shown to be effective in solving high-dimensional continuous control problems, such as robotic arm control, locomotion, and manipulation tasks.

Algorithm Deep Deterministic Policy Gradient

Input: initial policy parameters θ, Q-function parameters ϕ, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\theta_{\text{targ}} \leftarrow \theta$, $\phi_{\text{targ}} \leftarrow \phi$
repeat
  Observe state s and select action $a = \text{clip}(\mu_\theta(s) + \epsilon, a_{Low}, a_{High})$, where $\epsilon \sim \mathcal{N}$
  Execute a in the environment
  Observe next state s', reward r, and done signal d to indicate whether s' is terminal
  Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
  If s' is terminal, reset environment state.
  if it's time to update **then**
    for however many updates **do**
      Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
      Compute targets

$$y(r, s', d) = r + \gamma(1 - d)Q_{\phi_{\text{targ}}}(s', \mu_{\theta_{\text{targ}}}(s'))$$

      Update Q-function by one step of gradient descent using

$$\nabla_\phi \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_\phi(s, a) - y(r, s', d)\right)^2$$

      Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} Q_\phi(s, \mu_\theta(s))$$

      Update target networks with

$$\phi_{\text{targ}} \leftarrow \rho\phi_{\text{targ}} + (1 - \rho)\phi$$
$$\theta_{\text{targ}} \leftarrow \rho\theta_{\text{targ}} + (1 - \rho)\theta$$

    end for
  end if
until convergence

2.5 Literature Review

Human body tracking has become a critical aspect of many applications, including gaming, sports analysis, and medical rehabilitation.

The use of robotics in such applications has also grown in recent years, with robots being used to assist in geriatric physiotherapy rehabilitation, imitating human actions for autism treatment, and lifting objects. The aim of this literature review is to explore studies that have utilized the Kinect sensor and the NAO robot to track human body movements.

Kinect Controlled NAO Robot for Telerehabilitation: This study showcases the Kinect V2 sensor's capability in marker less vision-based motion tracking, using modified Denavit-Hartenberg convention for kinematic modeling of the human upper arm to mimic therapist movements [1].

Lower-body Control of Humanoid Robot NAO via Kinect: Proposes a joint angle-based control scheme, emphasizing Kinect's potential in capturing and translating lower-body movements to robotic actions [3].

NAO Robot Teleoperation with Human Motion Recognition: Introduces a Kinect-based teleoperation framework using Hidden Markov Model (HMM), highlighting the Kinect v2.0's effectiveness in human motion capture [6].

Robot Sensor System for Supervised Rehabilitation: Demonstrates the effective use of NAO and Kinect sensors in supervising rehabilitation exercises, particularly for shoulder and elbow joints [4].

Motion Recognition using Deep Convolutional Neural Network for Kinect-based NAO Teleoperation: Presents a motion recognition framework enhancing the NAO robot's ability to imitate human behavior, using an adaptive balancing technique and a 7-layer one-dimensional convolutional neural network [5].

Dynamic-goal Deep Reinforcement Learning for Industrial Robot Telemanipulation: The paper by Zhang et al. provides foundational knowledge for robot movement control using Kinect. inspires the use of deep reinforcement learning for human motion data mapping to robot trajectories [2].

These works form a crucial basis for a project leveraging deep reinforcement learning with Nao robots and Kinect sensors for applications in education, entertainment, and healthcare.

2.6 Summery

The "Kinect based teleoperation of a humanoid robot" project is designed to offer an affordable and effective method for capturing and analyzing human motion data. This project is set to enable the real-time replication of human movements with notable accuracy and fidelity. This project is also aimed at enhancing research in human-computer interaction by providing a versatile platform to explore interactions between humans and robots. It promises to encourage the integration of robotics and artificial intelligence in educational settings. Furthermore, this initiative is expected to serve as a cost-effective alternative to more traditional and expensive motion capture technologies. It will also pave the way for the development of new applications in various domains, including physical therapy, sports training, and entertainment.

Chapter 3: System Design

In this chapter, we introduce a system designed to utilize a Nao 6 robot and Kinect v2 sensor for capturing and emulating human movements. The objective is to equip the robot with the ability to learn and replicate intricate human motions, thereby enabling it to execute tasks reliant on precise motor skills and agility. The chapter will go throw into the system's fundamental aspects and constituents, encompassing both hardware and software elements, explain the communication protocols between the robot and sensor, and detailing the mathematical models employed for movement analysis and mapping. In addition to machine learning, we will explore the utilization of mathematical models to interpret and replicate human gestures. Furthermore, we will address the inherent challenges and constraints of the system, along with prospects for enhancing its efficacy and broadening its scope in future endeavors.

3.1 Requirements

1. The system should enable human teleoperation of the Nao robot's motion using the Kinect sensor as an input device.
2. The system should track the user's movements and translate them into corresponding movements for the Nao robot within his possibilities and 25 degrees of freedom.
3. The system should perform complex movements and actions with accuracy in terms of centimeters.
4. The system shall be designed with modular architecture, ensuring that its components can be easily upgraded or replaced. It shall support integration with additional technologies and platforms through standardized interfaces and protocols.
5. The system's performance shall maintain or improve with the addition of new modules or integration with different technologies.

3.2 Hardware design

The Hardware design section provides an overview of the architecture and components of a robotic system designed to enable movement mimicking functionality as shown in Figure 3.1.

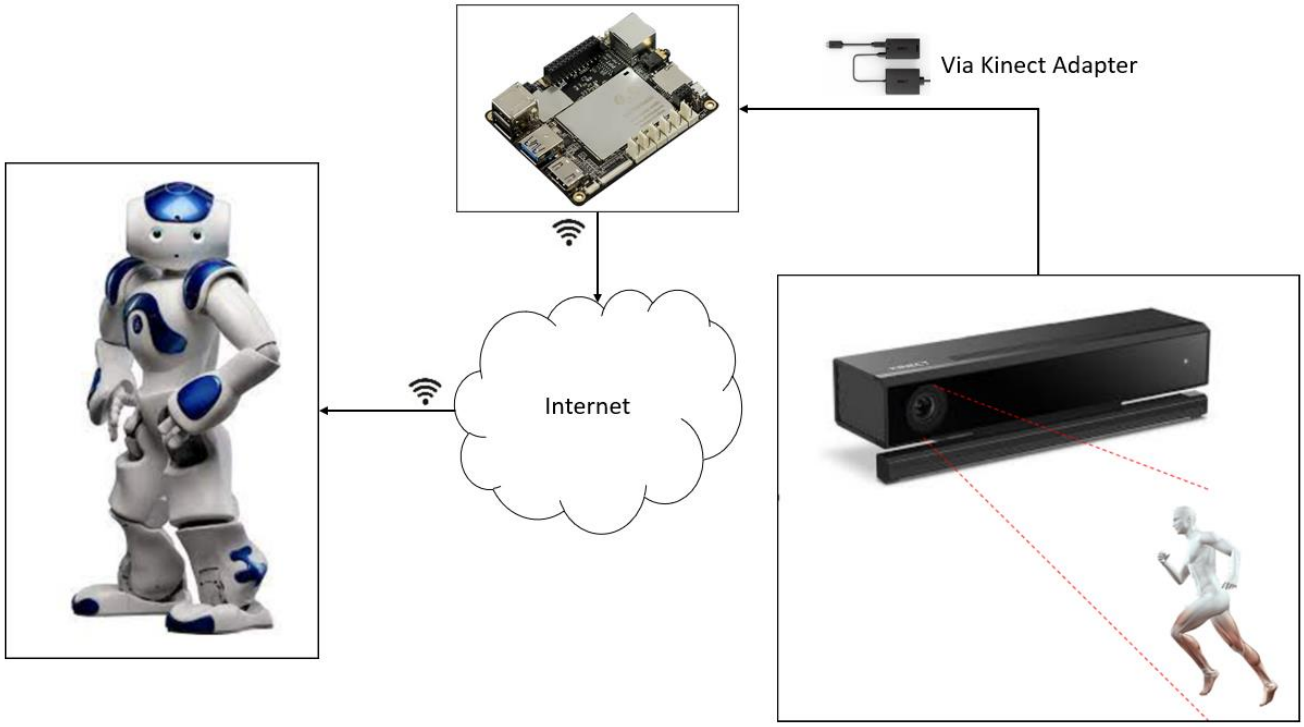


Figure 3-1 System Diagram

3.2.1 System overview

The robotic system is designed to mimic human movements using mathematical models [25]. The system consists of various hardware and software components that enable the capturing and mapping of human movements to robot movements. The system architecture includes sensors to capture human movements, a controller to process the captured data, and actuators to control the robot's movements. The end goal of the system is to enable the robot to replicate human movements accurately in real-time.

3.2.2 Block diagram

This Figure 3.2 shows the main parts of the system and how they interact.

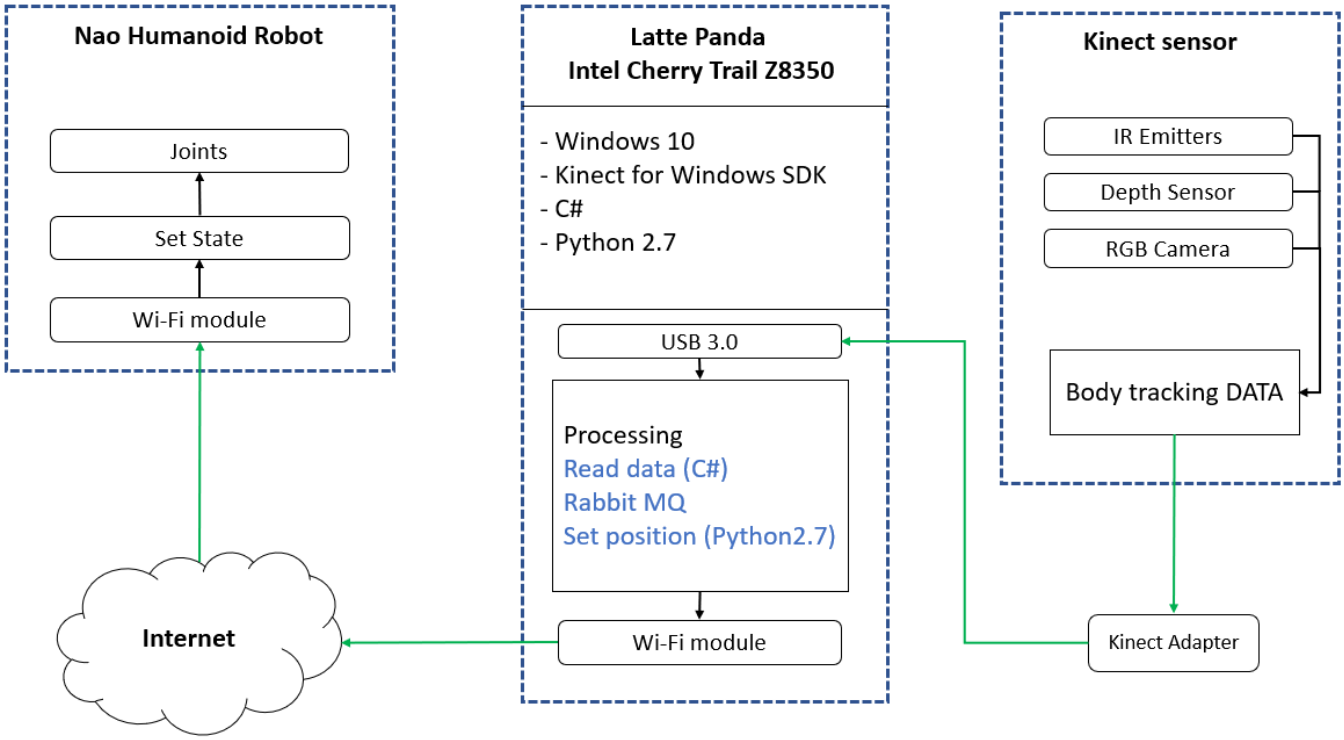


Figure 3-2 System Block Diagram

3.2.3 System Pseudo Codes

Algorithm 2 shown below runs on Microsoft Kinect V2. First, initialize Kinect, provide it a power supply, connect special adapter to it (USB B-type 3.0). Second, from settings select a resolution: 512x424 px, frame rate: 30 Frame Per Second and enable body tracking. Finally, inside loop, capture frames, read body tracking data, then send it to Latte Panda.

ALGORITHM 2: KINECT SENSOR PSEUDO CODE

1. **INITIALIZE** Kinect
2. **CONNECT** Kinect to Kinect adapter
3. **SELECT** resolution, framerate
4. **ENABLE** Body Tracking
5. **WHILE** capturing frames:
6. **CAPTURE** frames (512x424@30fps)
7. **READ** Body tracking data
8. **SEND** Body tracking data to Latte Panda
9. **ENDWHILE**

Algorithm 3 runs on Nao humanoid robot. Initialize Nao and authenticate him to Latte Panda. In loop part, while Latte Panda sends Instructions; receive Instructions from Latte Panda, compile and regenerate executable Instructions for Nao. Finally Execute Instructions for Nao to change his state as like as possible to human state.

ALGORITHM 3: NAO ROBOT PSEUDO CODE

1. **INITIALIZE** Nao robot
2. **AUTHENTICATE** Latte Panda
3. **WHILE** Latte Panda sends Instructions:
4. **RECV** Instructions from Latte Panda using Wi-Fi
5. **SET** position
6. **ENDWHILE**

Algorithm 4 runs on Latte Panda intel cherry trail z8350. At the beginning initialize Latte Panda intel cherry trail z8350, install Windows 10, setup python and C#. Next step, implement mathematical models and connect the Kinect adapter to USB 3.0 port on Latte Panda. Inside loop, receive body tracking data from Kinect via Kinect adapter, execute the mapping algorithm, then convert the next state data set into python Instructions for Nao to change his state and send instructions to Nao using
Wi-Fi.

ALGORITHM 4: LATTE PANDA INTEL CHERRY TRAIL Z8350 PSEUDO CODE

1. **INITIALIZE** Latte Panda
2. **SETUP** windows 10
3. **SETUP** Kinect for Windows SDK
4. **SETUP** Python, C#
5. **IMPORT** numpy, pika, TensorFlow, Naoqi, Microsoft Kinect(C#)
6. **CONNECT** Kinect adapter to USB 3.0 port in Latte Panda
7. **WHILE** data! = NULL:
8. **RECEIVE** Body Tracking DATA from Kinect via Kinect adapter
9. **CONVERT** 3D depth data into motor angles
10. **SEND** Instructions to Nao using Wi-Fi
11. **ENDWHILE**

3.3 System software design

The software system design segment offers insights into the architecture and design principles driving the progression of our project. Here, we emphasize the overarching structure of the software framework and the intricate interplay among its diverse components. By introducing the software system design at the outset, our goal is to furnish readers with a lucid comprehension of the system's holistic organization and the logic guiding our design decisions. This preamble lays the groundwork for ensuing discussions on training, implementation, and supplementary project facets, empowering readers to traverse the technical intricacies armed with a firm understanding of the system's architecture and design fundamentals.

3.3.1 Inverse Kinematics

Definition

Finds the joint angles or positions needed to achieve a specific end-effector pose.

Complexity

Inverse kinematics problems can be complex, especially for robots with multiple joints and degrees of freedom. In this project inverse kinematics for the Nao robot involves handling the complexity of multiple joints and degrees of freedom (25 degree of freedom).

Challenges

- **Singularities**: Consider how to handle singularities and multiple solutions, especially when mapping human movements to the robot's joint space.
- **Real-world constrains**: Account for physical limitations of the Nao robot, such as joint limits and mechanical restrictions.

3.3.2 Dataset

3.3.2.1 Dataset Collection and Specifications

The primary dataset for training the AI model, comprising data from 127 volunteers with varying heights and body sizes, was captured using the Kinect sensor [26]. To enrich the dataset, particularly for leg movement analysis, an additional dataset which focuses on lower-body motion, has been integrated. This dataset provides extensive and specialized leg movement data, enhancing the model's ability to accurately recognize and replicate lower-body actions.

3.3.2.2 Pose Variations and Movements

Each volunteer performed a series of eight predefined positions or poses, including:

1. Y Stretch
2. SOMU Stretch
3. MERMAID Stretch
4. SEATED Stretch
5. TOWEL Stretch
6. WALL Stretch

In addition to these predefined poses, the volunteers also performed a range of movements that fall under the same position category. This inclusion of movements allows for capturing more dynamic and natural pose variations.

3.4.2.3 Data Collection Process

For each volunteer, a total of 240 frames were recorded. Each frame consists of 25 joint camera coordinates in X, Y, and Z dimensions.

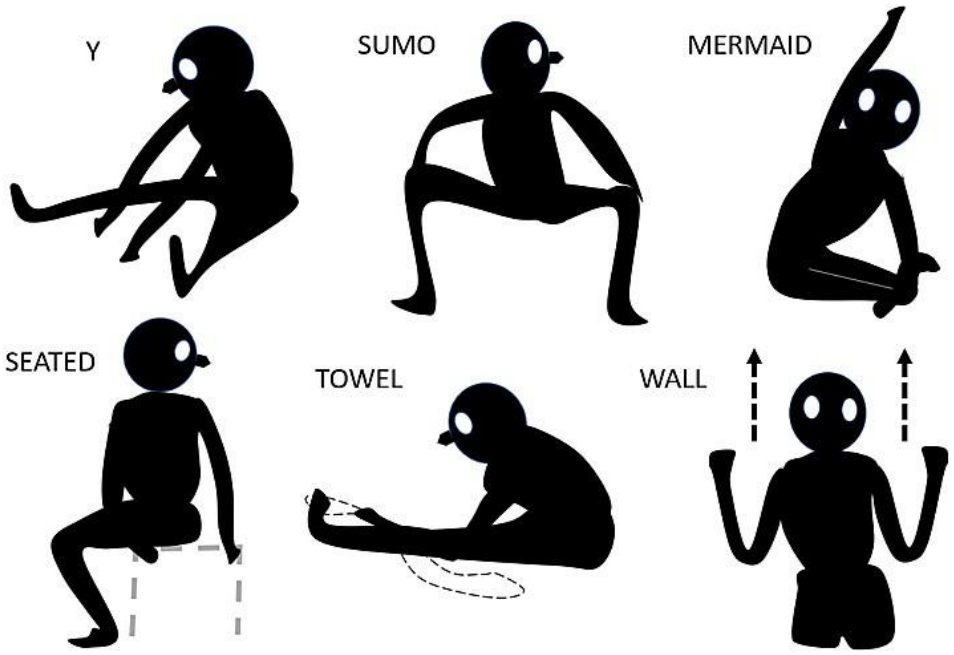


Figure 3-3 Stretches included in Dataset [26]

This joint coordinate provides a comprehensive representation of the human body's pose and Movement. Look at figure 3.3.

3.4.2.4 Training and Testing Data Split

The recorded dataset consists of a total of 30,480 frames. To ensure an effective training process, the dataset was split into two subsets: a training dataset and a testing dataset.

Approximately 72% of the frames, amounting to 21,926 frames, were used as the training data. This larger portion of the dataset is dedicated to training the AI model to learn and imitate the desired poses accurately [26].

The remaining 28% of the frames, approximately 8,554 frames, were reserved for testing the trained model's performance. This testing dataset serves as an independent evaluation set to assess the AI model's ability to generalize and reproduce the poses accurately [26].

3.3.3 Neural Network

We have two networks in this model:

Actor Network: The actor network is responsible for learning and generating the optimal probability for the desired movements [25]. It takes the joint positions and movements as input and outputs the policy that determines the next action to be taken by the Nao robot.

Critic Network: The critic network evaluates the actions taken by the actor network and estimates the corresponding Q-value. The Q-value represents the expected cumulative reward associated with a particular action given the current state of the environment. The critic network helps guide the actor network by providing feedback on the quality of the chosen actions [25].

The neural networks, along with the DDPG algorithm, form an iterative process. During training, the networks interact with the environment (represented by the Nao robot) by receiving sensory input from the Kinect sensor and taking actions based on the learned policy. The resulting feedback and rewards from the environment are used to update the networks' weights and improve their performance over time [25].

By employing this flow of steps and the interaction between the actor and critic networks with the environment, we can train the neural network to learn and generate human-like movements based on the provided dataset and the reinforcement learning framework.

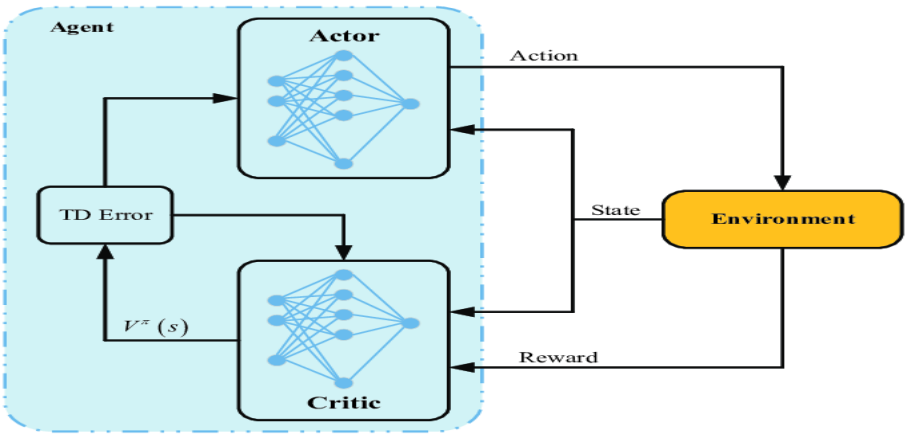


Figure 3-4 Actor and Critic Networks (RNN) [27]

3.3.3.2 Reward function

The reward function in our project plays a crucial role in shaping the learning process of the agent and guiding it towards the desired behavior. It quantifies the performance of the agent based on its actions and the current state of the environment. The goal is to design a reward function that encourages the agent to learn the desired movements and behaviors for the Nao robot.

In our project, we define the reward function to incorporate the concept of distance between the goal position and the current position of the Nao robot's joints in the virtual environment. We calculate the Euclidean distance between the goal position and the current position and use it as a basis for assigning rewards.

The reward function can be defined as in equation (3.1) [15]:

$$reward = e^{-\alpha \cdot distance} \qquad (3.1)$$

In Equation **3.1**, **α** is a scaling factor that determines the influence of the distance on the reward. A higher **α** value results in a steeper reward decay as the distance increases, while a lower **α** value makes the reward decay more gradually. By using the exponential function, we ensure that the reward decreases exponentially as the distance between the goal and the current position increases.

The purpose of this reward function is to encourage the Nao robot to move closer to the goal position, as it will receive higher rewards for smaller distances. This incentivizes the agent to learn movements that bring it closer to the desired goal and helps in training it to mimic human-like movements.

3.4 Summary

Chapter 3 presents a comprehensive design for a system that integrates a Nao 6 robot with a Kinect v2 sensor, aiming to replicate human movements with high precision. This intricate design encapsulates the interplay between sophisticated hardware components and advanced software algorithms, including mathematical models and deep learning.

The system's core is its ability to accurately track and translate human movements into robotic actions, ensuring high fidelity in tasks requiring dexterity and fine motor skills. By detailing the requirements, hardware architecture, and software strategies, including dataset utilization, neural network configurations, and the rationale behind choosing Mathematical Models over traditional kinematics and machine learning, the chapter lays a solid foundation for a future where robots can seamlessly mimic and assist in human-centric tasks, demonstrating a significant leap in the field of robotics and human-robot interaction.

Chapter 4: Implementation

4.1 Introduction

In this chapter, we provide an in-depth exploration of the practical implementation of the system outlined in Chapter 3. The implementation process encompasses the configuration of hardware components, the setup of software systems, and the execution of algorithms essential for enabling the Nao 6 robot to replicate human movements using the Kinect v2 sensor. We will delineate the steps taken, challenges faced, and solutions applied during this crucial implementation phase.

4.2 Hardware Setup

The hardware setup for the system was a meticulous process, involving the assembly and configuration of the Latte Panda, Kinect v2 sensor, and Nao robot. This setup was integral to the successful implementation of the system, ensuring each component functioned correctly and communicated effectively.

Step 1: Connecting Kinect to Latte Panda for Simulation

Interconnection: power supply adapter as show in figure **4-1** Kinect v2 special port in Kinect sensor

USB 3.0 in Latte Panda

Figure 4-1 Kinect Interconnection

Kinect for Windows SDK Installation: The Kinect for Windows SDK and necessary drivers were installed on the PC to facilitate the connection with the Kinect sensor. This setup enabled the capture and processing of motion data for model training.

Step 2: Transitioning from PC to Latte Panda intel cherry trail z8350: the PC was used for training because it had more resources power.

Preparing Latte Panda intel cherry trail z8350 with Windows 10 Home: The Latte Panda intel cherry trail z8350 was set up with Windows 10, optimized for the system's requirements. The operating system was chosen for its compatibility with the Kinect sensor and its ability to handle the computational needs of the system.

Optimizing TensorFlow Model for Latte Panda: The trained TensorFlow model was transferred from the PC to the Latte Panda. This process involved not only moving the model files but also ensuring they were optimized for performance on the Latte Panda's hardware.

Step 3: Setting Up Nao Robot and Establishing Connectivity

Installing Choregraphe on PC/Latte Panda: Choregraphe software was installed for programming and controlling the Nao robot. This software provided a user-friendly interface for creating robot behaviors and ensuring seamless interaction with the system.

Establishing Wi-Fi Connectivity with Nao Robot: The Latte Panda was configured to connect with the Nao robot via Wi-Fi. This wireless setup facilitated real-time control and data exchange, crucial for the system's responsiveness and functionality. To get IP from Nao, hold the button as on the Figure 4.2 below:

In conclusion, the hardware setup process involved a strategic combination of simulation and model training using Webots and TensorFlow on a PC, configuring the Latte Panda with Windows 10 Home, and establishing a Wi-Fi connection with the Nao robot. This careful assembly and setup of the components laid the groundwork for the successful functioning of the system, paving the way for the effective implementation of the project objectives.

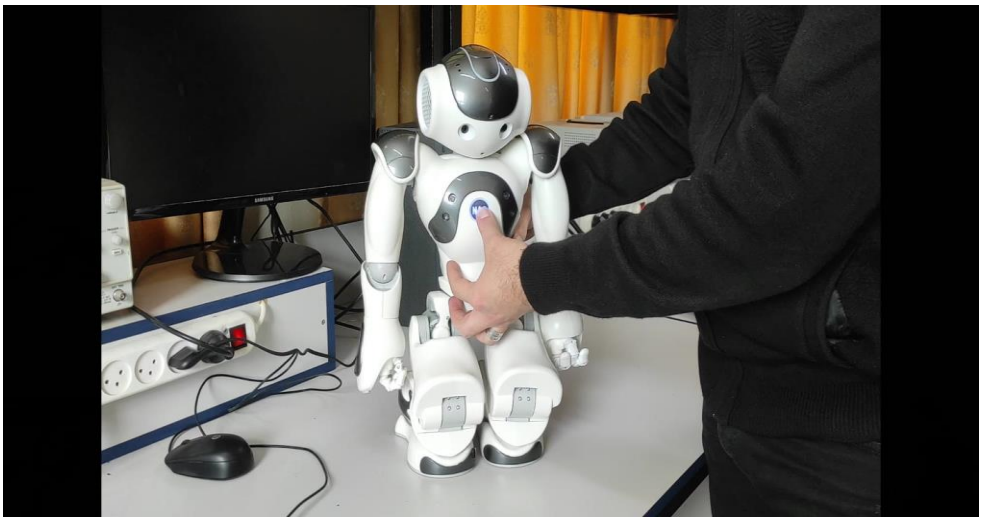


Figure 4-2 How to get Nao's IP

4.3 Software Configuration

The software configuration phase was pivotal in the system's implementation, ensuring seamless integration and functionality of all software components. This phase encompassed the installation and setup of various tools, libraries, and frameworks on the Latte Panda and the PC.

4.3.1 Installing the required software

Operating System Setup: The Latte Panda was configured with Windows 10 Home, chosen for its compatibility with the hardware components and suitability for the system's requirements. The PC was configured with Windows 10 Pro, providing a robust and reliable platform to support the development and simulation environment. Figure 4.3 shows the selection of Windows 10 Home

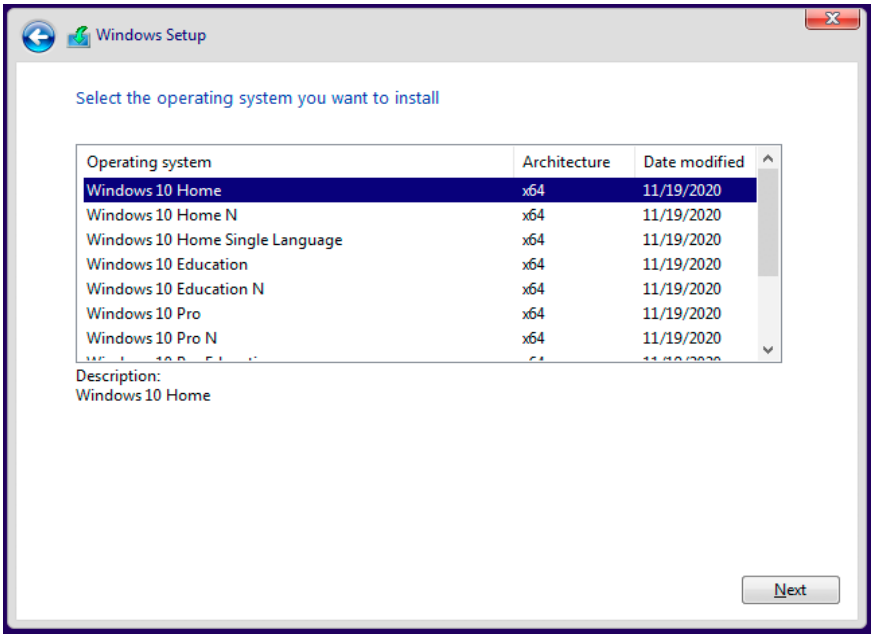


Figure 4-3 Windows 10 Home

TensorFlow Installation: Essential for machine learning tasks, TensorFlow was installed on both the Latte Panda and the PC to handle deep learning models and process data from the Kinect sensor.

Webots Installation: Webots robotics simulation software was installed on the PC to facilitate a virtual testing environment, crucial for developing and refining the model without constant physical testing. Figure 4.4 shows the Webots simulator window configured with Nao robot

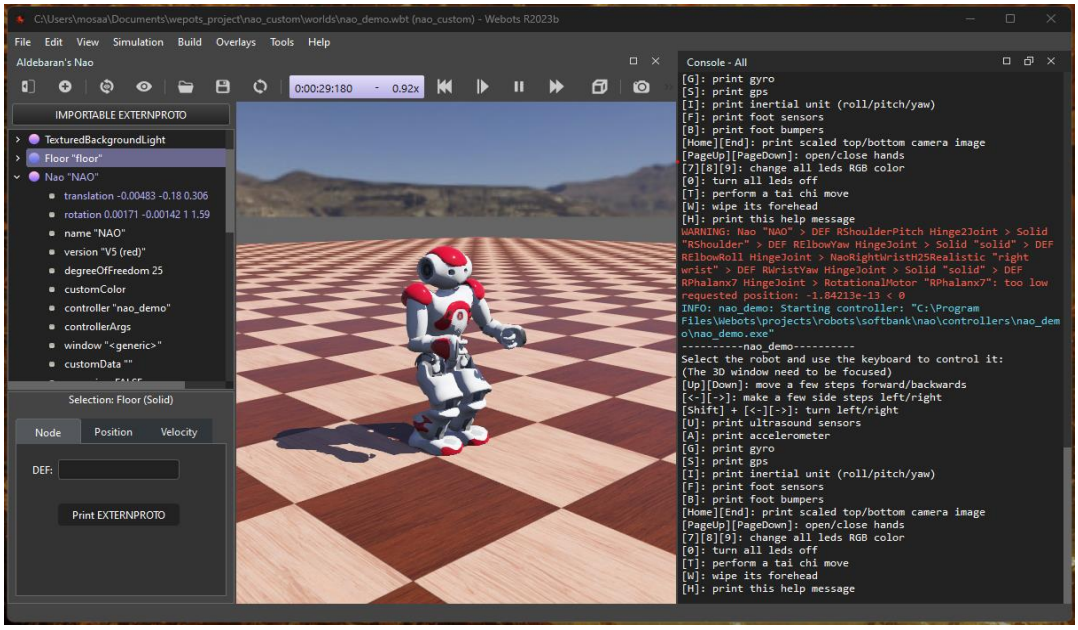


Figure 4-4 Webots Testing

Kinect Software Setup: The Kinect for Windows SDK and necessary drivers were installed on the PC, enabling proper communication and data capture from the Kinect v2 sensor. Figure 4.5 explains that everything works well in Kinect for Windows SDK.

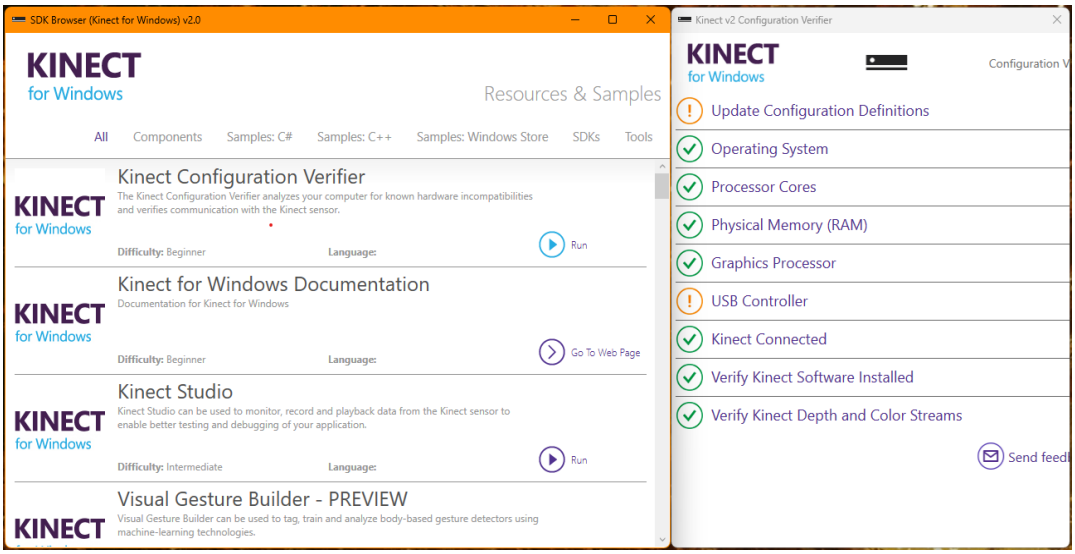


Figure 4-5 Kinect Setup [23]

Rabbit MQ Installation: Rabbit MQ was installed to facilitate data transfer from the Kinect to the Latte Panda. This message broker was essential for reliable and efficient communication within the system. Figure 4.6 shows the installing process of RabbitMQ Server.

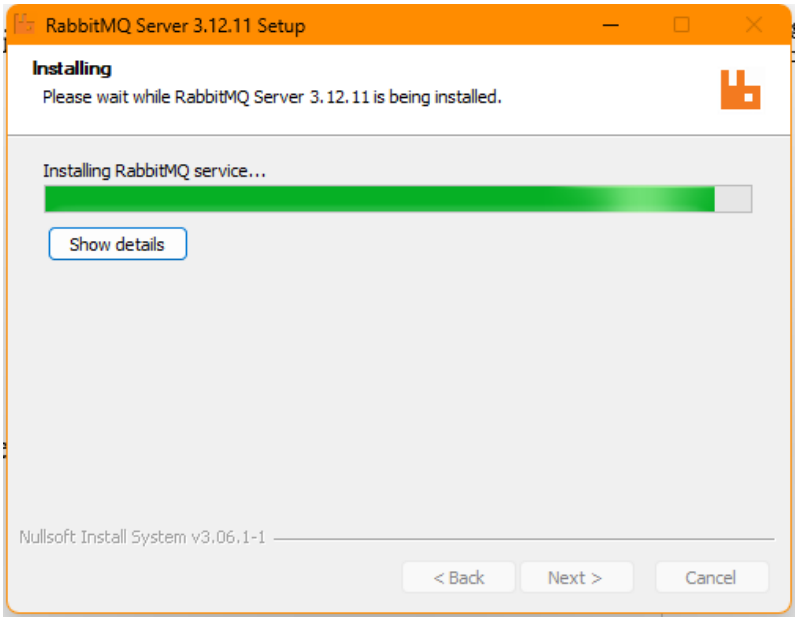


Figure 4-6 RabbitMQ Installation

4.3.2 Setting up the development environment

Configuring Nao with Webots: Nao was set up in the Webots platform to simulate its movements and interactions. This step was crucial for testing and validating the inverse kinematics models and the overall system behavior.

Installing Python 2.7 and C# with Required Libraries:

- **Python 2.7 Installation and Library Setup:** Python was installed as a primary programming language for the system due to its versatility and support for a wide range of libraries. The installation included several specific libraries to enhance its functionality:
    - **naoqi:** it is considered as driver to fully control Nao joints and sensors

- **Pika 1.1.0:** Pika, a RabbitMQ client library, was installed to provide an interface for Python applications to connect and interact with RabbitMQ, enabling efficient message queuing and dispatch.
- **Pandas (as pd):** Pandas, a powerful data analysis and manipulation library, was incorporated for handling and processing structured data. It was imported as 'pd' for ease of use and readability in the code.
- **NumPy:** This library was used for its extensive support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **Math:** is a standard library module that provides mathematical functions and operations. It includes a variety of functions for basic arithmetic operations, as well as more advanced mathematical functions. We need from library trigonometric functions to implement inverse kinematics.
- **TensorFlow Agents (Tf agent):** A specialized TensorFlow library for reinforcement learning, TensorFlow Agents was included to implement and run the Deep Deterministic Policy Gradient (DDPG) algorithm, crucial for training the system in replicating human movements.

- **C# Installation for Kinect Data Capture:** C# was installed to handle the Kinect data capture. C# is well-suited for interacting with the Kinect SDK and efficiently processing the sensor data. The choice of C# for this task was based on its robustness and seamless integration with the Kinect for Windows SDK, ensuring reliable and real-time capture of movement data from the Kinect sensor.

4.3.3 System Software Interconnection Implementation

4.3.3.1 Read Body Tracking Data (C#)

- Kinect for windows SDK provides C# code to read Body Tracking data. We can install it as **.sln** project and debugging it in Visual Studio. Figure 4.7 explain how to install **.sln** Body Basics-WPF

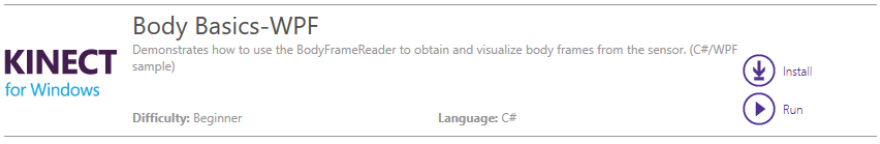


Figure 4-7 Body Basics-WPF [23]

- To deal with Kinect v2, we should import Microsoft.Kinect.
- The code performs 3 main missions:

1. Initialize Kinect

```
// one sensor is currently supported
this.kinectSensor = KinectSensor.GetDefault();
// get the coordinate mapper
this.coordinateMapper = this.kinectSensor.CoordinateMapper;

// get the depth (display) extents
FrameDescription frameDescription = this.kinectSensor.DepthFrameSource.FrameDescription;

// get size of joint space
this.displayWidth = frameDescription.Width;
this.displayHeight = frameDescription.Height;

// open the reader for the body frames
this.bodyFrameReader = this.kinectSensor.BodyFrameSource.OpenReader();

// set IsAvailableChanged event notifier
this.kinectSensor.IsAvailableChanged += this.Sensor_IsAvailableChanged;

// open the sensor
this.kinectSensor.Open();
```

Code Snippet 1

2. Declare bones list

```
// a bone defined as a line between two joints
this.bones = new List<Tuple<JointType, JointType>>();

// Torso
this.bones.Add(new Tuple<JointType, JointType>(JointType.Head, JointType.Neck));
this.bones.Add(new Tuple<JointType, JointType>(JointType.Neck, JointType.SpineShoulder));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder, JointType.SpineMid));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineMid, JointType.SpineBase));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder, JointType.ShoulderRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineShoulder, JointType.ShoulderLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineBase, JointType.HipRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.SpineBase, JointType.HipLeft));

// Right Arm
this.bones.Add(new Tuple<JointType, JointType>(JointType.ShoulderRight, JointType.ElbowRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.ElbowRight, JointType.WristRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.WristRight, JointType.HandRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.HandRight, JointType.HandTipRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.WristRight, JointType.ThumbRight));

// Left Arm
this.bones.Add(new Tuple<JointType, JointType>(JointType.ShoulderLeft, JointType.ElbowLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.ElbowLeft, JointType.WristLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.WristLeft, JointType.HandLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.HandLeft, JointType.HandTipLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.WristLeft, JointType.ThumbLeft));

// Right Leg
this.bones.Add(new Tuple<JointType, JointType>(JointType.HipRight, JointType.KneeRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.KneeRight, JointType.AnkleRight));
this.bones.Add(new Tuple<JointType, JointType>(JointType.AnkleRight, JointType.FootRight));

// Left Leg
this.bones.Add(new Tuple<JointType, JointType>(JointType.HipLeft, JointType.KneeLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.KneeLeft, JointType.AnkleLeft));
this.bones.Add(new Tuple<JointType, JointType>(JointType.AnkleLeft, JointType.FootLeft));
```

Code Snippet 2

3. Read skeleton data (frame by frame)

```
this.bodyFrameReader.FrameArrived +=  this.BodyFrameReader_FrameArrivedAsync;
```

Code Snippet 3

4.3.3.2 Move Body Tracking Data (Rabbit MQ)

Send:

- Body tracking data sent from C# Body Basics project to python.
- Required library: `RabbitMQ.Client`

```
public void sendData(List<JointData> jointDataList)
{
    var factory = new ConnectionFactory() { HostName = "localhost" };
    using (var connection = factory.CreateConnection())
    using (var channel = connection.CreateModel())
    {
        channel.QueueDeclare(queue: "joint_queue",
                             durable: false,
                             exclusive: false,
                             autoDelete: false,
                             arguments: null);

        // Assuming 'body' is an instance of some class containing the joint data

        // Serialize the list of joint data to JSON
        string json = JsonConvert.SerializeObject(jointDataList);
        var bodyBytes = Encoding.UTF8.GetBytes(json);

        // Send the JSON data to RabbitMQ
        channel.BasicPublish(exchange: "",
                             routingKey: "joint_queue",
                             basicProperties: null,
                             body: bodyBytes);
    }
}
```

Code Snippet 4

Receive:

- Body tracking data received form C# to python environment
- Required library: `pika`

```
def recieveData():
    connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))

    channel = connection.channel()

    channel.queue_declare(queue='joint_queue')

    channel.basic_consume(queue='joint_queue',
                          on_message_callback=callback,
                          auto_ack=True)
    channel.start_consuming()
```

Code Snippet 5

4.3.3.3 Set Nao Angles

As we will explain later, Nao angles will be set based on Inverse Kinematics from Kinect body tracking data or a mathematical model. The set position python code is the same for both algorithms.

```
def move(data_dict):

    shoulderRight=data_dict["ShoulderRight"]
    elbowRight=data_dict["ElbowRight"]
    wristRight=data_dict["WristRight"]
    wristLeft=data_dict["WristLeft"]
    elbowLeft=data_dict["ElbowLeft"]
    shoulderLeft=data_dict["ShoulderLeft"]
    head=data_dict['Head']
    neck=data_dict['Neck']
    hipRight=data_dict['HipRight']
    kneeRight=data_dict['KneeRight']
    hipLeft=data_dict['HipLeft']
    kneeLeft=data_dict['KneeLeft']
    ankelRight=data_dict['AnkleRight']
    ankelLeft=data_dict['AnkleLeft']
    listAngles=[]
    listAngles.append(
                    angleRShoulderPitch(shoulderRight[0], shoulderRight[1], shoulderRight[2], elbowRight[0],
                    elbowRight[1],
                                        elbowRight[2]))
    listAngles.append(
        angleRShoulderRoll(shoulderRight[0], shoulderRight[1], shoulderRight[2], elbowRight[0], elbowRight[1],
                        elbowRight[2]))
    listAngles.append(
        angleRElbowRoll(shoulderRight[0], shoulderRight[1], shoulderRight[2], elbowRight[0], elbowRight[1],
                        elbowRight[2], wristRight[0], wristRight[1], wristRight[2]))
    listAngles.append(
        angleRElbowYaw(elbowRight[0], elbowRight[1], elbowRight[2], wristRight[0], wristRight[1],
                    wristRight[2], angleRShoulderPitch(shoulderRight[0], shoulderRight[1], shoulderRight[2],
                    elbowRight[0], elbowRight[1],
                            elbowRight[2])))
    listAngles.append(
        angleLShoulderPitch(shoulderLeft[0], shoulderLeft[1], shoulderLeft[2], elbowLeft[0], elbowLeft[1],
                            elbowLeft[2]))
    listAngles.append(
        angleLShouderRoll(shoulderLeft[0], shoulderLeft[1], shoulderLeft[2], elbowLeft[0], elbowLeft[1],
                        elbowLeft[2]))
    listAngles.append(
        angleLElbowRoll(shoulderLeft[0], shoulderLeft[1], shoulderLeft[2], elbowLeft[0], elbowLeft[1],
                        elbowLeft[2], wristLeft[0], wristLeft[1], wristLeft[2]))
    listAngles.append(
        angleLElbowYaw(elbowLeft[0], elbowLeft[1], elbowLeft[2], wristLeft[0], wristLeft[1],
                    wristLeft[2], angleLShoulderPitch(shoulderLeft[0], shoulderLeft[1], shoulderLeft[2],
                    elbowLeft[0], elbowLeft[1],
                            elbowLeft[2])))
    listAngles.append(angleHeadPitch(neck,head))
    listAngles.append(angleHipRoll(hipRight,kneeRight))
    listAngles.append(angleHipRoll(hipLeft,kneeLeft))
    listAngles.append(angleHipPitch(hipRight,ankelRight))
    listAngles.append(angleHipPitch(hipLeft,ankelLeft))
    global i
    if i%20==0:
        sendAngles(listAngles, RobotIP, RobotPort) # takes userinput
        print(i)
    i+=1
```

Code Snippet 6

4.3.4 Inverse kinematics Implementation

Equations for Nao joints (θ) and Kinect Skeleton (sides of a triangle) were developed and integrated into the system. This involved creating mathematical models and algorithms to accurately translate human movements captured by Kinect into robotic movements.

- Head Pitch

```
def angleHeadPitch(neck,head):
    y=neck[1]-head[1]
    z=neck[2]-head[2]
    Pitch=-2*math.atan(z/y)
    return Pitch
```

Code Snippet 7

- Sholder Pitch

```
def angleLShoulderPitch(x2, y2, z2, x1, y1, z1): #calula
using geometry
    if (y2 < y1):
        angle = math.atan(abs(y2 - y1) / abs(z2 - z1))
        angle = math.degrees(angle)
        angle = -(angle)
        if (angle < -118):
            angle = -117
        return math.radians(angle)
    else:
        angle = math.atan((z2 - z1) / (y2 - y1))
        angle = math.degrees(angle)
        angle = 90 - angle
        return math.radians(angle)
```

Code Snippet 8

- Sholder Roll

```
def angleRShoulderRoll(x2, y2, z2, x1, y1, z1):
using geometry
    if(z2<z1):
        test = z2
        anderetest = z1
        z2=anderetest
        z1=test
    if (z2 - z1 < 0.1):
        z2 = 1.0
        z1 = 0.8
    angle = math.atan((x2 - x1) / (z2 - z1))
    angle = math.degrees(angle)
    return math.radians(angle)
```

Code Snippet 9

- Elbow Yaw

```
def angleRElbowYaw(x2, y2, z2, x1, y1, z1,shoulderpitch): #calulates the ElbowYa
by using geometry
    if(abs(y2-y1)<0.2 and abs(z2-z1) < 0.2 and (x1<x2) ):
        return 0
    elif(abs(x2-x1)<0.1 and abs(z2-z1)<0.1 and (y1>y2)):
        return math.radians(90)
    elif(abs(x2-x1)<0.1 and abs(z2-z1)<0.1 and (shoulderpitch > 50)):
        return math.radians(90)
    elif(abs(y2-y1)<0.1 and abs(z2-z1)<0.1 and (shoulderpitch < 50)):
        return 0
    elif(abs(x2-x1)<0.1 and abs(y2-y1)<0.1 and (shoulderpitch > 50)):
        return math.radians(90)
    else:
        angle = math.atan((z2 - z1) / (y2 - y1))
        angle = math.degrees(angle)
        angle = - angle + (shoulderpitch)
        angle = - angle
        return math.radians(angle)

def angleLElbowYaw(x2, y2, z2, x1, y1, z1, shoulderpitch): #calulates the ElbowY
by using geometry
    if(abs(y2-y1)<0.2 and abs(z2-z1) < 0.2 and (x1>x2) ):
        return 0
    elif(abs(x2-x1)<0.1 and abs(z2-z1)<0.1 and (y1>y2)):
        return math.radians(-90)
    elif(abs(x2-x1)<0.1 and abs(z2-z1)<0.1 and (shoulderpitch > 50)):
        return math.radians(-90)
    elif(abs(y2-y1)<0.1 and abs(z2-z1)<0.1 and (shoulderpitch > 50)):
        return 0
    elif(abs(x2-x1)<0.1 and abs(y2-y1)<0.1 and (shoulderpitch > 50)):
        return math.radians(-90)
    else:
        angle = math.atan((z2 - z1) / (y2 - y1))
        angle = math.degrees(angle)
        angle = - angle + (shoulderpitch)
        angle = - angle
        return math.radians(angle)
```

Code Snippet 10

- Elbow Roll

```
def angleRElbowRoll(x3, y3, z3, x2, y2, z2, x1, y1, z1): #calulates the ElbowRoll value for the Right elbow
by using geometry
    a1=(x3-x2)**2+(y3-y2)**2 + (z3-z2)**2
    lineA= a1 ** 0.5                    # calculates length of line between 2 3D coordinates
    b1=(x2-x1)**2+(y2-y1)**2 + (z2-z1)**2
    lineB= b1 ** 0.5                    # calculates length of line between 2 3D coordinates
    c1=(x1-x3)**2+(y1-y3)**2 + (z1-z3)**2
    lineC= c1 ** 0.5                    # calculates length of line between 2 3D coordinates
    cosB = (pow(lineA, 2) + pow(lineB,2) - pow(lineC,2))/(2*lineA*lineB)
    acosB = math.acos(cosB)
    angle = math.degrees(acosB)
    angle = 180 - angle
    return math.radians(angle)

def angleLElbowRoll(x3, y3, z3, x2, y2, z2, x1, y1, z1): #calulates the ElbowRoll value for the Left elbow by
using geometry
    a1=(x3-x2)**2+(y3-y2)**2 + (z3-z2)**2
    lineA= a1 ** 0.5                    # calculates length of line between 2 3D coordinates
    b1=(x2-x1)**2+(y2-y1)**2 + (z2-z1)**2
    lineB= b1 ** 0.5                    # calculates length of line between 2 3D coordinates
    c1=(x1-x3)**2+(y1-y3)**2 + (z1-z3)**2
    lineC= c1 ** 0.5                    # calculates length of line between 2 3D coordinates
    cosB = (pow(lineA, 2) + pow(lineB,2) - pow(lineC,2))/(2*lineA*lineB)
    acosB = math.acos(cosB)
    angle = math.degrees(acosB)
    angle = -180+ angle
    return math.radians(angle)
```

Code Snippet 11

- Hip Roll and Hip Pitch

```
def angleHipRoll(hip,knee):
    x=hip[0]-knee[0]
    y=hip[1]-knee[1]
    angle=math.atan(x/y)
    return angle

def angleHipPitch(knee,ankel):
    x=knee[0]-ankel[0]
    y=knee[1]-ankel[1]
    angle=math.atan(x/y)
    return -angle
```

Code Snippet 12

4.3.5 DDPG Implementation and training

4.3.5.1 Description

In our training approach, we have selected inverse kinematics to serve as the reward function within the DDPG algorithm (see Figure 4.8). The dataset itself forms the environment for the learning process. Initially, the agent critic commences with a zero reward and undergoes updates grounded in the imitation of inverse kinematics. As training progresses, the agent learns to optimize its actions based on the provided inverse kinematics rewards, effectively refining its performance in the given environment.

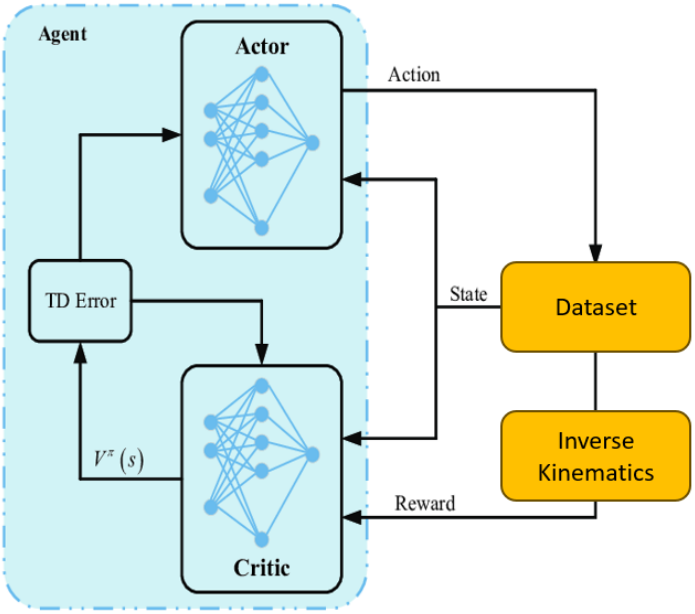


Figure 4-8 Training Approach

4.3.5.2 Kinect v2 Dataset Selection for Training

For training our system, we selected the "3D Kinect Total Body Database for Back Stretches,"[26] a specialized dataset captured using the Kinect V2 sensor. This dataset's specifications are particularly suited for our project requirements [26]:

- **Data Collection:** Captured as a set of X, Y, Z coordinates at 60 fps during six different yoga-inspired back stretches, providing a detailed representation of human back movements.
- **Dataset Composition:** Contains 541 files, each detailing position and velocity for 25 body joints, including the head, neck, spine, shoulders, hips, wrists, knees, and feet.
- **Standardization and Consistency:** The Kinect was positioned at a height of 2 ft and 3 in, with subjects 6.5 ft away from the camera, ensuring consistent data capture. Each participant completed 10 repetitions of each stretch, providing a rich set of repeated movement data.
- **Participant Demographics:** Data was collected from 9 adults aged 18-21, including 4 females and 5 males, offering a diverse range of motion data.
- **Pre-processing:** Velocity data was calculated using a discrete derivative equation, which was applied to all body parts and axes individually, enhancing the data's utility for motion analysis.

4.3.5.3 DDPG Implementation

- Required libraries: TensorFlow, tf_agents, math, pandas, numpy
- Procedures

1. Initial procedure

```
def __init__(self,supervisor):
    self.nao=NaoRobot(supervisor)
    self.i=0
    self._episode_steps = 10
    self._step_counter=0
    # Initialize target joint positions
    self.target_positions=[]
    for item  in Dataset[self.i].items():
        for elment in item[1]:
          self.target_positions.append(elment)
    self.len_target_position=len(self.target_positions)
    self.current_positions = np.zeros(self.nao.num_joints)
    self._episode_steps = 0
    self.numOfstate=self.nao.num_joints+self.len_target_position
    self._action_spec = array_spec.BoundedArraySpec(
        shape=(self.nao.num_joints,),
        dtype=np.float32,
        minimum=-np.pi ,
        maximum=np.pi,
        name='action'
    )
    self._observation_spec = array_spec.BoundedArraySpec(
        shape=(self.numOfstate,),  # Combined current and target joint positions
        dtype=np.float32,
        minimum=[-np.pi] * self.numOfstate,
        maximum=[np.pi] * self.numOfstate,
        name='observation'
    )
    self._state=np.array(self.target_positions+self._get_current_positions(),dtype=np.float32)
    self._episode_ended = False
    self._reward=0.0
```

Code Snippet 13

2. Reset Procedure

```
def _reset(self):
    self.nao.supervisor.simulationReset()
    self._state=np.array(self.target_positions+self._get_current_positions(),dtype=np.float32)
    self._episode_steps = 0
    self._step_counter=0
    self._episode_ended=False
    return ts.restart(self._state)
```

Code Snippet 14

3. Update target position Procedure

```
def _update_target_positions(self):
    i=self.i+1
    self.i+=1
    self.target_positions=[]
    for item  in Dataset[self.i].items():
        for elment in item[1]:
          self.target_positions.append(elment)
    self._state=np.array(self.target_positions+self._get_current_positions(),dtype=np.float32)
```

Code Snippet 15

4. Step Procedure

```
def _step(self, action):
    self._step_counter = self._step_counter + 1
    Move(Dataset[self.i],self.nao.joint_devices)
    self.nao.supervisor.simulationReset()
    self.nao.supervisor.step(self.nao.timeStep)
    right=self._get_current_positions()
    self._apply_action(action)
    result=self._get_current_positions()
    self._state=np.array(self.target_positions+self._get_current_positions(),dtype=np.float32)
    if self._step_counter >= self._episode_steps:
        self._episode_ended = True
        self._update_target_positions()
    reward=self._calculate_reward(result,right)
    if self._episode_ended:
        return ts.termination(self._state, reward)
    else:
        return ts.transition(self._state, reward, discount=1.0)
```

Code Snippet 16

5. Calculate Reward Procedure

```
def _calculate_reward(self,result,right):
    print(self.i)
    a=self.nao.senesors['LFsr'].getValues()
    b=self.nao.senesors['RFsr'].getValues()
    a=a[2]
    b=b[2]
    if a<2 and b<2:
        self.nao.supervisor.simulationReset()
        self._episode_ended=True
        return -10-np.sum(np.abs(np.array(right)-np.array(result)))
    else :
        return -np.sum(np.abs(np.array(right)-np.array(result)))
```

Code Snippet 17

4.4 Challenges

The implementation of the system presented several significant challenges that needed to be addressed to ensure its successful operation. These challenges were critical in understanding the limitations and capabilities of the system and in guiding future improvements.

1. Balancing

One of the primary challenges faced was maintaining the balance of the Nao robot while replicating human movements. Unlike humans, who have a highly developed sense of balance, robots like Nao require complex algorithms to remain stable, especially when performing dynamic movements or standing on one leg.

2. Degree of Freedom between Nao and Human

Challenge: The discrepancy in the degree of freedom between humans and the Nao robot presented a significant challenge. Humans have a vast range of motion compared to the Nao robot, which has limited degree of freedom in its joints.

Approach: To overcome this, we developed algorithms that could translate the complex human movements into simpler motions that the Nao robot could perform. This translation process involved determining the most critical aspects of the human movement and simplifying them to fit the robot's capabilities without losing the essence of the action.

Chapter 5: Testing and Results

5.1 Introduction

Chapter 5 is dedicated to the testing phase of the system, focusing on evaluating each hardware component and the overall interconnection within the system. This stage is crucial to ensure that all parts function correctly and cohesively.

5.2 Hardware Testing

a. Kinect v2

Testing the Kinect v2 involved verifying its ability to accurately capture motion data. This included assessing the sensor's responsiveness, precision in tracking movements, and consistency in different lighting conditions. Sometimes it appears some glitches in reading body data because the environment of SDK itself as shown in Figure 5.1.

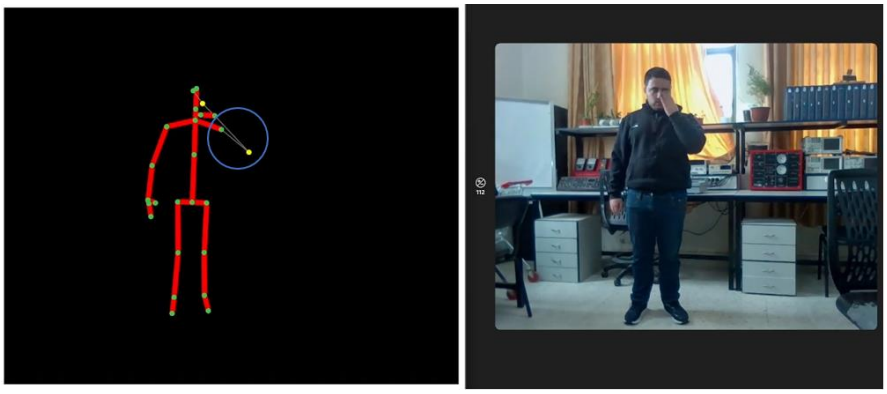


Figure 5-1 Kinect v2 glitch (example)

b. Nao Robot

Tests for the Nao robot focused on its movement replication accuracy, joint articulation, balance, and response time to commands received from the Latte Panda.

c. Latte Panda

The Latte Panda was tested for its computing performance, especially its ability to process and relay motion data to the Nao robot efficiently. Stability and reliability of the operating system and installed software were also evaluated.

d. System Interconnection

The final part of the testing phase involved evaluating the system's overall interconnection. This included testing the communication flow between the Kinect v2, Latte Panda, and Nao robot, ensuring seamless data transfer and synchronization across the system.

5.3 Software Testing

1. Testing of inverse kinematics involved validating the equations and algorithms used for translating human movement into robotic actions. This included simulations to ensure the movements were accurately mirrored by the robot.

2. Webots was tested for its simulation capabilities, ensuring accurate rendering and physics simulation of the robot's movements and environment interactions.

3. Choregraphe software was tested for its functionality in programming and controlling the Nao robot. We used Choregraphe to test the final software (inverse kinematics and trained model), because it simulates the real environment of Nao. This included validating the user interface, motion creation tools, and the ability to upload scripts to the robot. Figure 5.2 show that Choregraphe works well.

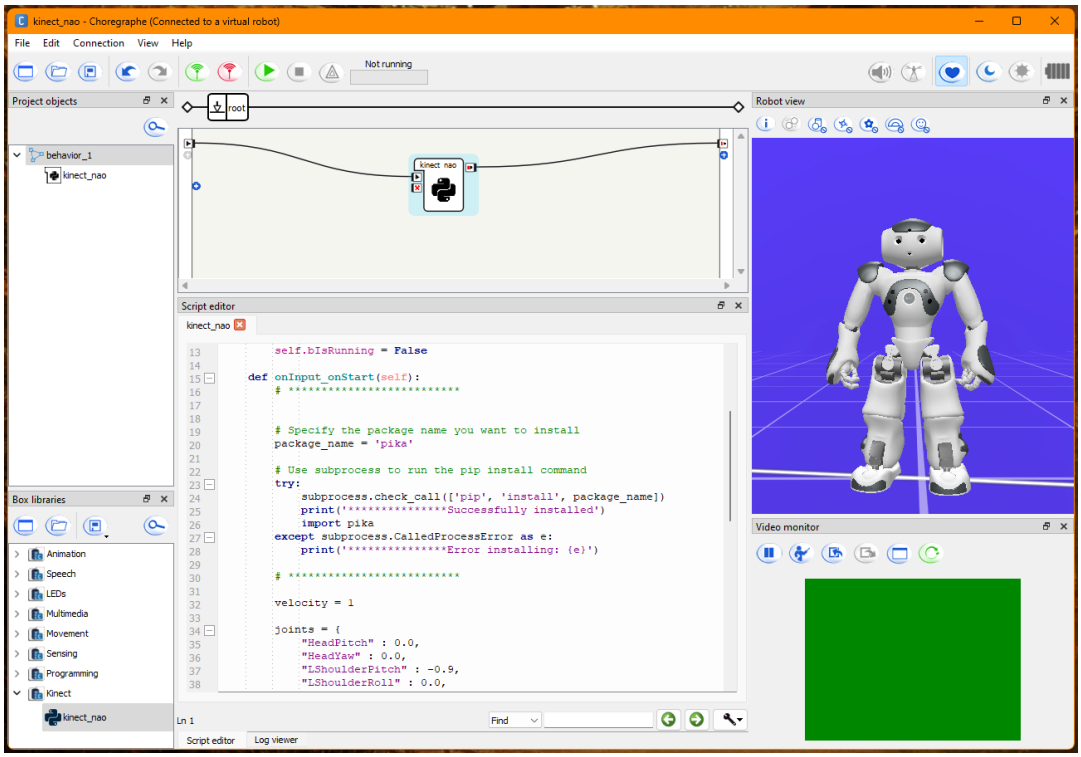


Figure 5-2 Choregraphe Testing

4. Libraries in C# and Python were tested for their roles in data capture and processing. For C#, the focus was on Kinect data acquisition, while for Python, the emphasis was on data analysis, machine learning tasks, and communication with other system components.

5.4 Results

5.4.1 Model training results

Initially, our project aimed to leverage machine learning techniques, including the Deep Deterministic Policy Gradient (DDPG) algorithm, in conjunction with our implementation of the Inverse Kinematics approach. This dual approach was designed to achieve two primary objectives: ensuring effective balancing and addressing challenges associated with the Nao robot's degrees of freedom.

However, despite our efforts to fully implement and train the DDPG algorithm, we encountered significant difficulties. The utilization of DDPG resulted in suboptimal performance, characterized by erratic and random movements. Despite extensive training with large datasets, the algorithm struggled to produce coherent and accurate motion plans.

We believe that the problem is implementing of the reward function, represented by inverse kinematics imitation and the pressure of the Nao feet to ensure balancing. These two criteria affected each other negatively, which led to production trained model characterized by erratic and random movements.

In light of these challenges and the unsatisfactory results obtained with the DDPG algorithm, we made the strategic decision to pivot our approach. We opted to prioritize the development of invers kinematics mathematical model to address the complexities inherent in our teleoperation system. While acknowledging the limitations of a mathematical model, including its inability to capture all nuances of human movement, we concluded that it offered a more viable solution than persisting with the problematic outcomes of the DDPG approach.

By focusing on the development of a robust inverse kinematics mathematical model, we aim to overcome the shortcomings encountered with the DDPG algorithm. Despite its inherent limitations, we believe that the mathematical model provides a more stable and predictable framework for controlling the Nao robot's movements, ultimately offering superior performance compared to the unpredictable behavior observed with DDPG.

5.4.2 Inverse Kinematics results

We have achieved favorable outcomes in Inverse Kinematics, enabling real-time and precise imitation of human head, arms, and legs movements:

- Response time: 200 – 500 milliseconds
- Accuracy: in terms of centimeters, we did not measure exactly.

We have applied many tests for Inverse Kinematics. Each test performed 5 times, we chose two main types of tests, first Test performed on Nao chain, the second test is on poses of the Kinect dataset that we used.

Testing Characteristics:

- Limitations:
  - Angle: Nao motor out of range
  - Mapping: Mismatch between Kinect skeleton and Nao chains
  - Dimension: Bone length ratio in Kinect not necessarily be the same as Nao body
  - Balancing: Distribution of the center of gravity in the feet of Nao
  - Complex: some movements depend on each other, in some cases Inverse Kinematics approach cannot handle it especially when the movement composed of more than one chain

- Imitation Accuracy Standards:
  - The final form of a movement
  - Imitation path (From the beginning of imitation until the end)
  - Balancing

➢ **Chains imitation testing results:**

Chain	Move	Limitations	Imitation Accuracy /10	Extra
Arm	Up		8.5	The elbow is bent more than expected.
	Down		9.5	This is default position of arm.
Left Arm	Left		9.5	
	Right	Angle	7	Nao LsholderPith angle.
Right Arm	Left	Angle	7	Nao RsholderPith angle.
	Right		9.5	

Head	Up		9	
	Down		9	
	Left	Mapping	0	No motor (Head Roll).
	Right	Mapping	0	No motor (Head Roll).
Leg	Forward	Balancing	8	Nao falls down.
	Backward	Balancing	8	Leg does not move because of pressure.
Hip	Forward	Balancing	4	Hole body lying forward.
	Backward	Balancing	4	Body falls down backwards.
Left Hip	Left	Mapping Balancing	6	Both feet stuck on the ground.
	Right		9	Motion not smooth.
Right Hip	Left		9	Motion not smooth.
	Right	Mapping Balancing	6	Both feet stuck on the ground.

Table 5-1 Chains imitation testing results

- **Kinect Dataset Poses testing results:**

Pose	Limitations	Imitation Accuracy /10	Extra
Y	Balancing	8	The elbow is bent more than supposed.
SUMO	Dimension Balancing	8.5	The hand was not placed on the knee, also sometimes the robot falls on the ground.
MERMAID	Complex	6	Nao robot rise his hand efficiently. The problem is in the leg and hip chains, this dual movement causes overlap in overall imitation.
SEATED	Dimension Complex	5	This is the most complex pose. The problem is that the Nao dimensions various from Kinect skeleton dimensions. For example, if a human chair height is 50cm, the corresponding chair for Nao is 15cm
TOWEL	Angle	8.5	Nao HipPitch angle
WALL		9.5	This pose imitated perfectly

Table 5-2 Kinect Dataset Poses testing results

There are some tests form results as shown in figure 5.3:

Figure 5-3 Project Testing

Chapter 6: Conclusion and future work

6.1 Conclusion

This project successfully developed and implemented a system integrating a Nao 6 robot, Kinect v2 sensor, and Latte Panda to mimic human movements. The hardware components, including the Kinect v2 sensor, Nao robot, and Latte Panda, were meticulously assembled and rigorously tested to ensure optimal performance and synchronization. In the software domain, key components such as inverse kinematics, Webots simulation, Choregraphe programming, and various C# and Python libraries were methodically tested and refined.

The system's ability to accurately replicate human movements showcases the potential of robotics in various applications. The successful integration of the hardware and software components demonstrates the feasibility and effectiveness of such complex robotic systems.

6.2 Future Work

Looking forward, there are several avenues for future development and enhancement:

Advanced Learning Algorithms: Exploring machine learning algorithms and Deep learning could further improve the system's accuracy and efficiency in mimicking human movements.

Developing Deep learning approach: Updating the reward function in DDPG algorithm, in a way that ensures achieving balancing and accurate imitation between human and humanoid

Broader Movement Repertoire: Expanding the system to replicate a wider range of human movements, including more nuanced and subtle gestures, would increase its applicability.

Real-World Applications: Applying the system in real-world scenarios, such as in rehabilitation therapy or educational settings, would provide valuable insights into its practical utility and areas for improvement.

User Interaction and Feedback: Integrating user feedback mechanisms could make the system more interactive and user-friendly, adapting to individual user’s needs and preferences.

In conclusion, this project represents a significant step of robotics and human-robot interaction. The lessons learned and the foundation established here pave the way for further innovations and applications in this exciting and rapidly evolving field.

References

1. Assad-Uz-Zaman, M., Islam, M., Rahman, M., Wang, Y. & McGonigle, E. (2021). Kinect Controlled NAO Robot for Telerehabilitation. *Journal of Intelligent Systems*, *30*(1), 224-239. https://doi.org/10.1515/jisys-2019-0126

2. Zhang, Y., Liu, Y., & Wang, J. (2019). Dynamic-goal Deep Reinforcement Learning for Industrial Robot Telemanipulation.

3. Chen, J., Wang, G., Hu, X. et al. Lower-body control of humanoid robot NAO via Kinect. Multimed Tools Appl 77, 10883–10898 (2018). https://doi.org/10.1007/s11042-017-5332-3

4. Assad Uz Zaman, M., Islam, M.R., Rahman, M.H. et al. Robot sensor system for supervised rehabilitation with real-time feedback. Multimed Tools Appl 79, 26643–26660 (2020). https://doi.org/10.1007/s11042-020-09266-x

5. Balmik, A., Paikaray, A., Jha, M., & Nandy, A. (2022). Motion recognition using deep convolutional neural network for Kinect-based NAO teleoperation. Robotica, 40(9), 3233-3253. https://www.cambridge.org/core/journals/robotica/article/abs/motion-recognition-using-deep-convolutional-neural-network-for-kinectbased-nao-teleoperation/AA337EC3B6AE611A13765DF1D4D1ECA5.

6. Balmik, A., Jha, M. & Nandy, A. NAO Robot Teleoperation with Human Motion Recognition. Arab J Sci Eng 47, 1137–1146 (2022). https://doi.org/10.1007/s13369-021-06051-2

7. Hansard, M., Lee, S., Choi, O., & Houraud, R. (2012). Time-of-flight cameras: Principles, methods and applications. Springer. Retrieved from https://books.google.ps/books?id=PiF4narL1Z0C.

8. Cruz-Ramírez, S.R., García-Martínez, M. & Olais-Govea, J.M. NAO robots as context to teach numerical methods. Int J Interact Des Manuf 16, 1337–1356 (2022). https://doi.org/10.1007/s12008-022-01065-y

9. Wikipedia, "TensorFlow," [Online]. Available: https://en.wikipedia.org/wiki/TensorFlow. [Accessed 2022].

10. Cyberbotics.(2023).Webots:RobotSimulator.Retrieved from https://www.cyberbotics.com.

11. MIT Sloan. (2021). Machine learning, explained. Retrieved from https://mitsloan.mit.edu/ideas-made-to-matter/machine-learning-explained

12. Sewak, M. (2019). Introduction to Reinforcement Learning. In: Deep Reinforcement Learning. Springer, Singapore. https://doi.org/10.1007/978-981-13-8285-7_1

13. Towards Data Science. (2018). Training Deep Neural Networks. Retrieved from https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964.

14. Sumiea, E. H., AbdulKadir, S. J., Al-Selwi, S. M., Alqushaibi, A., Ragab, M. G., Fati, S. M., & Alhussian, H. S. (2023). Deep Deterministic Policy Gradient Algorithm: A Systematic Review.

15. Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Proceedings of the International Conference on Learning Representations (ICLR). Google DeepMind, London, UK.

16. Sutton, R. S., & Barto, A. G. (2005). [The actor-critic architecture.]. Reinforcement Learning: An Introduction Retrieved from http://incompleteideas.net/book/first/ebook/node66.html#fig:actor-critic.

17. Vaughan, H. (2023). Develop your first AI agent - Deep Q Learning. Towards Data Science. Retrieved from https://towardsdatascience.com/develop-your-first-ai-agent-deep-q-learning-375876ee2472.

18. MathWorks. (2023) .DDPG Agents. Retrieved from https://in.mathworks.com/help/reinforcement-learning/ug/ddpg-agents.html.

19. Aldebaran. (2022). NAO. Retrieved from https://www.aldebaran.com/en/nao.

20. Wevolver. (2023) . NAO Robot Specifications. Retrieved from https://www.wevolver.com/specs/nao.robot

21. Packt Publishing. (n.d.). Components of Kinect for Windows. In Kinect for Windows SDK Programming Guide. Retrieved from https://subscription.packtpub.com/book/programming/9781849692380/1/ch01lvl1sec08/components-of-kinect-for-windows.

22. Soltaninejad, Sara & Cheng, Irene & Basu, A.. (2019). Kin-FOG: Automatic Simulated Freezing of Gait (FOG) Assessment System for Parkinson's Disease. Sensors. 19. 2416.

23. Microsoft. (2022). Kinect for Windows. Retrieved from https://learn.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows.

24. Latte Panda (2018). Latte Panda V1 Specifications. Retrieved from https://www.Latte Panda.com/Latte Panda-v1

25. Hua, J., Zeng, L., Li, G., & Ju, Z. (2021). Learning for a Robot: Deep Reinforcement Learning, Imitation Learning, Transfer Learning. Sensors, 21(4), 1278. https://doi.org/10.3390/s21041278

26. Kaggle. (2020). 3D Kinect Total Body Database for Back Stretches. Retrieved from https://www.kaggle.com/datasets/dasmehdixtr/3d-kinect-total-body-database-for-back-stretches?resource=download&fbclid=IwAR0JeW4wkyeMfBQbXyVOe8Cl4eWXns4NMwmmjtHxuASBFOjKQ9p8DdxUFwQ.

27. Giang, Hoang & Hoan, Tran & Thanh, Pham & Koo, Insoo. (2020). Hybrid NOMA/OMA-Based Dynamic Power Allocation Scheme Using Deep Reinforcement Learning in 5G Networks. Applied Sciences.