Ryerson University
Department of Electrical, Computer & Biomedical Engineering
Faculty of Engineering & Architectural Science

BEAMstiquito ("The Insect") Walking Robot

by

Prajay Patel, Apurva Patel, John (Can) Turk, and Usman Mazhar

Computer/Electrical Engineering Capstone Design Project
Toronto Metropolitan University, 2024

Acknowledgements

We would like to express our gratitude to those whose support and guidance were essential in the completion of our capstone research phase:

Dr. Kassam: He helped guide the path of our research and efforts if they ever became unreasonable and helped guide our project in a general direction.

Freenove: We thank Freenove for the use of their robot kit and the associated codebase. The advanced controls, including PWM control of servos and servo stabilization, were necessary to complement our project and its goals. Freenove also responded to our email inquiries which helped us confirm that the robot was open to our modification.

We are grateful for the support received from both our professor and Freenove, who enriched our learning experience and contributed to the success of this capstone research phase.

Certification of Authorship

Our group: Prajay, Apurva, Usman, and Can, write this statement to certify that:

1. The work presented in our report "BEAMstiquito (“The Insect”) Walking Robot" is entirely our original work, except wherever otherwise indicated.

2. Any contributions from co-authors or collaborators have been appropriately acknowledged and credited.

3. The content of this report has yet to be submitted for academic credit elsewhere.

We understand that any intentional misrepresentation or failure to disclose academic misconduct is a violation of policy.

Prajay Patel (P.P), Apurva Patel (A.P), Usman Mazhar (U.M), and John(Can) Turk (C.T)
April 12th, 2024

Table of Contents

1. Abstract

The BEAMstiquito project demonstrates the innovative integration of Biology-Electronics-Aesthetics-Mechanics (BEAM) principles in designing and implementing an autonomous hexapod robot. This project leverages cutting-edge sensor technology and microcontroller-based control systems to enhance the robot’s navigation and operational capabilities. The hexapod robot, modelled after insect morphology, aims to efficiently traverse complex terrains and autonomously perform surveillance, agricultural monitoring, and search and rescue operations. The core of the project is intelligent planning and implementing movement control that is efficient and effective. The robot must have a robust way of travelling from point A to B while having sophisticated edge cases for obstacle avoidance and variable terrain, which the students could implement at the primary level. Another project component is developing a user-friendly Android app, enabling manual and semi-autonomous control via a robust Bluetooth communication protocol. This app provides a real-time interface for direct control and monitoring of the robot, enhancing user interaction and operational flexibility. Through the development of a specialized walking gait, robust obstacle avoidance systems, and a dual-mode guidance system utilizing both infrared and ultrasonic sensors, coupled with advanced Bluetooth communication and app functionality, the BEAMstiquito establishes itself as a cost-effective and versatile solution in robotics. This abstract summarizes the project’s background, design philosophy, key technological implementations, and the primary objectives of enhancing mobility, autonomy, and functionality in robotic applications.

Keywords:

Hexapod robot, autonomous navigation, BEAM robotics, sensor integration, obstacle avoidance, guidance systems, Android app, Bluetooth communication

2. Introduction & Background

The BEAMstiquito project is a groundbreaking exploration of versatile and autonomous hexapod robots in the ever-advancing landscape of robotics. Within this burgeoning field, the hexapod's potential extends far beyond the confines of mere novelty, finding invaluable application in addressing real-life challenges. From navigating treacherous terrains and agricultural monitoring to conducting search and rescue operations in harsh conditions or providing surveillance assistance, the hexapod emerges as a transformative solution with many practical uses. [2][3]

The BEAMstiquito project is rooted in the innovative methodology known as Biology-Electronics-Aesthetics-Mechanics (BEAM), a design philosophy that marries traditional digital circuitry with advanced microcontroller and sensor technologies. This fusion propels the hexapod into the realm of semi-autonomous task planning and operational execution, paving the way for its versatility in tackling diverse challenges. [2][3]

Inspired by the BEAM model, the BEAMstiquito is an economical hexapod robot designed to look like an insect. Comprising two essential components—the base, responsible for fluid movement in any direction, and the embedded microcontroller serving as its cognitive hub—the project seeks to unlock the latent potential of this hexapod marvel for practical, real-world applications. [2][3]

The implications of the BEAMstiquito project are far-reaching. With its hexapod design, the robot embodies adaptability and resilience, characteristics crucial for navigating complex environments. Its applications span domains such as exploring rough terrains, monitoring agricultural landscapes, undertaking search and rescue missions, conducting surveillance in challenging conditions, and more. The hexapod becomes a dynamic and transformative solution capable of surmounting challenges that traditional forms of robotics may find daunting. [2][3]

This engineering design project represents not just a technical innovation but a paradigm shift in the application of robotics. By seamlessly blending traditional design principles with cutting-edge technologies, the BEAMstiquito project propels hexapod robots into practical problem-solving, marking a significant stride toward integrating autonomous and versatile robotics in our daily lives. [2][3]

3. Objectives

Our project concerns creating a six-legged hexapod robot with manual and autonomous control. The robot can intelligently and independently move around obstacles using sensor-based collision avoidance. The hexapod robot will also have a manual override mode, which turns off its autonomous behaviour, controlling the robot wirelessly via an Android phone app. Along with this, it should also exhibit a fully autonomous behaviour which allows it to return to a "base station," for example, to recharge, without needing any directions from the user on how to walk. In this instance, the robot will be guided by a navigation system that helps the robot reach the base station. The primary goals of the project include: [2]

- **Understanding Hexapod Construction and Behavior:** To comprehend the design and operational principles underlying the construction of Freenove's big hexapod robot, including its mechanical structure, electronic components, and behavioural algorithms, thereby facilitating a comprehensive understanding of hexapod robotics and its applications.
- **Design a suitable walking gait:** Designing a suitable walking gait for Freenove's big hexapod robot involves creating a coordinated sequence of leg movements that optimize stability, efficiency, and adaptability to different terrains. The gait should ensure smooth locomotion while minimizing energy consumption and maximizing payload capacity. Key considerations include leg coordination, body oscillation, and sensor feedback integration to effectively navigate obstacles and uneven surfaces. An optimal walking gait can be developed to enhance the robot's overall performance and functionality across various operational scenarios by iteratively adjusting parameters such as step length, step height, and phase offsets.
- **Guidance System Implementation:** Design and implement a robust homing mechanism for the robot, facilitating precise detection of its home position, efficient navigation, and accurate docking and halting at the home location. This involves thoroughly comprehending design challenges, implementing an effective signal detection mechanism, devising an optimal navigation algorithm tailored to the specific requirements, and seamlessly integrating the underlying technology into the system.
- **Obstacle Avoidance:** To recognize the necessity of sensory technology in enhancing the functionality of Freenove's big hexapod robot by implementing obstacle avoidance capabilities, thereby enabling it to navigate its environment autonomously and safely. This objective encompasses understanding the principles of sensor integration, signal processing, and decision-making algorithms to develop an effective obstacle detection and avoidance system tailored to the robot's requirements and operational scenarios.
- **Wireless Connectivity**: Establish a robust wireless link, utilizing Bluetooth technology, between the hexapod robot and a base controller for manual or semi-autonomous operation, and enable a direct USB connection to a laptop or a Bluetooth link to a smartphone-based GUI app.

4. Theory and Design

4.1. Hexapod System

4.1.1. Robot Construction

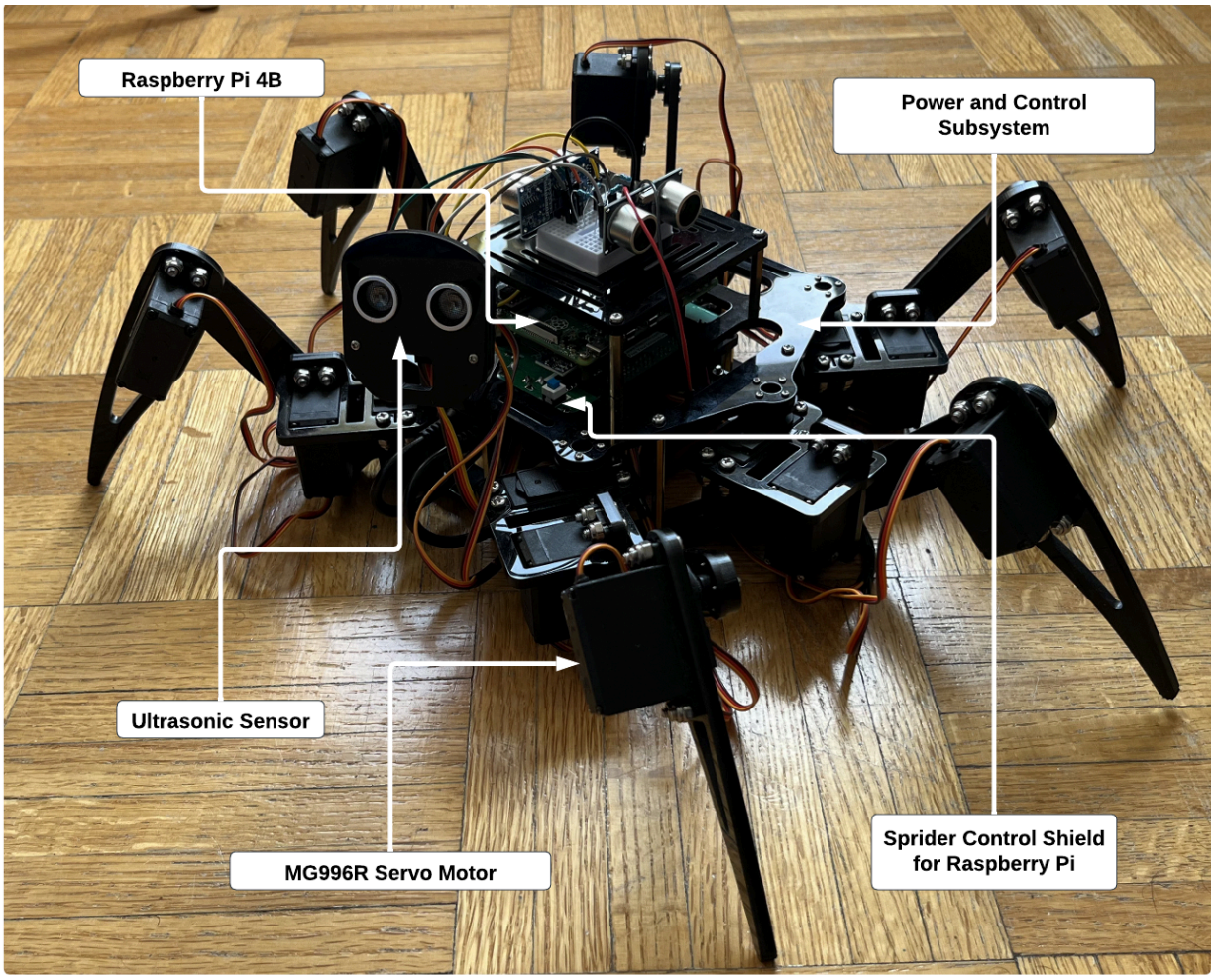


Figure 4.1.1.1: Robot System

The hexapod robot's construction is centred around a robust and lightweight acrylic chassis that supports eighteen servo motors, each controlling an individual leg joint. This structure allows for a high degree of mobility and flexibility in the robot’s movements. The mechanical design ensures stability even on uneven terrains, which is crucial for real-world applications. Each leg of the hexapod consists of three segments mimicking biological counterparts, which are pivotal for complex movements such as rotation, extension, and contraction.

4.1.2. Raspberry Pi 4B

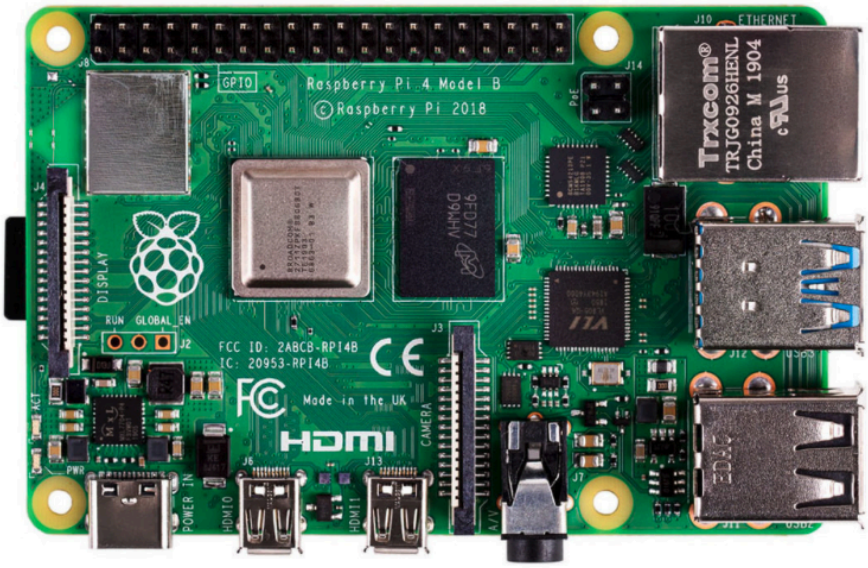


Figure 4.1.1.2: Raspberry Pi 4B

The Raspberry Pi 4 Model B is distinguished by its high-performance capabilities and extensive connectivity features, making it a preferred choice for complex robotic projects like the Freenove Hexapod Robot. It features a Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC running at 1.5GHz, providing ample computational power for data processing and real-time control. The microcontroller is equipped with multiple connectivity options, including Bluetooth 5.0, dual-band WiFi (2.4 GHz and 5.0 GHz), and Gigabit Ethernet, all of which enable versatile networking solutions. Additionally, it supports up to 4GB of LPDDR4-3200 SDRAM, offering significant memory capacity for applications requiring intense data manipulation and storage. These features are crucial for handling the complex algorithms required for motion control, sensor data integration, and real-time communication in robotic applications. [15]

The Raspberry Pi 4 Model B interfaces effectively with the Freenove Spider Shield, a key component in managing the Hexapod's servo-driven mechanics. This interface is facilitated through the GPIO pins available on the Raspberry Pi, which connect directly to the Spider Shield. These connections allow the Raspberry Pi to send control signals to the shield, which then distributes these signals to each of the eighteen servo motors responsible for the Hexapod's leg movements. This integration underscores Raspberry Pi’s role not just as a processor but as a central control hub, orchestrating complex sequences of actions that bring the robotic system to life. Through this setup, the Raspberry Pi 4 leverages its advanced capabilities to effectively manage the nuanced requirements of robotic control and sensor data processing inherent in the Hexapod Robot. [1]

4.1.3. Control Subsystem and Sensors

4.1.3.1. Control Board Design

The control board of the Freenove Hexapod Robot serves as the central hub for interfacing the Raspberry Pi with the various modules and components essential for the robot's functionality. It accommodates necessary connectors for the PCA9685 and ADS7830, ensuring optimal power distribution and signal integrity for reliable operation. This board's layout is meticulously planned to facilitate easy connections to the servos, sensors, and the Raspberry Pi, streamlining the assembly and programming process for users. [1][7]

Central to the control board's functionality is the PCA9685 integrated circuit (IC), which plays a critical role in managing servo control for the hexapod robot. This IC facilitates the precise and synchronized movement of the robotic limbs by generating Pulse Width Modulation (PWM) signals across its 16 output channels. Each channel controls an individual servo motor, with the PWM duty cycle determining the position or angle of the servo shaft. This fine-grained control is pivotal for allowing the hexapod to execute complex movements with high accuracy and fluidity, enhancing its mobility and agility. Additionally, the PCA9685 can operate in an external clock mode, which is crucial for maintaining synchronized movement across all servos, thus ensuring balance and coordination in the robot's gait. [1][7]

The PCA9685's adaptability with various power supply options and its ability to operate over an extended temperature range also make it suitable for challenging environments where the hexapod might operate. Moreover, the Spider Shield, which connects directly to the Raspberry Pi, leverages this IC to manage the complex servo operations efficiently, enhancing the hexapod's overall performance in navigation, obstacle avoidance, and terrain adaptation. This comprehensive integration of the control board with the Raspberry Pi and critical ICs like the PCA9685 underpins the advanced capabilities of the Freenove Hexapod Robot, enabling sophisticated operational tasks essential for real-world applications of robotic technology. [1][7]

4.1.3.2. MPU6050 Module

The MPU6050 is an inertial measurement unit (IMU) module that combines a gyroscope and an accelerometer to provide comprehensive motion sensing capabilities. The gyroscope measures the rate of rotation around three orthogonal axes, while the accelerometer measures acceleration along those same axes. Utilizing microelectromechanical systems (MEMS) technology, the MPU6050 module detects changes in angular velocity and acceleration by measuring the deflection of tiny internal structures in response to inertial forces. [7]

By integrating the output from both the gyroscope and accelerometer, the MPU6050 module provides precise information about the orientation and movement of the sensor module in three-dimensional space. This data is crucial for tasks such as stabilizing and controlling the

orientation of robotic platforms, tracking movement trajectories, and enabling gesture recognition.Additionally, the MPU6050 module often incorporates a Digital Motion Processor (DMP), which offloads complex sensor fusion and motion processing tasks from the host microcontroller, thereby reducing computational overhead and enhancing system performance. [7]

4.1.3.3. Ultrasonic Module (HC-SR04)

The HC-SR04 ultrasonic sensor module operates on the principle of echolocation, similar to how bats navigate in the dark. Utilizing ultrasonic sound waves, it determines distances to objects by measuring the time taken for the sound waves to travel to a target and bounce back to the sensor. The module consists of two main components: a transmitter and a receiver. The transmitter emits short bursts of ultrasonic waves, typically at a frequency of 40 kHz. These waves propagate through the air until they encounter an obstacle, at which point they reflect back toward the sensor. The receiver then detects these reflected waves. By precisely measuring the time interval between sending the ultrasonic pulse and receiving its echo, the HC-SR04 module calculates the distance to the obstacle using the formula:

$$Distance = \frac{Speed\ of\ Sound \times Time}{2}$$

where the speed of sound is the velocity of sound in air and the time is the duration between transmission and reception of the ultrasonic pulse. The HC-SR04 module typically provides distance measurements with high accuracy and reliability, making it a popular choice for applications such as obstacle avoidance, distance measurement, and object detection in robotics.

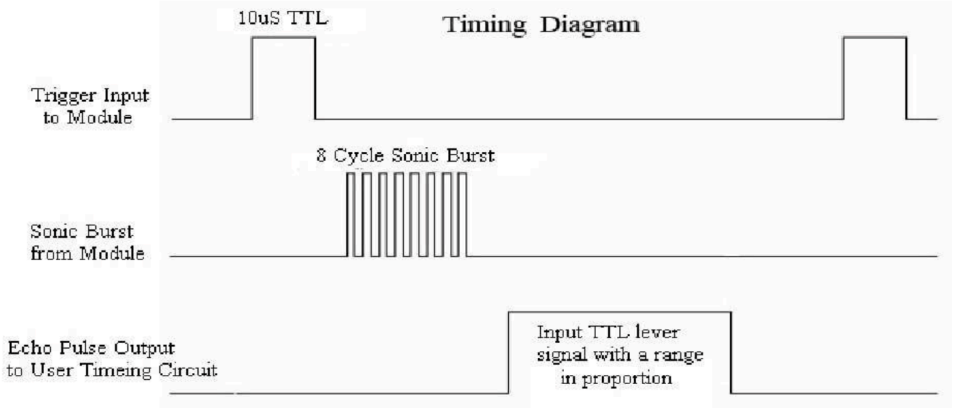


Figure 4.1.3.3.1: Timing Diagram [6]

To ensure proper operation of the HC-SR04 Ultrasonic sensor with the Raspberry Pi, two considerations were addressed. Firstly, compatible power ports were verified on the robot to supply the sensor with its required 5V operating voltage. Secondly, due to the Raspberry Pi GPIO pins operating at 3.3V, voltage level conversion was implemented to downconvert the 5V sensor readings. This measure not only ensured accurate readings but also protected the GPIO pins from potential damage due to voltage mismatches.

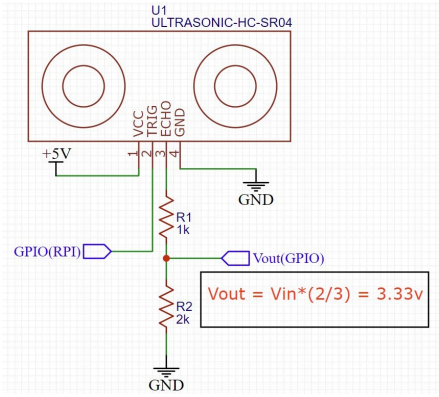


Figure 4.1.3.3.2: Circuit Schematic of Ultrasonic Sensor

4.1.3.4. SHF-4544 IR Transmitter And SHF-213FA Receiver

The SHF-4544 IR transmitter operates as a directional transmitter emitting IR radiation at 950 nm within a 20-degree cone. It provides a minimum intensity of 280mW/sr at 100mA current, with a forward voltage drop of 1.6V. The maximum safe continuous current is 100 mA.The intensity of the signal decreases as one moves away from the transmitter. Usually, the intensity is given in mW/stearadin. By steradian definition, to find the intensity at a distance r that can divide intensity per steradian by the front area of the steradian. [11]

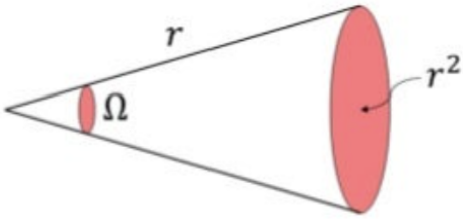


Figure 4.1.3.4.3: Signal intensity = (280mW/sr)/r^2

This calculation provides us with the intensity per unit area at the specific location, enabling us to determine the range of intensity requirements for the receiver to detect the signal. The forward voltage drop across this model is 1.6 volts. To control the current based on intensity requirements, an external resistance in series with the diode is necessary. The resistance value can be calculated using:

$$I = (Vbat - Vdiode)/R$$

When combining multiple transmitters to cover a wider area, precautions must be taken to avoid blind spots. Since IR transmitters emit radiation in a cone shape, positioning them apart may create spots between transmitters with weak or no signal coverage. To mitigate this, transmitters can be positioned closely together at desired angles to ensure comprehensive coverage.

The SHF-213FA IR receiver is a compatible model with a maximum detection wavelength of 950nm. For this receiver to detect a signal, the minimum intensity required is 0.002uW/cm^2, resulting in a current of 0.2uA passing through the IR diode. Its view angle is +/- 10 degrees, indicating that it can only detect signals within this angular range. However, this directive ability decreases at high signal intensities. Also, it requires an unobstructed line of sight for optimal functionality. [12]

To maintain directional sensitivity even at close proximity to the transmitter, a cylindrical aluminum foil shield is utilized. This shield limits unwanted signals, allowing detection only within a range of approximately +/- 8 degrees towards the transmitter.

To enhance sensitivity, the IR receiver is reverse-biased with a maximum reverse voltage limit of 20V. A resistor in series with the IR photodetector diode is crucial for signal detection by the microcontroller. The voltage drop across this resistor, proportional to the current passing through the diode, can be read by the microcontroller. The resistor value is chosen based on the resolution of the Analog to Digital Converter (ADC). Amplification may be necessary to boost the voltage signal, especially at low signal intensities where the voltage drop is minimal due to the small current flow.

4.2. Motion Control Kinematics

4.2.1. Design Philosophy

Unlike a simple robot with wheels, which can be easily steered and driven forward and backward, there was a lot more to be considered when it came to designing a 6 legged robot. The first step was to research everything about how a hexapod worked and what it needed to function. Considerations like how will the legs be structured in terms of the servos, and their lengths. How many degrees of freedom are needed? How will these six legs coordinate to move?

There are many complex possibilities for the movement one can design for a robot like this. It came to the idea that for general planar navigation from point A to B, a periodic, repeatable movement gait would be a reasonably simple and effective solution. When it came to more specific requirements like navigating over specific bumpy terrain, more advanced movement algorithms could then be implemented.

In the domain of limbed robots, a concept that can be derived from nature can be applied to robotic movement. Just like how animals take on a periodic cyclical movement of their limbs to walk, movement gaits can be used to move robots. For the hexapod robot, six legged movement gaits have already been devised in the past, such as the tripod gait, wave and ripple gaits. These three gaits share a similarity, with each leg performing 4 phases of movement in a cycle: lifting the leg, moving it forward in the air, and then placing it down. It is finally dragged back to its original position relative to its shoulder hinge.

The difference between these 3 gaits can be described as a different choice of phase offsets for these per-leg cyclical movements. As a simple overview, the tripod gait moves 3 legs synchronously at a time, whereas wave gait moves 2, and ripple gait moves 1. The tripod gait

was chosen due to its stability and simplicity which allowed for an ease of designing for precise movement.

4.2.2. Walk plan

The first plan was to create movement which could travel a straight path in any specified direction. Next was to decide on implementation specifics of creating the walking gait. The plan was to have the six legs move all towards a singular direction with a simple gait and to define an x,y "velocity" vector which all six legs would follow which could allow for movement in any direction. Other considerations like how the legs must clear a certain height as they move, or the speed at which they travel should also be generalized in the code. Making modular code is important here because it allows for flexible modifications.

4.2.3. Our starting point requirements

To extend on what was mentioned of the servos in part 4.1, for a design of 18 degrees of freedom 18 servos are required. The discussion of movement code above assumes an ability to freely and reliably set precise angles or "positions" for all these servos at any given time, however there are multiple ways to control these servos and mechanisms to make them more precise.

For servo control, there are many methods, such as direct analog voltage control, which maps an input voltage to a range of servo positions, or Pulse Width Modulation (PWM) control. Due to the constraint of needing a precise and stable analog voltage source, PWM is more commonly used, where the input is a pulse of some range of duty cycle which is mapped to the servo position. If one decides to use a "smart" servo which is able to decode instructions (along with the benefits of other features which vary between models), then one can use a serial communication to send instructions to the servos (something like UART can work).

Beyond how information is sent to the servo, considerations must be made to the required speed and accuracy of the servo. Some servo models have the ability to measure their current position (which may not match the intended input position at every given time), and from this a control system can be used to make the control of the servo match more closely with the intended input. Some servos take this a step further and come with built in feedback systems.

As for this project, it was discovered very quickly that direct servo PWM control and control systems for stability would be a lot of work to take on. So, a robot kit that dealt with the mechanical part of the construction of the legs, along with low level servo control and control systems was required. Ideally, one would interface with the legs by being able to set a target position of cartesian coordinates for the foot of each leg to follow, which would require inverse kinematics.

More specific to the application of limbed robots is a coordinate to servo angle conversion process, which is done using inverse kinematics. Depending on the amount of joints of the limb, and the length of the limb segments, trigonometry can be done to make this conversion. A difficult task of programming servo angles for leg movement can be made much

more precise when working with cartesian coordinates of the feet instead. Though, similarly to operating within the maximum angles of the servos, one must operate within the maximum range of coordinate that can be achieved by the limb. Overall, without inverse kinematics, one would not be able to imagine a starting point for programming servo angles for robotic movement. However, with the framework of programming cartesian foot positions over time, making a set of discrete movements to make a stepping motion becomes clearer.

The kit selected met all these baseline requirements (pre-built, servo control, control system, and IK). However, since a lot came pre-decided by the kit, this choice constrained our microcontroller and code language options. Due to the code base provided being in Python, Python was chosen as the language of development for our motion controls, and it would all be implemented onto a Raspberry Pi, which was specified by the kit to control other aspects of the robot.

4.2.4. Accurate computer simulation

Beyond determining the movement code, another thing to consider is simulating the movement of the robot before fully constructing and testing it physically. This step could avoid damaging physical parts with erroneous code. Like always, research came first in deciding the software to choose, and Webots was settled on. It is free, supports Python, which the kit's code base requires, and physical simulation is accurate enough for this project's purposes given the approximations used for the model anyway. A 3-D computer model of our robot is also needed to be created, with the same leg dimensions, amount of servos (18) and positional "shoulder" placement of legs on the robot's body.

At a general level, a robot simulation software works by placing a model of a robot in a simulated environment where physics are calculated. Code controlling the servos directly can be tested to verify the movement of the physical robot in a 3d environment that physically acts like reality. As for how the robot model can be represented, many different formats related to the field of 3d modeling on computers can be used, along with any associated 3d modeling software.

For this project, the simulation started in Webots by getting more familiar with demo projects in order to gain experience and knowledge with the software. To start, a simple hexapod demo project provided by Webots that is coded in C was manually converted to Python line by line, to gain familiarity with the Python-based Webots simulation that would be undertaken later on. Due to the ideal servos of Webots, the Freenode control system Python code would not have to be ported in, however, the inverse kinematics would, so development in Python would still be required.

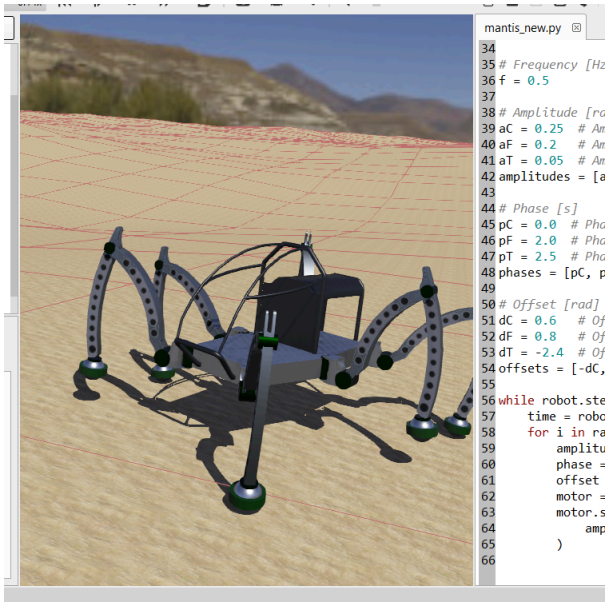


Figure 4.2.4.1: Webots and the demo hexapod walking around using the replacement Python demo code.

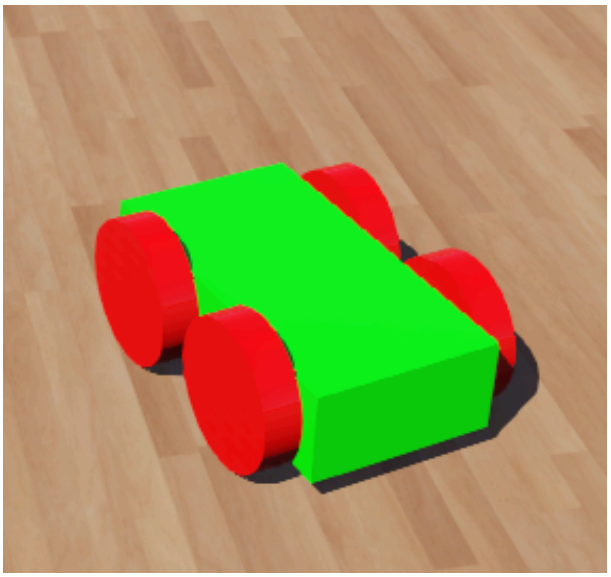


Figure 4.2.4.2: The simple 4 wheel car in Webots.

With the demo code conversion being successful, the focus shifted to modelling. A more realistic model of the robot would be needed in order to more accurately simulate the robot walking. To gain experience with creating robot models, a simple 4 wheeled car was first created and properly configured with motors and joints, along with a basic script to control its movement. The script simply had it drive forward and backward, along with turning in place by having the right wheels spin backward, and the left wheels spin forward.

Now with this experience, an accurate model could be made. At its core, to create a computer model of the robot, the only factors that truly matter are the limb segment lengths, and their positions relative to each other. If these dimensional factors are accurately replicated, the movement of the robots will be the same. Accurately copying the specific geometry and shape of the components is irrelevant.

The physical measurements of the physical robot parts were taken. Special attention was paid to the fact that the body was not a perfectly regular hexagon. Its long hexagonal side (13cm) and short side (10.5cm) were measured, along with its long axis (19cm) and short axis (17cm). These particular measurements were important not only positionally for the hinge joints of the 6 legs but also for the angle that they made in default with respect to the center of the body, on which the internal coordinate system bases its axes. The leg segments of length 5,9 and 11cm were also measured (listed from top to bottom).

In the 3d modeling software blender, a thin 3d hexagon was made and exported into Webots. From there, the rest of the 18 segments could be modeled directly in Webots using their basic “cylinder solid”. The 6 legs were created one segment at a time, mindful of the object hierarchy between the segments with hinge joints and the axis of rotation that they made with the robot relative to higher hierarchical objects. To simplify, the “shoulder” servos moved in the x-y

plane, and the “leg” and “foot” servos moved parallel to the z plane. After some issues (like properly enabling physical collision between the right objects), and more manual work the following model was achieved.

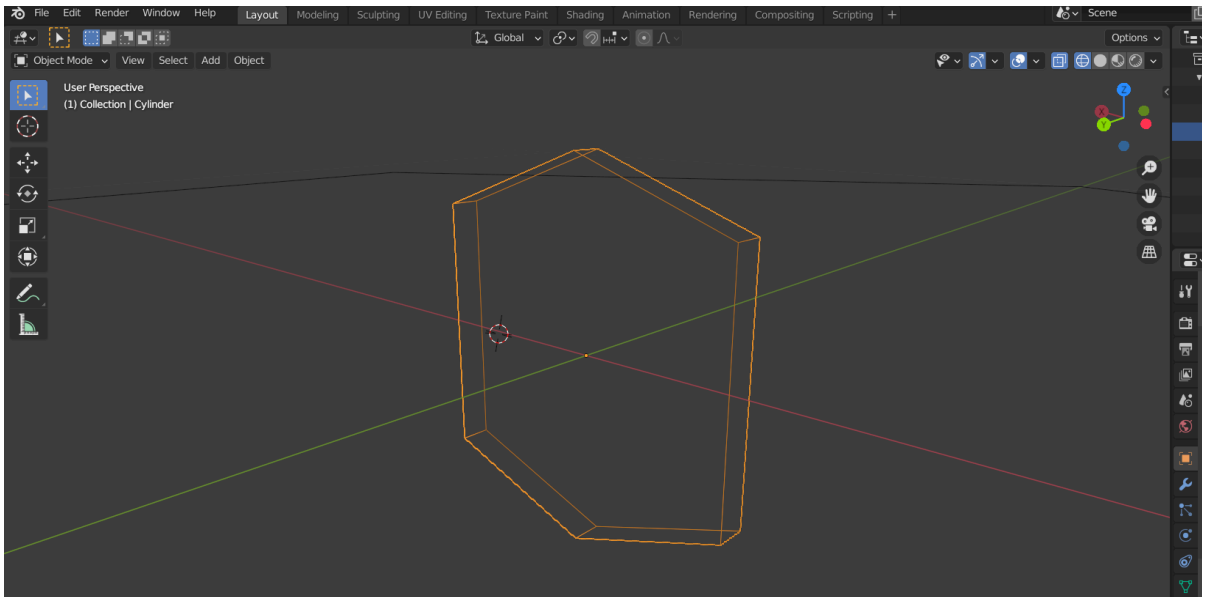


Figure 4.2.4.3: The hexagon mesh created in Blender that is specific to the robot’s measurements.

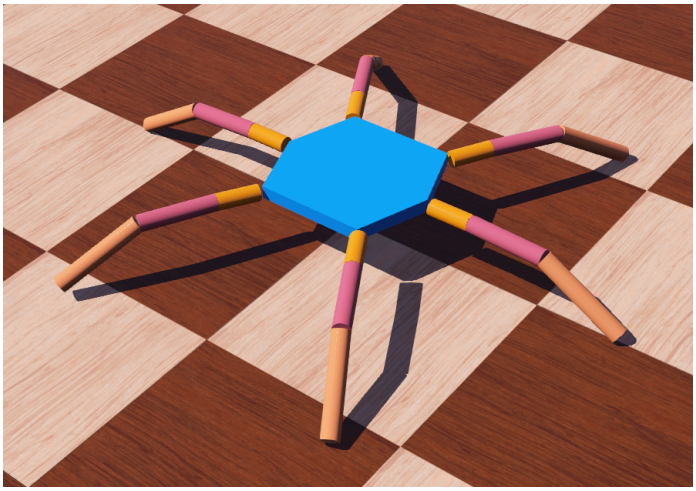


Figure 4.2.4.4: The completed robot design in Webots, properly set up with servo control and physics simulation. Separate leg segments are color-coded for visibility.

4.2.5. Sinusoidal tripod testing gait

Since the Freenove lower-level Python code was not yet converted and ported over to Webots, in order to test that the model was working properly, a “sinusoidal tripod walking gait” was started with, in which all the joints were given sinusoidal functions for position with time, with predefined phase offsets and amplitudes that were tuned empirically for a working result.

This ensured the model’s hinges and motors were properly created, and that the simulation would be effective.

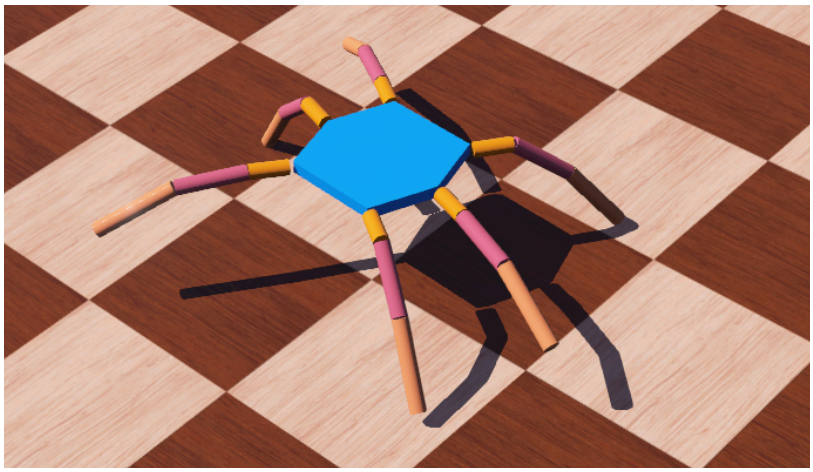


Figure 4.2.5.1: An image of the simulated robot in the middle of its walk cycle.

4.2.6. Porting over and converting the given lower-level code

After being successful in creating the custom robot model achieve a basic walk cycle, the next step was to port over the lower-level inverse kinematics code into Webots, and all other required code that allowed us to be able to implement our own walking cycle from the level of controlling the cartesian coordinates of the feet. After some careful debugging to get everything working, the desired motion control could be built off this foundation of code.

4.2.7. The custom implementation of a walking gait and rotation in place.

In the end, the design allowed for omnidirectional walking at a given stride length and speed, and an ability to rotate the robot in place by a given angle. On a general level, the code stores cartesian coordinates of the feet, with the axis for each foot pointing radially outward from the center of the robot’s body. Our walk method takes in an x and y argument, and from this, a “velocity” vector is created in the given direction the robot wants to move, from which one can create an organized set of movements with each set of 3 legs to achieve a basic walk cycle. The approach consisted of a set of relative target positions (defined by walk direction, speed, and stride length) that the feet would need to reach at my specific defined time intervals. That is, the walking code defines a specific stage-based movement of the legs by defining relative coordinates in space that need to be achieved at certain times of the walk cycle. In order for the legs to reach the correct positions in space for any given movement walk cycle speed, the code had to make it flexible. Linear interpolation was used between the distinct walk cycle XY positions for each 1/100th of a second, with the delta between interpolations being larger for faster speed settings. As for raising the legs in the Z axis, half the cycle was grounded, and the other half followed an arc path defined by a sinusoidal function, which also can be interpolated at any speed “delta”.

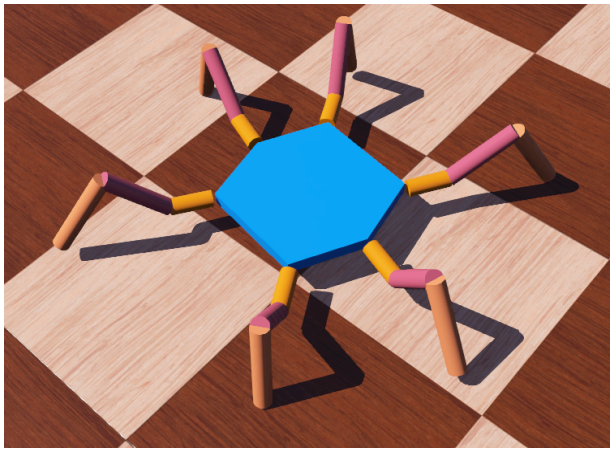


Figure 4.2.7.1: The robot walking forward.

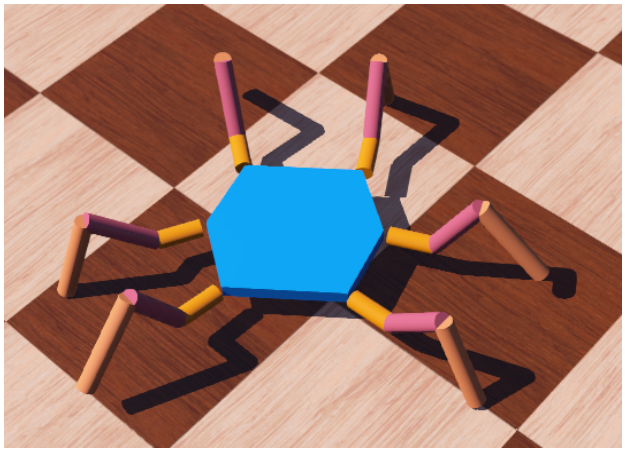


Figure 4.2.7.2: The robot rotating in place.

4.2.8. Converting the code to work on the Raspberry Pi and fine-tuning

Since the simulation software had a different mechanism for servo control and time simulation, The code's method of setting the servos and processing changes in time had to be converted to work on the Raspberry pi. PID control on the simulation side was omitted, as the simulated servos were made "ideal".

Finally, the converted Raspberry Pi code was tested on the physical robot, and slight tuning adjustments were made to ensure the best performance. In the end, the motion controls allowed for omnidirectional walking at a given stride length and speed, and an ability to rotate the robot in place by a given angle. Code was also created to implement the walking code into executing with our joystick controller app.

4.3. Obstacle Avoidance

4.3.1. Sensor Fusion

Three ultrasonic sensors are strategically positioned to cover the front and sides of the robot's environment. The code continuously reads distance measurements from these sensors, obtained through echolocation principles, where ultrasonic waves emitted by the sensor bounce off nearby objects and return to the sensor. By comparing these distance measurements with a predefined obstacle threshold, the code determines whether obstacles are present within the robot's vicinity. This information directs the robot's movement.

4.3.2. Reactive navigation Approach

Pseudocode:

1. Import necessary modules and classes for sensor control, servo control, and threading.
2. Create instances of the Ultrasonic class for each sensor.
3. Create instances of the Control class and Servo class for robot control.
4. Define constants and parameters such as obstacle threshold, head positions, and movement direction.
5. Set the initial position of the robot's head using the servo.

6. Define functions for rotating right and moving forward.
7. Enter a continuous loop for robot operation.
   8. Update the position of the robot's head and measure distances using ultrasonic sensors.
   9. If the current head position is 1, set the head to scan left and measure the distance.
      If the current head position is 2, set the head to scan the center and measure the distance.
      If the current head position is 3, set the head to scan right and measure the distance.
   10. Check if the distances measured by the sensors indicate obstacles.
      If all distances are above the obstacle threshold, move the robot forward.
      If any distance is below the obstacle threshold, rotate the robot right.
   11. Increment the loop counter.
12. End of loop.

Initially, the code imports necessary classes and modules for sensor control, robot movement, and servo operation. It then initializes instances of the Ultrasonic class for each sensor and a Control class object for robot control. Servo positioning is managed by an instance of the Servo class. The code defines parameters such as obstacle detection thresholds, head movement positions, and delays. A continuous loop is established to monitor the environment using ultrasonic sensors, rotating the robot's head to scan for obstacles in different directions. Distance measurements obtained from the sensors guide the robot's actions: if all distances are above the threshold, indicating a clear path, the robot moves forward; otherwise, it executes a rotation maneuver to avoid obstacles. This systematic approach to sensor-based decision-making ensures the hexapod robot can navigate its surroundings intelligently and autonomously.

4.3.3. State Machine Approach

Pseudocode:

1. Import necessary modules and classes for sensor control, robot movement, and threading.
2. Create instances of the Ultrasonic class for each sensor.
3. Create instances of the Control class and Servo class for robot control.
4. Define obstacle detection thresholds for each sensor and a state enumeration for robot behaviour.
5. Set initial head position and delay parameters.
6. Define functions for robot movement, servo positioning, obstacle detection, and decision-making.
7. Initialize the state to MOVE_FORWARD and a variable for boundary following direction.
8. Enter a continuous loop for robot operation.
   9. Check if the path is clear ahead.
      10. If clear, move forward; else switch to DETECT_OBSTACLE state.
   11. In the DETECT_OBSTACLE state, determine the direction of the obstacle.
   12. If obstacles are ahead, move forward; else switch to the FOLLOW_OBSTACLE state.
   13. In the FOLLOW_OBSTACLE state, navigate around the obstacle based on its direction.
   14. Once the obstacle is cleared, switch to CHECK_CLEAR_PATH state to verify if the path ahead is clear.
   15. If the path ahead is clear, return to the MOVE_FORWARD state; else continue FOLLOW_OBSTACLE.
   16. Add a small delay to the loop to prevent excessive CPU usage.
17. End of loop.

It first imports the necessary classes and modules for sensor control, robot movement, and threading. Ultrasonic sensors are instantiated to monitor the robot's surroundings, while instances of Control and Servo classes handle robot control and servo operation, respectively. Obstacle detection thresholds and a state enumeration define the robot's behaviour. The script continuously operates in a loop, checking the path ahead for obstacles. If the path is clear, the robot moves forward; otherwise, it switches to obstacle detection mode. During obstacle detection, the robot determines the direction of obstacles using servo-controlled scanning and decides whether to move forward or follow the obstacle. State-based logic guides the robot's actions, ensuring it can navigate its environment intelligently and autonomously, even in the presence of obstacles.

4.4. Guidance System

The autonomous homing mechanism is designed to enable a robot to navigate and dock accurately at a designated home location autonomously. This mechanism is divided into two distinct phases, each employing different technologies to achieve specific objectives.

4.4.1 Design Phase 1: Bringing Robot Closer to the Home Location - Infrared Technology Approach

In this phase, infrared technology is employed.

4.4.1.1 Infrared Transmitter and Receiver Setup:

The setup includes five infrared transmitters, each with a 20-degree beam angle and positioned at 20-degree intervals starting at 10 degrees. The transmitters are powered by a 9-volt battery. The optimal placement is in a corner, covering a 90-degree area. The receiver, linked to the robot's microcontroller, demands a minimum signal intensity of 0.002mW/cm² for detection. Both transmitters and the receiver are elevated at the same height for unobstructed line-of-sight communication.

4.4.1.2 Right-Hand Algorithm for Signal Detection:

For returning home, the right-hand algorithm is initially employed to find the signal. During this process, a RotateInPlace() function will be called which allows the robot to rotate on its axis clockwise or counterclockwise with a step of the desired amount of degrees. In our case, it is ideal to rotate in 8-degree steps until the signal is detected. The receiver's acceptance angle of +/- 10 degrees widens as the robot nears home, requiring a cylindrical aluminum foil cover for signal detection accuracy.

However, this rotation approach will take almost 42 seconds to complete the one complete cycle which is quite a lot of time. To address the lengthy rotation time for one full cycle, a revised approach is considered. The robot is only allowed to rotate one complete cycle in steps of 8 degrees in specific scenarios such as to find the signal after the robot encounters the obstacles or for the first time when the auto home function is initiated. In regular cases, if the

robot has lost the signal the robot will rotate +8 degrees to the right if the signal is not detected again then the robot will rotate -8 degrees to the left for two steps. If still not detected then the robot will be allowed to perform a complete clockwise rotation until the signal is detected.

4.4.1.3 Signal Detection and Response:

Upon detecting infrared radiation, the IR photodiode receiver initiates a current flow, leading to a voltage drop across the series connected resistance. The microcontroller detects this voltage drop, facilitating signal detection. Upon detection, it stops rotating and proceeds in the signal's direction by calling the walk() function while continually checking for signal presence. The walk() function takes parameters such as walking speed and direction. If the signal is lost during movement, the robot repeats the right-hand algorithm to reacquire the signal.

While effective for bringing the robot closer to the charging station, infrared technology alone may not provide the precision required for docking accurately. Therefore, an additional technology is introduced in Phase 2.

4.4.2 Design Phase 2: Ultrasonic Sensor Approach

The robot proximity with respect to the home will be determined by the IR signal strength. Once the robot is within proximity, the robot will rely on the ultrasonic sensors only. This phase focuses on achieving precise docking using ultrasonic sensors. The key components and steps involved are

4.4.2.1 Ultrasonic Sensor Setup:

Two ultrasonic sensors are utilized—one fixed at the home location with a 15-degree beam angle, angled 45 degrees in the corner, and the other sensor mounted on the robot, rotatable using a servo motor.

4.4.2.2 Docking Process:

The robot's ultrasonic sensor rotates to measure distances from the two side walls along the home, aligning itself centrally in the corner by calling the walk() function. This time the parameters will be passed to the function to move side by side instead of moving forward. The sensor at the charging station detects the robot's presence calculates the distance between the station and the robot and sends this information to the robot microcontroller via Bluetooth. Simultaneously, the robot's sensor measures the distance to the station. If these distances match, the robot proceeds forward with its movement; otherwise, it realigns itself.

4.4.2.3 Criteria for Docking:

The robot docks and stops at the home location when the distance between the robot and the home ultrasonic sensor location is less than or equal to 3cm. Challenges include designing appropriate walls for reflection, managing angle incidence for sensor detection, accurate

alignment despite the robot's step movements, and coordinating sensor triggering to avoid cross-communication.

4.4.3 Analysis

4.4.3.1 Power Requirement Assessment

Initially, The receiver is selected with a narrow acceptance angle that is affordable in the market, focusing on the minimum intensity needed for signal detection. After considering these criteria, the SHF 213 IR photodiode receiver is considered. The SHF 213 receiver has an acceptance angle of +/- 10 degrees, with a minimum required signal intensity of 0.002mW/cm^2.

To determine the minimum signal power necessary at the emission point to achieve an intensity of 0.002mW/cm^2 at a distance of 2 meters, The following calculations are performed.

The minimum intensity required to be detected by the receiver.

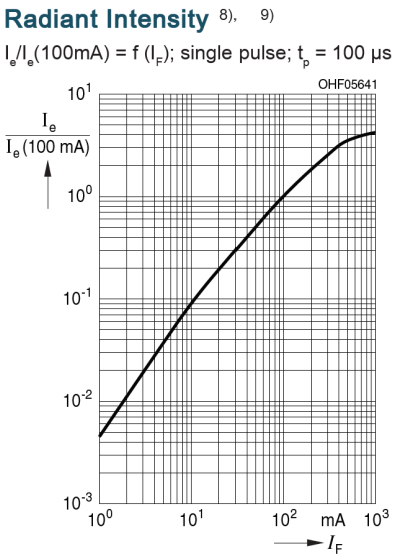
$$\frac{o.oo2mW}{cm^2} \times \frac{10000cm^2}{1m^2} = 20mW/m^2$$

At a 2-meter distance, the steradian front is $A = r^2 = 4m^2$

$$The\ transmitter\ intensity = 4 \times 20 = 80mW/sr$$

This model has an absolute maximum continuous current rating of 100 mA, producing a minimum signal intensity of 280mW/sr at this current. However, it has a 20-degree beam angle and requires the use of 5 transmitters to cover a 90-degree area when placed in a corner. It consumes a lot of power however the current passing through the diode can be reduced just to get the desired power only which is 80mW/sr. It's important to note that the calculations above are based on ideal conditions. In reality, environmental factors and external light interference can attenuate the signal, affecting performance

4.4.3.2 Transmitter Circuit Analysis:

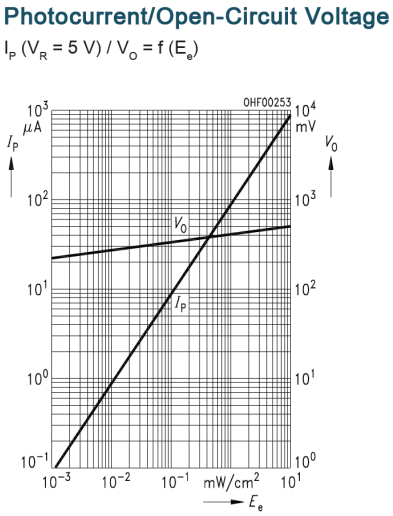


[11]

According to the datasheet the forward voltage across IR LED is 1.6V typically, and the intensity at 100mA is 280 mW/sr. At I=30mA the intensity is 84 mW/sr which is required by the above analysis. There are 5 transmitters in series, 1.6 volts forward voltage drop across each transmitter.

Based on that $R = \frac{V}{I} = \frac{9-(1.6\times5)}{30mA} = 33.3\ ohm$

4.4.3.3 Receiver circuit analysis:



[12]

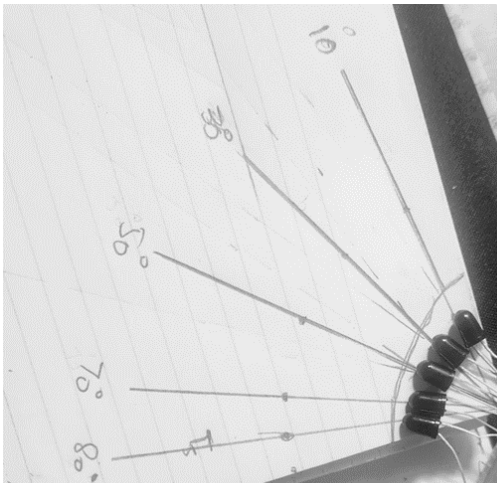
The microcontroller has a 10-bit analog-to-digital converter; the receiver is connected with a 5V pin of the microcontroller. The smallest digital voltage it can read is 5/2^10= 4.88mV. At minimum intensity, the requirement for the current pass through the diode in reverse bias is 0.2 uA. The minimum resistor required to detect the voltage across the resistor at minimum detectable signal is

$$R = \frac{v}{I} = \frac{4.88mV}{0.2uA} = 24.4k\Omega$$

Therefore, a 25$k\Omega$ resistor was selected.

4.4.3.4 Geometrical Analysis

Given that the robot's height is 10 cm and the obstacle height is 15 cm, it's crucial to position the transmitters and receiver above the obstacle height to ensure unobstructed infrared signal transmission. The infrared radiation emitted from the receiver forms a cone shape, with a beam radius of 0.35 meters at a 2-meter distance. To optimize signal transmission, positioning the IR transmitter at 30 centimetres above ground level is advisable. This height allows the radiation to avoid reflecting back from the ground, ensuring effective communication. Additionally, excessively high positioning may risk the robot tilting or losing balance, making the 30-centimeter height a practical choice.



Distance Range = 2m
Beam angle = 20 degree
Beam radius at 2m distance = 2sin(10)= 0.35 meters
Height of the robot = 10 cm
Height of the obstacle = 15 cm
Height of the transmitter and receiver = 30 cm
No. of transmitters required to cover a 90-degree area = 5

4.4.3.5 - Robot Rotation Analysis

Considering the original acceptance angle of the transmitter at +/- 10 degrees, narrowing the acceptance angle enhances directivity to approximately +/- 8 degrees. To ensure effective rotation without missing the signal, the step size for the robot's rotation should be smaller than or equal to 8 degrees. This choice accommodates variations in directivity as the robot moves closer or farther from the transmitter, maintaining reliable signal detection.

However, this rotation approach will take almost 42 seconds to complete the one complete cycle which is quite a lot of time. To address the lengthy rotation time for one full cycle, a revised approach is applied. The robot is only allowed to rotate one complete cycle in steps of 8 degrees in specific scenarios such as to find the signal after the robot encounters the obstacles or for the first time when the auto home function is initiated. In regular cases, if the robot has lost the signal the robot will rotate +8 degrees to the right if the signal is not detected again then the robot will rotate -8 degrees to the left for two steps. If still not detected then the robot will be allowed to rotate 360 clockwise until the signal is detected.

4.4.4 Design Limitations:

The functionality of this design is limited to a range of a few meters, requiring the IR transmitters and receiver to be positioned higher than obstacles to maintain a direct line of sight. When the robot is close to its home, no other obstacles should obstruct its path. The home for the robot must be situated in a corner, with side walls aiding in distance measurement for accurate alignment at the center for precise docking.

4.5. Wireless Connectivity and App Development

For proof-of-concept projects like this, the communication system plays a critical role in realizing the entire project, it maps the different functionalities that a robot can execute or perform with the attributes of a controller. A controller can be of various types such as a physical remote control, a Windows PC application, an Android or iOS application and many other forms. However, a controller may be inefficient without a wireless communication protocol such as TCP/IP (Wifi - IEEE 802.11), Near Field Communication, Classic Bluetooth, etc. Let us learn about these communication protocols in brief:

- TCP/IP (Transmission Control Protocol/Internet Protocol): TCP/IP is a suite of communication protocols used to interconnect network devices on the internet. WiFi, which operates based on the IEEE 802.11 standard, is a wireless networking technology that enables devices to communicate over a local area network (LAN) or the internet without the need for wired connections. It provides a reliable and standardized method for data transmission between devices. It consists of multiple layers, including the physical layer (802.11 for WiFi), data link layer, network layer, transport layer (TCP), and application layer. WiFi enables wireless communication by transmitting data over radio frequencies, allowing devices to connect to a wireless router or access point.
- Near Field Communication (NFC): NFC is a short-range wireless communication technology that allows devices to establish communication by bringing them close together (usually within a few centimetres). It operates on the principle of electromagnetic induction and is often used for contactless data exchange and transactions. NFC enables communication between devices by modulating and demodulating radio frequency signals within the 13.56 MHz frequency range. It supports three modes of operation: peer-to-peer mode (for device-to-device communication), reader/writer mode (for reading and writing NFC tags), and card emulation mode (for

emulating NFC cards or tags). However, with the context of this project, such a communication protocol would be very inefficient to use.

- Classic Bluetooth: Classic Bluetooth is a wireless communication technology that enables short-range data exchange between devices over radio frequencies in the 2.4 GHz band. It is one of the most widely used wireless communication protocols for connecting devices such as smartphones, laptops, robots, IoTs and other such devices. Bluetooth uses a master-slave communication model, where one device (the master, in our case the Raspberry Pi processor) establishes a connection with one or more slave devices (in our case the Android device). It supports various profiles and protocols for different types of data exchange, including audio streaming, file transfer, and serial communication (This project uses: RFCOMM). Using Bluetooth, devices can communicate over short distances (typically up to 10 meters). RFCOMM (Radio Frequency Communication) is a protocol used within the Bluetooth protocol stack. It provides a virtual serial port emulation over Bluetooth, allowing Bluetooth-enabled devices to establish serial communication channels similar to those provided by RS-232 serial ports.

These protocols are used for data transfer between the controller and the robot(s). For this project, TCP/IP (Transmission Control Protocol) and Classic Bluetooth were the choices of protocols to select between, but since Bluetooth technology does not require constant power for the communication to take place and consumes less power unlike TCP/IP, it makes Bluetooth a suitable protocol for battery-operated robots or devices where energy efficiency is crucial. This allows for prolonged operation without frequent battery replacements or recharges. Bluetooth typically requires less configuration and setup compared to TCP/IP networks. This can be advantageous in scenarios where robots need to establish communication quickly and without extensive network infrastructure. Pairing devices via Bluetooth is often a straightforward process, making it accessible even to users without extensive technical knowledge. Bluetooth can offer lower latency compared to TCP/IP, making it suitable for real-time communication requirements in some robotic applications. However, real-time video streaming becomes difficult using Bluetooth as the latency increases, in such cases, the TCP/IP network provides an advantage.

The six-legged hexapod developed in this project is controlled using an Android App that runs on an Android phone operating on Android 11.0 OS (Oneplus 6T). The application used for controlling the robot is developed on Android Studio using Java, there was an option to use Kotlin as well, but since student B learnt Java in COE 318 (Software Systems, Second Year course), it was much easier to apply it in this case. The Android app applies Bluetooth-Socket programming that enables the use of Classic Bluetooth protocol on the Client (The Android Phone) and the same concept is used on the Server (Raspberry Pi) side as well. Android Studio is a great easy-to-use tool/platform where one can start the development of an Android Application from scratch irrelevant of the amount of experience one has. Furthermore, Bluetooth remains a less complex Personal Area Network that can be used as a means to communicate between two devices. Its lower latency, power consumption and fixed range make it a suitable and reliable protocol on which a communication system can be developed, particularly for a proof-of-concept project like this one. Hence, the 6-legged hexapod is controlled using an Android Application developed on Android Studio using Java and Socket API on the Raspberry Pi processor using Python. The overall interface of the application is shown in the snippet below:

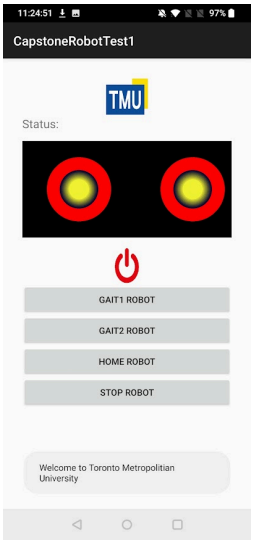


Fig.4.5.1: Application Interface of boot.

Let us discuss a few features or attributes of the application, to begin with, when the app is booted, the user is asked for two permissions: Location and Bluetooth ENABLE. The location permission gives us an additional feature to integrate a GPS or Robot locator feature in the future (not implemented in this project, not required, but it is a possible feature) and the Bluetooth Enable permission gives the user a choice to securely enable the devices Bluetooth for pairing with the server.

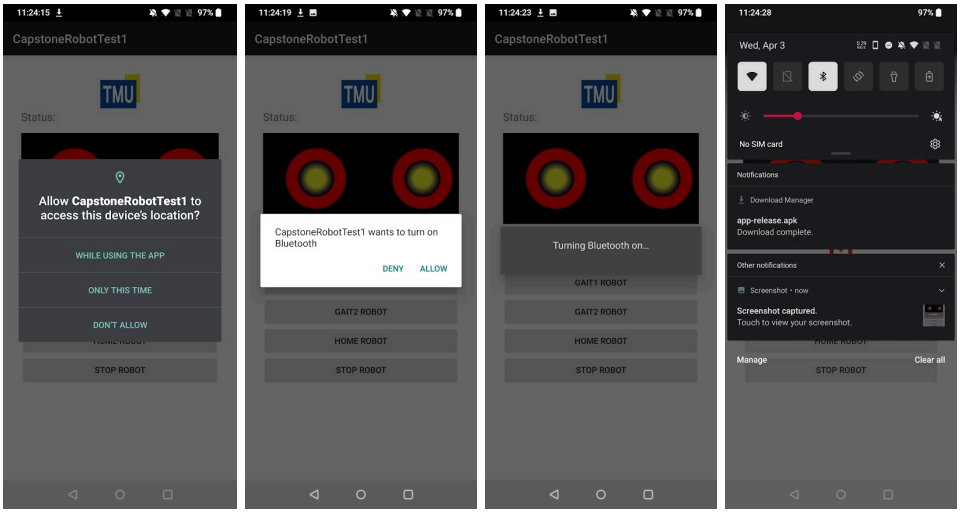


Fig.4.5.2: Location and Bluetooth Permission prompts.

Moving on, there are two joystick triggers, four buttons and one toggle switch on the interface of the application. The LEFT JOYSTICK is used for the movement of the hexapod and the RIGHT JOYSTICK is used for rotating it. The GAIT1 ROBOT and GAIT2 ROBOT buttons enable the robot’s walking modes. HOME ROBOT directs the robot autonomously to head toward the homing base (an alternative is a specific coordinate relative to the surface it is moving on). STOP ROBOT is used to issue an interrupt and stop the robot when in autonomous mode while homing. The toggle switch is used to turn the robot ON or OFF. There is a Status display

feature that displays the current task that the robot is performing so the user is informed at all times. The frontend layout of the buttons and joysticks used in this app can be adjusted upon user preferences as shown in the snippet below:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="30dp"
    tools:context="com.example.capstonerobottest1.MainActivity">

    <Button
        android:id="@+id/tmu"
        android:layout_width="84dp"
        android:layout_height="59dp"
        android:layout_gravity="center_horizontal"
        android:background="@drawable/tmu_logo"
        android:baselineAligned="true"
        android:rotation="0" />

    <TextView
        android:id="@+id/statusTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Status: "
        android:textSize="18sp" />

    <LinearLayout
        android:layout_width="352dp"
        android:layout_height="180dp"
        android:gravity="center"
        android:orientation="horizontal">

        <com.example.capstonerobottest1.JoystickView
            android:id="@+id/joystickLeft"
            android:layout_width="0dp"
            android:layout_height="150dp"
            android:layout_weight="1"
```

Fig.4.5.3: Front End Layout.

An important module that was separately developed and integrated into the project was the joystick/trigger. It uses a special formula which calculates the coordinates of the X-Y plane it rotates in. In this project, the decision was made to keep the Upper and Lower limits between 1 to -1 respectively. As shown in the snippet below, the joystick module uses special Android libraries that have built-in functions to fetch the values of the coordinates on a relatively circular surface. The formula essentially checks if the center of the trigger is at a coordinate value that is lower than the radius of the entire joystick and depending on the conditions, it calculates the x and y coordinates. Below is the formula and code used to develop the joystick module:

```
public boolean onTouch(View v, MotionEvent e)
{
    if(v.equals(this))
    {
        if(e.getAction() != e.ACTION_UP)
        {
            float displacement = (float) Math.sqrt((Math.pow(e.getX() - centerX, 2)) + Math.pow(e.getY() - centerY, 2));
            if(displacement < baseRadius)
            {
                drawJoystick(e.getX(), e.getY());
                joystickCallback.onJoystickMoved((e.getX() - centerX)/baseRadius, (e.getY() - centerY)/baseRadius, getId()); // Gets t
            }
            else
            {
                float ratio = baseRadius / displacement;
                float constrainedX = centerX + (e.getX() - centerX) * ratio;
                float constrainedY = centerY + (e.getY() - centerY) * ratio;
                drawJoystick(constrainedX, constrainedY);
                joystickCallback.onJoystickMoved((constrainedX-centerX)/baseRadius, (constrainedY-centerY)/baseRadius, getId());
            }
        }
        else
            drawJoystick(centerX, centerY);
        joystickCallback.onJoystickMoved(0,0,getId());
    }
    return true;
}
```

Fig.4.5.4: Logic used for developing the Joystick.

The joystick trigger during initial development:

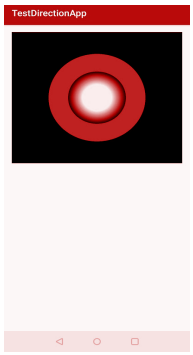


Fig.4.5.5: Joystick module execution.

The application has been developed such that it has the ability to issue feedback in the form of TOAST MESSAGES (messages that pop up on the screen and disappear after 4-5 seconds) in the case where the Bluetooth connection or pairing fails, Bluetooth is not supported by the device or when the data stream is not successfully received by the server upon sending. This ensures the user is informed about any data or packet loss in real-time. As shown in the code snippet below, there are message prompts places considering the worst-case scenarios for both: When the device pairing via Bluetooth fails and when the output stream (data sent out from the application) is not successful.

```
// Method to send data over Bluetooth
private void sendData(String data) {

    if (outputStream != null) {
        try {
            outputStream.write(data.getBytes());
            Log.d("Bluetooth", "Data sent: " + data);
        } catch (IOException e) {
            e.printStackTrace();
            Log.e("Bluetooth", "Error sending data: " + e.getMessage());
            runOnUiThread(new Runnable() {
            @Override
            public void run() {
                Toast.makeText(MainActivity.this, "Error sending data: " + e.getMessage(), Toast.LENGTH_SHORT).show();
                }
            });
        }
    } else {
        Log.e("Bluetooth", "OutputStream is null. Data: '" + data + "', not sent.");
        runOnUiThread(new Runnable() {
        @Override
        public void run() {
            Toast.makeText(MainActivity.this, "Error: OutputStream is null. Data not sent.", Toast.LENGTH_SHORT).show();
            }
        });
    }
}
```

```
            bluetoothSocket.connect();
            outputStream = bluetoothSocket.getOutputStream();
            inputStream = bluetoothSocket.getInputStream();
            startReceivingData();
            Toast.makeText(MainActivity.this, "Bluetooth connected", Toast.LENGTH_SHORT).show();
        } catch (IOException e) {
            e.printStackTrace();
            Toast.makeText(MainActivity.this, "Failed to connect to Bluetooth device", Toast.LENGTH_SHORT).show();
        }
    }

    // Add a timeout mechanism for Bluetooth connection attempts
    new Handler().postDelayed(new Runnable() {
        @Override
        public void run() {
            if (!bluetoothSocket.isConnected()) {
                // Connection attempt failed, notify user or handle accordingly
                Toast.makeText(MainActivity.this, "Failed to connect to Bluetooth device. Please try again.", Toast.LENGTH_SHORT).show();
            }
        }
    }, 10000); // 10 seconds timeout
```

Fig.4.5.6: Java code for Error handling in the Application.

When there is no pairing or connection taking place between the processor and the phone, the app lets the user know about the connection status:

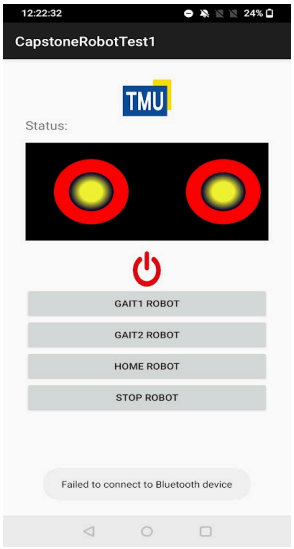


Fig.4.5.7: Message cast when Bluetooth connection fails.

Here is the outcome of the testing of the error feedback mechanism on the emulator:

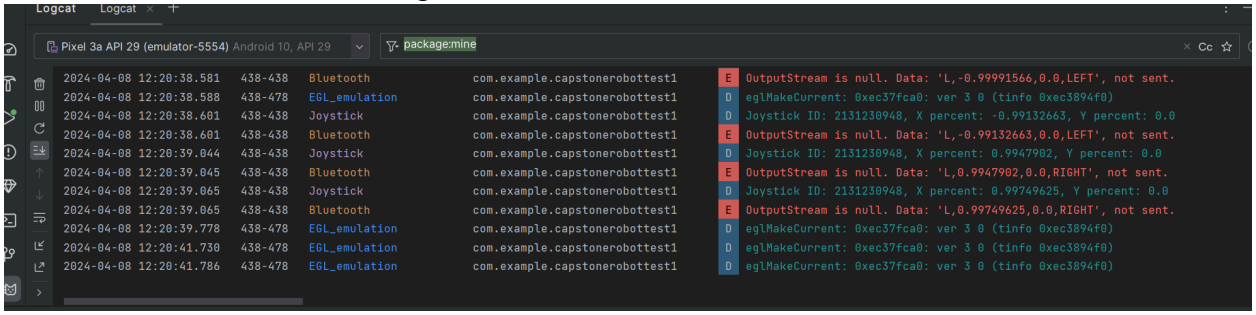


Fig.4.5.8: Terminal view of the Output stream sending out data.

The terminal shows the output stream error messages as there is no end-point device receiving the data sent out, which tells the user that the app is not functional and the application also shows a message for the same.

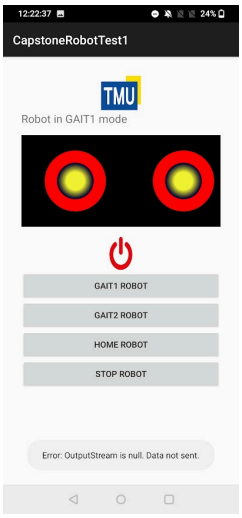


Fig.4.5.9: Message cast when there is no connection and the user tries to send data.

The critical path during the development of the communication system is the understanding and implementation of the bridge between the client and server, which is realized using Socket programming. This communication technique was taught in COE 768 (Computer Networks, Fourth Year course), which gave us hands-on experience in implementing Socket Programming in a TCP/IP environment, however, in this project, student B is implementing the same concept, but a different environment, which is Classic Bluetooth. So after extensive research and analysis, student B was able to configure the Raspberry Pi’s Bluetooth module and successfully send the data from the application to the Raspberry Pi. In the script snippet below, a Socket is initiated based on the Universally Unique Identifier (UUID) which is used to uniquely identify services, characteristics, and profiles within Bluetooth devices and once the connection is successful, the data is received via a socket at port 1024.

```
SERVICE_UUID = "00001101-0000-1000-8000-00805F9B34FB" # Pre-defined UUID
SERVICE_NAME = "BluetoothSerial" #Service/Socket name

def main():
    server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM) #Socket init.
    server_sock.bind(("", bluetooth.PORT_ANY))
    server_sock.listen(1)

    bluetooth.advertise_service(server_sock, SERVICE_NAME,
                                service_id=SERVICE_UUID,
                                service_classes=[bluetooth.SERIAL_PORT_CLASS],
                                profiles=[bluetooth.SERIAL_PORT_PROFILE])

    print("Waiting for connection...")
    client_sock, client_info = server_sock.accept()
    print("Accepted connection from", client_info)

    try:
        while True:
            data = client_sock.recv(1024)
            # print("Received:", data.decode()) ## Commands to be recvd from the
            if len(data) == 0:
                break
```

Fig.4.5.10: Socket Initiation on Raspberry Pi.

On receiving the data, it is processed using If-Else conditional statements and assigned to variables which can be imported into the main source code of the robot.

```
else: #Remove the else if it is causing an error or empty

    decodedData = data.decode()

    splitData = decodedData.split(',')

    whichJoystick = splitData[0].strip() #"R" + "," + xPer

    if whichJoystick == "L":

        xRotate = float(splitData[1])
        # continue

    elif whichJoystick == "R":
        xMove = float(splitData[1])
        if "R" in splitData[2]:
            yMOve = float(splitData[2].replace("R", ""))
        else:
            yMOve = float(splitData[2])
```

Fig.4.5.11: Data processing on Raspberry Pi.

```
mk02@raspberrypi:~/Documents $ sudo python RaspberryPiServer.py
Waiting for connection...
Accepted connection from ('64:A2:F9:EF:31:EC', 1)
Commands Received: GAIT1
Commands Received: GAIT2
Commands Received: HOME
Commands Received: STOP
Leftjoystick X Coordinate: 0.9909585
Leftjoystick X Coordinate: 0.99235755
Leftjoystick X Coordinate: 0.99185747
Leftjoystick X Coordinate: 0.99067384
Leftjoystick X Coordinate: -0.99014455
Leftjoystick X Coordinate: -0.9915608
Leftjoystick X Coordinate: -0.9930834
Leftjoystick X Coordinate: -0.99460244
Leftjoystick X Coordinate: -0.9965645
Leftjoystick X Coordinate: -0.99828196
Leftjoystick X Coordinate: -0.9994541
Leftjoystick X Coordinate: -0.9999769
Leftjoystick X Coordinate: -0.9998907
Leftjoystick X Coordinate: -0.99935925
Leftjoystick X Coordinate: -0.9990087
Leftjoystick X Coordinate: -0.9982364
Leftjoystick X Coordinate: -0.99701583
Leftjoystick X Coordinate: -0.99440044
Right joystick X Coordinate: 0.114285715
None
Right joystick Y Coordinate: 0.114285715
Right joystick X Coordinate: 0.0
None
Right joystick Y Coordinate: 0.0
Right joystick X Coordinate: 0.114285715
None
Right joystick Y Coordinate: 0.114285715
Right joystick X Coordinate: 0.0
None
```

Fig.4.5.12: Data and Commands received by the Raspberry Pi

5. Alternative Designs

5.1. Motion Control Kinematics

5.1.1. Sinusoidal servo control

In the beginning of the process of making a walk cycle through simulation, sinusoidal servo control was used due to the ease of implementation. This was achieved without the use of inverse kinematics as it was not yet imported and properly implemented into the Webots code base that was made. It worked by giving each of the 18 servos a specific sinusoidal function which controlled the angle it had. It was a tripod gait where 3 legs were grouped having the same functions, and the other 3 legs had an offset with similar parameters. An amplitude, phase offset and dc offset defined each servo’s sinusoidal angle function with respect to time, and these parameters were manually tuned. Refer to Figure 4.1.5.1 for an image.

Despite the ease of use, this code lacked flexibility in precise movement without inverse kinematic control of the legs, and though it had a relatively stable walking cycle, the legs swept far out and around which would make collision detection have to use further range and make the robot take up more space as it moved. It was also slow as it wasted time "accelerating" into each movement and decelerating at the end (due to the nature of the sinusoid's shape) which was unnecessary for our capable servos. This design was replaced with a more suitable tripod walking gait once the inverse kinematics code was properly ported over.

5.2. Obstacle Avoidance

5.2.1. Bug2 Algorithm

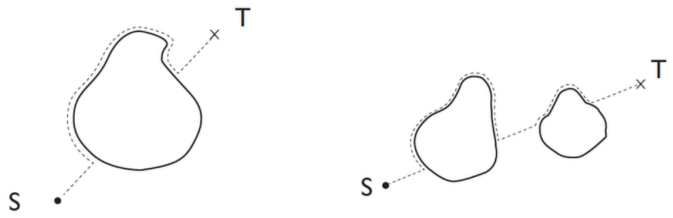


Figure 5.2.1.1: Bug2 Algorithm diagram for Single and Multiple Obstacles. [13]

The Bug2 algorithm provides a simple yet effective approach to autonomous robot navigation around obstacles. Initially, the robot sets out towards its target destination, following a predetermined path or compass direction. Upon encountering an obstacle, it switches to an obstacle avoidance phase. Here, the robot circumvents the obstacle while maintaining its target direction, ensuring it does not stray from its intended path. Once a safe distance from the obstacle is reached, the robot resumes its goal-seeking behavior. If the destination is reached during obstacle avoidance, the robot stops; otherwise, it continues navigating toward the target, iterating through the process until it either reaches its goal or encounters an impassable obstacle. By combining goal-oriented movement with reactive obstacle avoidance, the Bug2 algorithm enables efficient and adaptive navigation in dynamic environments [13]. Even though Bug2 proposes a good solution to the problem, it still has some limitations. Below are two of the main reasons to drop this approach:

Potential for Trapping: In certain scenarios, Bug2 may become trapped in local minima, where it repeatedly encounters obstacles it cannot circumvent. This can lead to inefficient or ineffective navigation, especially in cluttered environments.

Difficulty with Narrow Passages: Bug2 may struggle to navigate narrow passages or corridors, as it lacks mechanisms to efficiently maneuver through tight spaces. This limitation can restrict its applicability in certain environments, such as indoor settings with narrow doorways.

5.2.2. Potential Field Algorithm

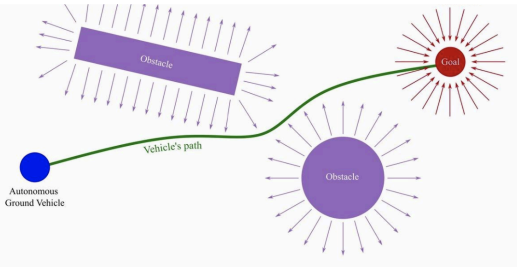


Figure 5.2.2.2: Visualization of Potential Field Algorithm [14]

This algorithm calculates the attractive force pulling the robot towards the goal position and the repulsive force pushing it away from nearby obstacles. The resultant force is then computed as the vector sum of these forces. If the magnitude of the resultant force is above a certain threshold, the robot adjusts its position accordingly. Additionally, collision detection and corrective actions are performed to ensure that the robot avoids obstacles effectively. Adjustments to parameters such as step size, repulsion radius, and force weights may be necessary to optimize the algorithm's performance for your specific robot and environment. [13]

The potential field algorithm turned out to be a very complex approach. The main drawback of our system was the absence of a set goal position, hence no goal vector for the robot to map to, and the ultrasonic sensor’s lack of precision led to issues in making the repulsive force vectors. With the limitations of our system, this approach can only be fruitful if the map of the arena is fed into the system to calculate the path, which is not the goal of the project.

5.3. Guidance System

The design described above has a notable drawback in terms of the time it takes for the robot to reach home, particularly due to its clockwise rotation along its axis, taking approximately 1 second for each step and about 45 seconds for a complete circular rotation in the worst-case scenario. To address this issue, an alternative approach could have been implemented by using a stepper motor to rotate the IR receiver instead of rotating the entire robot. This change could reduce the time to milliseconds. However, this approach presents its own challenges, such as synchronizing the stepper motor speed with the signal detection response time, stopping the motor accurately where the signal is detected, and then reorienting the robot to align with the sensor position. Additionally, integrating extra components like a stepper motor and GPIO extension board would be necessary.

Given these trade-offs, a simpler approach was chosen where the robot rotates about its axis to find the signal. To mitigate the lengthy rotation time for one full cycle, a revised strategy was devised. The robot is permitted to rotate one complete cycle only in steps of 8 degrees in specific scenarios, such as when finding the signal after encountering obstacles or during the initial auto-home function. In regular cases, if the signal is lost, the robot will first rotate +8 degrees to the right. If the signal is still undetected, it will then rotate -8 degrees to the left for

two steps. If the signal remains undetected, the robot will proceed with the complete rotation to detect the signal.

5.4. Wireless Connectivity and App Development

During the entire course of this project, student B developed a demo app over the Fall 2023 semester that gave student B lots of knowledge and experience about the entire process of developing an Application right from the backend layout to the frontend. To a certain point, this exercise showed student B how memory/RAM plays a crucial role when allocated incorrectly as it would not compile and run the app over an emulator

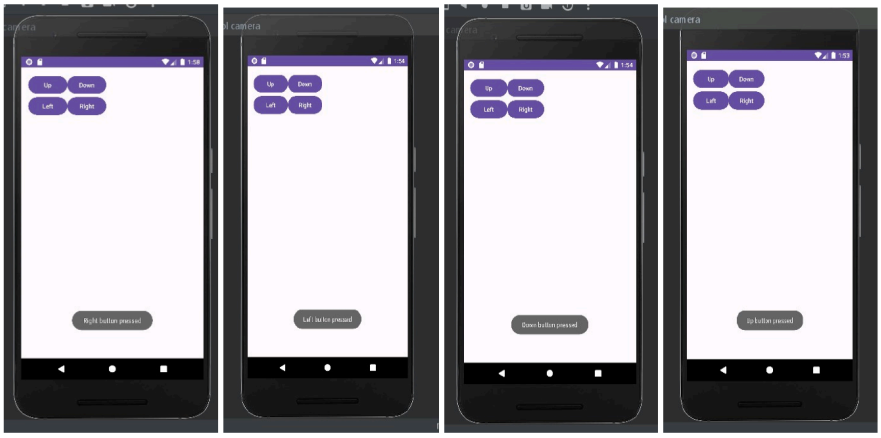


Fig.5.4.1: Sample application design.

Alternative designs for the communication system would have been an application that implements the TCP/IP (Wifi - IEEE 802.11) protocol. The approach would be similar to that of the Bluetooth Android Application as Socket programming is implemented in both protocols. The only downside would be increased complexity and higher latency than Bluetooth as unreliable networks can cause higher ping and longer delays in transmitting or receiving data. However, the data sent out in packets would be received reliably as TCP provides retransmission and error-checking mechanisms.

6. Material/Component list

Table 6.1: Component List

Component	Cost (C$)	Qty
Freenove hexapod Robot kit for Raspberry Pi	214.6	1
VAPCELL (K25 18650 2500MAH)	82.71	8
Raspberry Pi 4B	120.90	1
Battery Charger	38.41	1
Ultrasonic Sensor (HC-SR04)	6.4	2
Oneplus 6T Android	N/A	1

IR transmitters (SFH 4557)	3.21	3
IR transmitters (SFH 4544)	5.65	5
IR receiver (SFH 213 and BPW 24R)	8.87	2
9V battery	7.30	2
Battery connectors	14.68	1
Resistors	11	1
Analog to digital converter	10.56	1

7. Measurement and Testing Procedures

7.1. Motion Control Kinematics

The majority of testing was within a simulation of the robot software Webots. Using this software, simulations were able to quickly set up in a danger-free environment that allowed for a faster development process along with not risking any of the delicate physical parts. After verifying visually that the walk cycle worked as it was intended to, specific movements in the simulation using the custom-made movement methods were tested to ensure a consistent result was obtained. Due to the nature of the simulation being slightly off from reality, verification ended here.

The simulation software used, along with the 3d model that was created was accurate enough so that when it came time to test the robot physically, it worked very similarly, with only some fine-tuning to be done. As for testing physically, it was ensured that the walk cycle and rotation in place functionalities were consistent each time the code was run. The distance or rotation angles of the movement commands were measured at various speeds, angles, and distance parameters and ensured a consistent, linear relationship between the parameters and the actual walking result.

7.2. Hardware and Sensor Technologies

The servo test and calibration are critical steps to ensure that the robot functions correctly. The tutorial describes a structured process to test each servo's operation to verify that they respond accurately to the commands issued by the control system. This involves activating each servo individually to move through its range of motion to confirm that it operates smoothly and aligns with the expected parameters.

The calibration process is particularly important because it ensures that the servos are correctly aligned at their initial positions before full assembly. The robot’s legs need to be at specific angles to start correctly, and the calibration ensures that all servos are set to these starting positions. This setup is crucial for preventing misalignment, which could lead to improper robot movements and potential mechanical stress or damage during operation. This systematic testing and calibration ensure that the robot performs reliably and effectively once fully assembled and operational.

7.3. Obstacle Avoidance

The obstacle avoidance functionality was evaluated under time constraints using a direct upload and execution approach on the Raspberry Pi. Additionally, a bottom-up approach was employed, integrating the walking gait with the robot to navigate the desired environment while simultaneously collecting sensory data. Through iterative testing and analysis of this data, various strategies were tested and refined incrementally. This iterative method allowed for continuous improvement, with adjustments made to the algorithm as progress was made. The process continued until reaching a reasonable milestone, ensuring the development of a robust obstacle avoidance system through systematic refinement and testing.

7.4. Guidance System

After conducting a thorough analysis, a prototype transmitter and receiver circuit were rigorously tested. The transmitter circuit comprised 5 IR transmitters connected with a 25k resistor on a breadboard, powered by a 9-volt battery. The receiver was linked to an Arduino Nano microcontroller, and signal detection was monitored through the microcontroller's input pin. The Red LED was activated upon signal detection, with voltage and current outputs displayed on the Arduino IDE serial monitor. Testing encompassed various factors such as distance range, signal detection angles, different locations within the expected detection area, and the impact of reducing the view angle by encapsulating the receiver in an aluminum foil cylinder. https://drive.google.com/file/d/1druUtQHMLfqNX-Hc6PERg2RCfsA0fAFq/view?usp=drive_link

7.5. Wireless Connectivity and App Development

The application was tested using the in-built Android Studio terminal when the app is compiled during the development and debug stage. Once the client/app was developed, student B tested the application on the physical device that is used to install the application and use it as a robot controller, this makes sure that the application does not crash and is supported by the device. The server side script used for receiving the data was run on the Linux (piOS) OS terminal.

8. Performance Measurement Results

8.1. Motion Control Kinematics

As described in the previous section, from testing within the simulation, the walk cycle and rotate-in-place functionalities worked visually as expected and the movement results between repeated movement commands were consistent with each other. As for testing the physical robot. As described in part 7.1, the distance it traveled or the angle it rotated for various movement commands with differing parameters was measured. After the code was tuned to best work in the real physical environment, it was noticed that the relationship between distance and angle parameters to the actual walking distance and rotation angle was linear and the angle specified matched the actual physical rotation. However, for the speed control, at the highest

levels of speed, where instead of making larger strides (like the distance parameter), the rate of the movement of the legs would increase, some sliding and skidding behaviour was noticed which was slightly inaccurate and less stable than expected.

8.2. Hardware and Sensor Technologies

The performance of these tests described earlier was successful and consistent with the expected results. This step marked the correct assembly of the robot. It was a very crucial step as without it, the exact mapping of the new gait would not be possible.

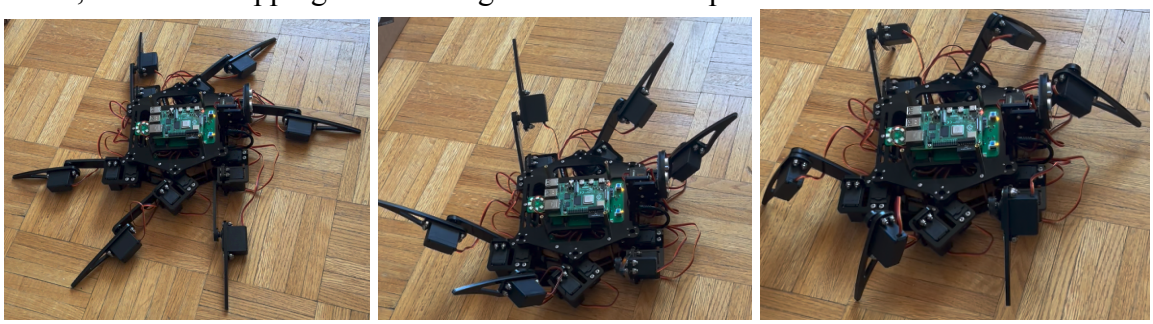


Figure 8.2.1: Successful Results of the Servo Test

8.3. Obstacle Avoidance

As mentioned, there are two successful approaches for obstacle avoidance that can traverse and avoid some basic obstacles. The images below will showcase the implementation of both approaches integrated with the tripod gait designed by us.

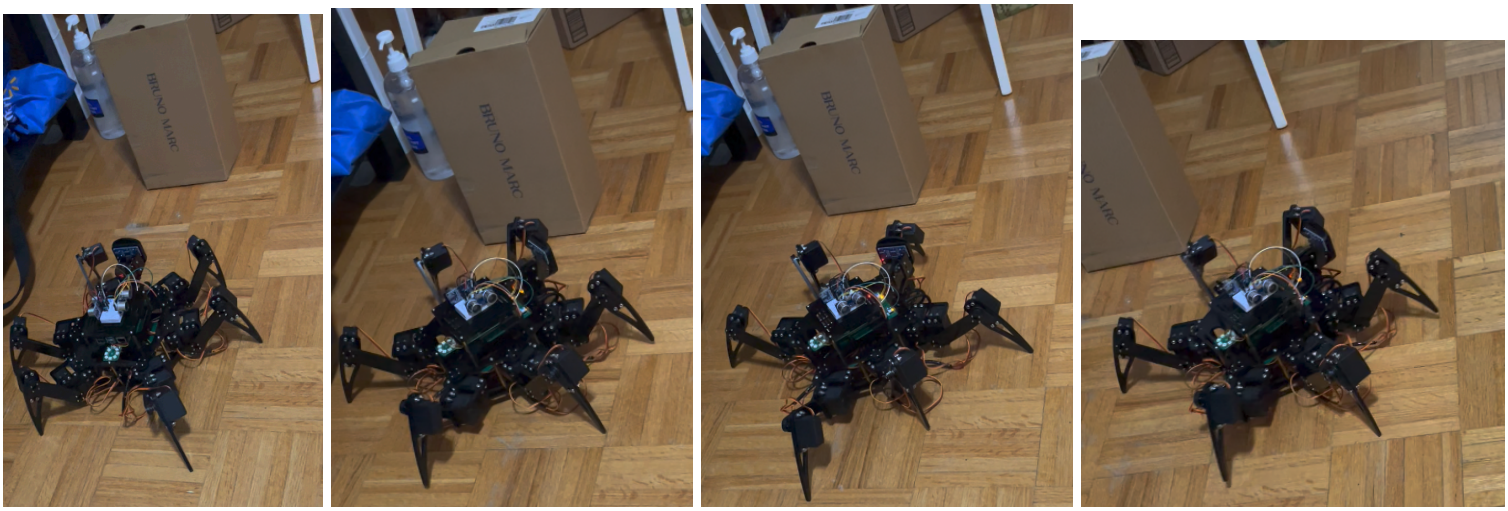


Figure 8.3.1: Results of Obstacle avoidance using **State Machine Approach**

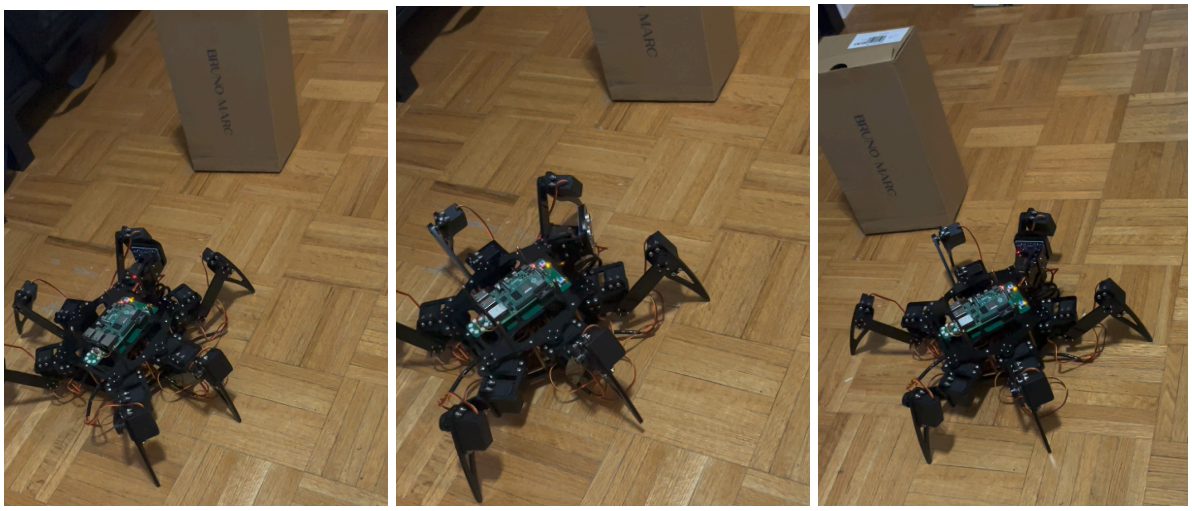


Figure 8.3.2: Results of Obstacle avoidance using **Reactive Navigation Approach**

8.4. Guidance System

During testing, the effective distance range achieved was approximately 1.7 meters. The transmitters were precisely angled to cover a 90-degree area. Initially, the receiver exhibited high directionality when positioned farther from the transmitters. However, as it approached closer, the signal intensity increased, and its directionality diminished, allowing detection even at a 40-degree view angle. Cylindrical aluminum foil is used to reduce the view angle to +/- 8 degrees even when the signal intensity is very high. the outcomes can be seen in the video link below.

https://drive.google.com/file/d/14oas4RTfkMkVXVUhrxmta9OAWM48-gkb/view?usp=drive_link

8.5. Wireless Connectivity and App Development

The application was tested using an in-built Android Studio IDE emulator that emulates Google Pixel 3a (Operating System/Kernel: Android 7.0) as well as tested on a physical device: Oneplus 6T (Operating System/Kernel: Android 11.0) to make sure the application is supported by a wide range of Android OS versions. The emulator does not provide Bluetooth to test the application so using a physical device was a requirement. Initially, the application was tested by sending out data as seen in the snippet below, the data that is transmitted, is printed on the Android Studio IDE terminal:

Once the client side was successfully tested, student B began developing the server side, the Raspberry Pi 4B+ supports Python as its main language of user program, so student B scripted a socket program that would verify if the connection is successful and with the correct device and then open a socket to receive data from the Android application. Once it receives the data, it is further processed and integrated into the robot’s walking and homing mechanisms.

9. Analysis of Performance

9.1. Motion Control Kinematics

The objective was to have a consistent and “visibly works as intended” result for our simulation, as any fine-tuning and verification is critical for the physical environment. So for simulation, we were able to achieve our objective. However, for the physical environment, it was noticed that the highest levels of speed control led to a skidding motion that was slightly less stable. This is due to a limitation of the precision of the servos operating at the highest speed that was designed, which was not reflected by the “ideal servos” in the simulation software that was used.

The reasonable solution is simply to not use these flawed absolute highest speed settings and bound the maximum speed to about 80% of what was designed instead. This is acceptable as this project is a proof of concept, where speed constrained by servo motor selection is not the critical issue. Precision and linearity of the distance and angle controls which allow for precise maneuvering are much more critical for the autonomous navigation purposes of our project.

9.2. Hardware, Sensor Technologies, and Obstacle Avoidance Control

The performance analysis of the control hardware and sensor technologies confirms their successful integration and operation within the robot system. Testing procedures, including servo testing and calibration, ensured accurate responses to control commands and proper alignment of mechanical components. This approach guarantees reliable and effective performance, laying the foundation for further development of the obstacle avoidance control system.

The obstacle avoidance performance analysis utilizing ultrasonic sensors demonstrated overall success in detecting and navigating basic obstacles within the environment. The ultrasonic sensors effectively provided proximity data, enabling the robot to identify and circumvent obstacles such as walls and large objects with consistent accuracy. However, limitations of ultrasonic sensors were evident in certain edge cases, such as detecting small or low-lying objects or accurately sensing obstacles at acute angles. In these scenarios, where obstacles are positioned diagonally or at sharp angles relative to the robot's direction of movement, ultrasonic sensors may experience reduced effectiveness due to their narrow detection cones. Integration of additional sensors, such as Sharp IR or Lidar, could significantly enhance the design by overcoming these limitations. Sharp IR sensors excel in detecting smaller objects and obstacles at acute angles, while Lidar offers high-resolution 3D mapping capabilities, providing a more comprehensive understanding of the robot's surroundings. Incorporating these sensor technologies would offer a more robust obstacle avoidance system capable of navigating a wider range of environmental conditions with greater precision and reliability.

9.3. Guidance System

The distance range is a little shorter than what was desired. This range is slightly reduced due to environmental factors. It could be because the ADC resolution is 10 bit which can only read a voltage of 4.88mV or higher incorporating amplifiers can help us increase the signal detection by amplifying the voltage. Also, it can be increased by increasing the power output of the transmitters. Also to overcome the directivity issue the receiver is encapsulated in the aluminum foil cylinder with the front end open. It helps to shield the unwanted signal from the sides and only allows the signal to be detected from the front side.

9.4. Wireless Connectivity and App Development

The Android application was successfully implemented and integrated with the project, as the client and server sides of the Bluetooth connection were successfully communicating with each other. On receiving the data, it is processed and fed into the robot’s walking and homing mechanism which instructs the robot to move according to the coordinates it receives in real time. However, there is a slight noticeable delay that occurs when the processor receives the data, processes the received data and performs the action. This delay occurs mainly due to network overhead as there is a lot of data being sent from the app especially when the user is moving a joystick (several X-Y coordinates are sent due to the sensitive nature of the joystick triggers). This delay due to transactional overhead was solved by adding a 2000 ms delay between each command sent from the app this way, the robot gets time to complete its action within the specified clock cycle and smoothly transition to the next received command. Further optimizations can include improving the processor, improving code efficiency and buffering received data instead of directly processing it. Below is a snippet showcasing the functionality of the application:

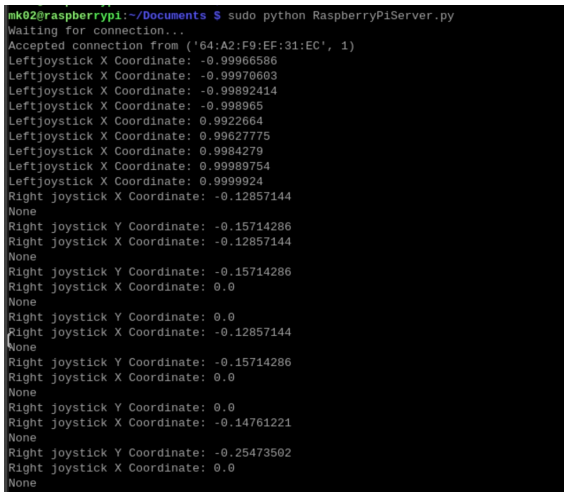


Fig.9.5.1: X-Y Coordinates on each joystick received by the Raspberry Pi

10. Conclusions

The BEAMstiquito project has successfully achieved its core aim of developing an autonomous hexapod robot by integrating innovative BEAM principles. The project resulted in the creation of a robot capable of navigating complex terrains and performing various tasks autonomously. Key accomplishments include the development of an efficient walking gait, the implementation of advanced sensor technologies for navigation and obstacle avoidance, and the establishment of a dual-mode guidance system. Additionally, the project successfully developed an Android app and Bluetooth communication, enhancing the robot's control and user interaction capabilities.

While the project met most of its initial objectives, there were some discrepancies in the extent of autonomy achieved. The initial goal was for the robot to operate with complete independence in navigating to the charging base and performing tasks. However, some aspects of the robot's autonomous functions, such as precision in obstacle avoidance and the effectiveness of the guidance system in complex scenarios, fell short of expectations. The robot demonstrated a high level of autonomy, but its dependence on predefined conditions highlighted areas needing improvement.

One of the major unresolved features of this project was implementing various sophisticated methods for traversal beyond going to points a and b, and rotation in place, such as navigation over bumpy terrain or climbing obstacles and hills effectively. Methods like observing the offset between the measured servo angle and the intended servo angle could have been possible solutions to account for and climb over small obstacles. Additionally, integrating sensor feedback into real-time navigation proved challenging, as latency issues occasionally affected the robot's ability to respond promptly to dynamic environmental changes.

Future work on the BEAMstiquito project will focus on enhancing the robot's stability at higher speeds by optimizing the mechanical design and refining the control algorithms. Improvements in sensor integration and processing power will be pursued to reduce latency and increase the robot’s responsiveness to environmental changes. Further development of the Android app will include full integration of the app and adding more features as per the enhancement of the system. Finally, expanding the robot’s capabilities to handle more complex tasks and environments will be a key area of development, potentially incorporating machine learning techniques for better decision-making and adaptation.

This conclusion concisely covers what has been completed, identifies discrepancies, acknowledges unresolved challenges, and outlines future directions for the project.

References

[1] Hexapod Kit, F. (n.d.). *Freenove/freenove_big_hexapod_robot_kit_for_raspberry_pi: Apply to FNK0052*. GitHub. https://github.com/Freenove/Freenove_Big_Hexapod_Robot_Kit_for_Raspberry_Pi

[2] Kassam, M. (2023, September 3). *MK02: BEAMstiquito ("The Insect") Walking Robot* . D2l. https://courses.torontomu.ca/d2l/le/content/794625/viewContent/5317813/View

[3] Wikimedia Foundation. (2023, December 2). *Beam robotics*. Wikipedia. https://en.wikipedia.org/wiki/BEAM_robotics

[4] Milestone Compliance Reports Fall [1-4]

[5] Milestone Compliance Reports Winter [1-4]

[6] Automatic level crossing system - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Timing-diagram-of-HC-SR04-5_fig13_325472568 [accessed 12 Apr, 2024]

[7] Data sheet provided with [1] (PCA9685 and MPU6050 Modules)

[8] *SunFounder digital accelerometer ADXL345 module compatible with Arduino and Raspberry Pi*. Amazon.ca: Electronics. (n.d.). https://www.amazon.ca/SunFounder-Digital-Accelerometer-ADXL345-Raspberry/dp/B0151FIBZO/ref=sr_1_3?crid=20MWWW1WQ2U5V&keywords=MPU650%2B6&qid=1701728518&s=electronics&sprefix=mpu650%2B6%2Celectronics%2C66&sr=1-3

[9] *5pcs HC-SR04 ultrasonic sensor, distance sensor with ultrasonic transmitter and receiver module fit Arduino Uno mega2560 nano robot xbee zigbee : Amazon.ca: Industrial & Scientific*. 5pcs HC-SR04 Ultrasonic Sensor, Distance Sensor with Ultrasonic Transmitter and Receiver Module fit Arduino UNO MEGA2560 Nano Robot XBee ZigBee : Amazon.ca: Industrial & Scientific. (n.d.). https://www.amazon.ca/Ultrasonic-Distance-Transmitter-Receiver-MEGA2560/dp/B07PFCVM9D/ref=sr_1_2_sspa?crid=398KWJP7D2GTE&keywords=ultrasonic%2Bsensor&qid=1701728651&s=electronics&sprefix=ultrasonic%2Bsensro%2Celectronics%2C76&sr=1-2-spons&sp_csd=d2lkZ2V0TmFtZT1zcF9hdGY&psc=1

[10] *Raspberry pi 4 model B 2019 Quad Core 64 bit WIFI bluetooth (4GB) : Amazon.ca: Electronics*. Raspberry Pi 4 Model B 2019 Quad Core 64 Bit WiFi Bluetooth (4GB) : Amazon.ca: Electronics. (n.d.). https://www.amazon.ca/Raspberry-Model-2019-Quad-Bluetooth/dp/B07TC2BK1X/ref=sr_1_3?crid=EZQUWHFPZEXY&keywords=raspberry%2Bpi%2B4&qid=1701728748&s=electronics&sprefix=raspberry%2Celectronics%2C86&sr=1-3&ufe=app_do%3Aamzn1.fos.71722c10-739d-471b-befb-3e4b9bf7d0d6

[11] ams-OSRAM USA Inc., "SFH-4544," Digi-Key Electronics, 2024. [Online]. Available: https://www.digikey.ca/en/products/detail/ams-osram-usa-inc/SFH-4544/5231484.

[12] ams-OSRAM USA Inc., "SFH-213-FA," Digi-Key Electronics, 2024. [Online]. Available: https://www.digikey.ca/en/products/detail/ams-osram-usa-inc/SFH-213-FA/2205882.

[13] Hosseini Rostami, Seyyed Mohammad & Kumar, Arun & Wang, Jin & Liu, Xiaozhu. (2019). Obstacle avoidance of mobile robots using modified artificial potential field algorithm. EURASIP Journal on Wireless Communications and Networking. 2019. 10.1186/s13638-019-1396-2.

[14] "Predictive Artificial Potential Field algorithm - energy-efficient local path planning algorithm," Online video. Available: https://www.youtube.com/watch?app=desktop&v=FJSlUPzLjqQ. [Accessed: April 12, 2024].
[15] "Raspberry Pi 4 Model B," Raspberry Pi Foundation, [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/. [Accessed: April 29th, 2024].

Appendices

The two solutions for the autonomous navigation system

State Machine Based Approach

```
# Import the Ultrasonic class from the Ultrasonic module
from Ultrasonic import Ultrasonic

# Import everything in the Control module
from Control import *

from enum import Enum

# Import the Servo class and time module
from Servo import Servo
import time
#for concurrent operations
import threading

# Create instances of the Ultrasonic class for each sensor
sensor1 = Ultrasonic(trigger_pin=27, echo_pin=22)
sensor2 = Ultrasonic(trigger_pin=25, echo_pin=8)
sensor3 = Ultrasonic(trigger_pin=7, echo_pin=1)

# Creating an object 'control' of 'Control' class
c = Control()

# Create an instance of the Servo class
s = Servo()

# Define the obstacle detection threshold
obstacle_threshold_front = 20 # in cm
obstacle_threshold_mid_left = 30
obstacle_threshold_left = 25
obstacle_threshold_mid_right = 30
obstacle_threshold_right = 25

# Define a state for the robot's behavior
class State(Enum):
    MOVE_FORWARD = 1
    DETECT_OBSTACLE = 2
```

```
    FOLLOW_OBSTACLE = 3
    CHECK_CLEAR_PATH = 4

# Define the head movement parameters
head_positions = [180, 145, 100]
head_delay = 0.6
D = 2
CD = 2
V =1
z=0

s.setServoAngle(1, 133)
time.sleep(0.5)

# Define a function to check if the path is clear
def is_path_clear():
    distance = sensor1.getDistance()
    return distance > obstacle_threshold_front

def L_R():
     c.rotateInPlace(-10)

def R_R():
    c.rotateInPlace(10)

# Define the move_forward function
def move_forward():
        c.walk(0, 1, 8)

# Define the move_left function
def move_left():
    c.walk(-1,0,8)

# Define the move_right function
def move_right():
     c.walk(1,0,8)

# Define the move_backward function
def move_backward():
     c.walk(0,-1,8)

def sweep_head(positions):
```

```
    distances = {}
    for angle in positions:
        s.setServoAngle(1, angle)
        time.sleep(head_delay)
        distances[angle] = sensor1.getDistance()
    return distances

def decide_direction():
    distance_front = sensor1.getDistance()
    if distance_front > obstacle_threshold_front:
        return 'forward'
    else:
        s.setServoAngle(1,100)
        time.sleep(0.5)
        distance_right = sensor1.getDistance()

        s.setServoAngle(1,180)
        time.sleep(0.5)
        distance_left = sensor1.getDistance()

        s.setServoAngle(1,133)

        if distance_left > distance_right and distance_left > obstacle_threshold_front:
            return 'left'
                    elif   distance_right   >   distance_left   and   distance_right   >
obstacle_threshold_front:
            return 'right'
        else:
             # If none of the conditions are satisfied, rotate right until a condition
is met
            R_R()
            # Recursively call decide_direction
            return decide_direction()

# Initialize the state
state = State.MOVE_FORWARD

# Initialize a variable to keep track of the boundary following direction
boundary_follow_direction = None

# Modify the main loop to include state-based logic
while True:
```

```
    if state == State.MOVE_FORWARD:
        # Move forward if the path is clear
        if is_path_clear():
            move_forward()
            time.sleep(0.2)
        else:
            # If we detect an obstacle, switch to DETECT_OBSTACLE state
            state = State.DETECT_OBSTACLE

    elif state == State.DETECT_OBSTACLE:
        direction = decide_direction()
        print("Direction Decision", direction)
        if direction == 'forward':
            move_forward()
        else:
            state = State.FOLLOW_OBSTACLE

    elif state == State.FOLLOW_OBSTACLE:
        if direction == 'left':
            # Rotate right until clear path to the right is found
            while not is_path_clear():
                        print("Dist_Front  :  Threshold", sensor1.getDistance(), ":",
obstacle_threshold_front)
                L_R()
                time.sleep(0.2)

            s.setServoAngle(1,145)
            while sensor1.getDistance() <= obstacle_threshold_mid_right:
                   print("Dist_mid_right  :  Threshold Mid right", sensor1.getDistance(),
":", obstacle_threshold_mid_left)
                move_left()
                print("Move_Left")

            while sensor3.getDistance() <= obstacle_threshold_left:
                      print("Dist_left  :  Threshold  Right", sensor2.getDistance(), ":",
obstacle_threshold_right)
                move_left()
                print("Move_Right")

            s.setServoAngle(1, 133)
            state = state.CHECK_CLEAR_PATH
```

```
        elif direction == 'right':
            # Rotate right until clear path to the right is found
            while not is_path_clear():
                        print("Dist_Front  :  Threshold", sensor1.getDistance(), ":",
obstacle_threshold_front)
                R_R()
                time.sleep(0.2)

            s.setServoAngle(1,180)
            while sensor1.getDistance() <= obstacle_threshold_mid_left:
                 print("Dist_mid_left : Threshold Mid Left", sensor1.getDistance(), ":",
obstacle_threshold_mid_left)
                move_right()
                print("Move_Right")

            while sensor2.getDistance() <= obstacle_threshold_left:
                      print("Dist_left  :  Threshold  Left", sensor2.getDistance(), ":",
obstacle_threshold_left)
                move_right()
                print("Move_Right")

            s.setServoAngle(1, 133)
               state = state.CHECK_CLEAR_PATH  # Move forward once both front and left
sensors are clear

        else:
            move_backward()
               state = state.CHECK_CLEAR_PATH  # Move forward once both front and left
sensors are clear

        time.sleep(0.2)  # Adjust as needed

    elif state == State.CHECK_CLEAR_PATH:
        # If after moving forward we're still clear, go to MOVE_FORWARD state
        if is_path_clear():
            state = State.MOVE_FORWARD
            print("state Move Forward")
        else:
            state = State.FOLLOW_OBSTACLE

    # Add a small delay to the loop to prevent excessive CPU usage
    time.sleep(0.1)
```

Reactive Navigation

```
# Import the Ultrasonic class from the Ultrasonic module
from Ultrasonic import Ultrasonic

# Import everything in the Control module
from Control import *

# Import the Servo class and time module
from Servo import Servo
import time
#for concurrent operations
import threading

# Create instances of the Ultrasonic class for each sensor
sensor1 = Ultrasonic(trigger_pin=27, echo_pin=22)
sensor2 = Ultrasonic(trigger_pin=25, echo_pin=8)
sensor3 = Ultrasonic(trigger_pin=7, echo_pin=1)

# Creating an object 'control' of 'Control' class
c = Control()

# Create an instance of the Servo class
s = Servo()

# Define the obstacle detection threshold
obstacle_threshold = 41 # in cm

# Define the head movement parameters
head_positions = [180, 133, 100]
head_delay = 0.6
D = 2
CD = 2
V =1
z=0

s.setServoAngle(1, 133)
time.sleep(0.5)

def R_R():
   c.rotateInPlace(10)
```

```
# Define the move_forward function
def move_forward():
   c.walk(0,1,8)

# Define the move_left function
def move_left():
   c.walk(-1,0,8)

# Define the move_right function
def move_right():
    c.walk(1,0,8)

# Define the move_backward function
def move_backward():
    c.walk(0,-1,8)

distance_left = 0
distance_center = 0
distance_right = 0

while True:

   if CD == 1:
       s.setServoAngle(1, 180)
       distance_left = sensor1.getDistance()
       time.sleep(head_delay)
       CD = 2

   elif CD == 2:
       s.setServoAngle(1, 133)
       distance_center = sensor1.getDistance()
       time.sleep(head_delay)
       CD = 3

   elif CD == 3:
       s.setServoAngle(1, 100)
       distance_right = sensor1.getDistance()
       time.sleep(head_delay)
       CD = 1
   else:
       time.sleep(0.2)
```

```
   time.sleep(head_delay)

   if distance_left > obstacle_threshold and distance_center > obstacle_threshold and
distance_right > obstacle_threshold:
       # If all distances are above threshold, move forward
       move_forward()
   else:
       # If any distance is below threshold, rotate right
       R_R()
       time.sleep(0.2)
```

The application code

>MainActivity.java:

```
// Import Android libraries and classes
package com.example.capstonerobottest1;

import android.os.Handler;
import android.Manifest;
import android.bluetooth.BluetoothServerSocket;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.content.pm.PackageManager;
import android.os.Build;
import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.Button;
import android.widget.TextView;
import android.widget.ListView;
import android.widget.Toast;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import androidx.core.app.ActivityCompat;
import android.widget.EditText;
import android.widget.AdapterView;
import android.os.Message;

import android.widget.ArrayAdapter;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Set;
import java.util.UUID;
```

```
public class MainActivity extends AppCompatActivity implements JoystickView.JoystickListener {

    private static final int PERMISSION_REQUEST_CODE = 1001;// Permission Request Variable assignment

   // Bluetooth programming variables
   private BluetoothAdapter bluetoothAdapter;
   private BluetoothSocket bluetoothSocket;
   private OutputStream outputStream;
   private InputStream inputStream;
            private Button robotGOButton, robotSTOPButton, robotHOMEButton, robotGAIT1Button, robotGAIT2Button, powerButton, tmuButton;//UI Elements
   private TextView statusTextView;
   private boolean isPowerOn = false;

   @Override
    protected void onCreate(Bundle savedInstanceState) { // onCreate method called when activity is first created
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
         // Check for Bluetooth permissions
            if (checkSelfPermission(Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED) { //Location can be used for getting access to the Raspberry Pi's Location.
            // Permission not granted, request it
                                    ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, PERMISSION_REQUEST_CODE);
         } else {
            // Permission granted, initialize Bluetooth
            initializeBluetooth();
         }
      } else {
            // Runtime permission not needed for versions below Marshmallow, initialize Bluetooth directly
         initializeBluetooth();
      }

      robotGAIT1Button = findViewById(R.id.gait1); // Initialize UI elements
      robotGAIT2Button = findViewById(R.id.gait2);
      robotSTOPButton = findViewById(R.id.stop);
      robotHOMEButton = findViewById(R.id.home);
      statusTextView = findViewById(R.id.statusTextView);
      powerButton = findViewById(R.id.power_button);
      tmuButton = findViewById(R.id.tmu);

      // Define click listeners for buttons
      powerButton.setOnClickListener(new View.OnClickListener() {
```

```
        @Override
        public void onClick(View v) {
            try{
                // Toggle the power state
                isPowerOn = !isPowerOn;
                if (isPowerOn) {
                    sendData("POWER ON");
                                    Toast.makeText(MainActivity.this, "Robot is powered ON",
Toast.LENGTH_SHORT).show();
                } else {
                    sendData("POWER OFF");
                                    Toast.makeText(MainActivity.this, "Robot is powered OFF",
Toast.LENGTH_SHORT).show();
                }
            }catch (Exception e) {
                    e.printStackTrace();
                                Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
            }
        }
    });

    tmuButton.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            try{
                        Toast.makeText(MainActivity.this, "Welcome to Toronto Metropolitian
University", Toast.LENGTH_LONG).show();
            }catch (Exception e) {
                e.printStackTrace();
                            Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_LONG).show();
            }
        }
    });

    robotGAIT1Button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            try {
                Log.d("Button Click", "GAIT1 Robot button clicked");
                statusTextView.setText("Robot in GAIT1 mode");
                sendData("GAIT1");
                        Toast.makeText(MainActivity.this, "Robot is in GAIT 1 Mode",
Toast.LENGTH_SHORT).show();
            } catch (Exception e) {
                e.printStackTrace();
                                Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
            }
        }
    });
```

```
        robotGAIT2Button.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                try {
                    Log.d("Button Click", "GAIT2 Robot button clicked");
                    statusTextView.setText("Robot in GAIT2 mode");
                    sendData("GAIT2");
                                Toast.makeText(MainActivity.this, "Robot is in GAIT 2 Mode",
Toast.LENGTH_SHORT).show();
                } catch (Exception e) {
                    e.printStackTrace();
                                    Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
                }
            }
        });

        robotHOMEButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                try{
                    Log.d("Button Click", "HOME button clicked");
                    statusTextView.setText("Robot is Homing");
                    sendData("HOME");
                      Toast.makeText(MainActivity.this, "Robot is Homing, Semi-Autonomous mode
activated", Toast.LENGTH_SHORT).show();

                } catch (Exception e) {
                    e.printStackTrace();
                                    Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
                }
            }
        });

        robotSTOPButton.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                try{
                    Log.d("Button Click", "STOP button clicked");
                    statusTextView.setText("Robot stopped");
                    sendData("STOP");
                         Toast.makeText(MainActivity.this, "Interrupt Requested, Robot is stopping",
Toast.LENGTH_SHORT).show();
                } catch (Exception e) {
                    e.printStackTrace();
                                    Toast.makeText(MainActivity.this, "Error: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
                }
            }
        });
    }
```

```
    @Override
    public void onJoystickMoved(float xPercent, float yPercent, int id) {

        // Check the ID to determine which joystick is triggered
        if (id == JoystickView.JOYSTICK_LEFT_ID) {
            if (xPercent >= -1.0 && xPercent <= -0.99){
                yPercent = 0;
                 Log.d("Joystick", "Joystick ID: " + id + ", X percent: " + xPercent + ", Y percent: "
+ yPercent);
                statusTextView.setText("Manual Mode - Robot is Rotating Left");
                sendData("L" + "," + xPercent + "," + yPercent + "," + "LEFT");
            }
            else if (xPercent <= +1.0 && xPercent >= +0.99){
                yPercent = 0;
                 Log.d("Joystick", "Joystick ID: " + id + ", X percent: " + xPercent + ", Y percent: "
+ yPercent);
                statusTextView.setText("Manual Mode - Robot is Rotating Right");
                sendData("L" + "," + xPercent + "," + yPercent + "," + "RIGHT");
            }
        } else if (id == JoystickView.JOYSTICK_RIGHT_ID) {
            statusTextView.setText("Manual Mode - Robot in Motion");
             Log.d("Joystick", "Joystick ID: " + id + ", X percent: " + xPercent + ", Y percent: " +
yPercent);
            sendData("R" + "," + xPercent + "," + yPercent + "," + "MOVE");
        }
    }



    private void initializeBluetooth() {
        bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();

        // Check if Bluetooth is supported on the device
        if (bluetoothAdapter == null) {
                            Toast.makeText(this, "Bluetooth is not supported on this device",
Toast.LENGTH_SHORT).show(); // Handle the case where Bluetooth is not supported
            return;
        }

        // Enable Bluetooth
        if (!bluetoothAdapter.isEnabled()) {
                                                    Intent enableBtIntent = new
Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
            startActivityForResult(enableBtIntent, 1);
        }

        // Get paired Bluetooth devices
        Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();

        //RFCOMM Service to connect to Raspberry Pi
        if (pairedDevices.size() > 0) {
             String deviceAddress = "D8:3A:DD:5C:C6:F5"; // Replace with Raspberry Pi's MAC
address
```

```
        BluetoothDevice device = bluetoothAdapter.getRemoteDevice(deviceAddress);
            UUID uuid = UUID.fromString("00001101-0000-1000-8000-00805F9B34FB"); //
BluetoothSocket using a UUID

        try {
            // Create a socket and connect
            bluetoothSocket = device.createRfcommSocketToServiceRecord(uuid);
            bluetoothSocket.connect();
            outputStream = bluetoothSocket.getOutputStream();
            inputStream = bluetoothSocket.getInputStream();
            startReceivingData();
                                    Toast.makeText(MainActivity.this,  "Bluetooth  connected",
Toast.LENGTH_SHORT).show();
        } catch (IOException e) {
            e.printStackTrace();
                    Toast.makeText(MainActivity.this, "Failed to connect to Bluetooth device",
Toast.LENGTH_SHORT).show();
        }
      }

      // Add a timeout mechanism for Bluetooth connection attempts
      new Handler().postDelayed(new Runnable() {
        @Override
        public void run() {
            if (!bluetoothSocket.isConnected()) {
                // Connection attempt failed, notify user or handle accordingly
                Toast.makeText(MainActivity.this, "Failed to connect to Bluetooth device. Please
try again.", Toast.LENGTH_SHORT).show();
            }
        }
      }, 10000); // 10 seconds timeout
   }

   // Method to send data over Bluetooth
   private void sendData(String data) {

      if (outputStream != null) {
        try {
            outputStream.write(data.getBytes());
            Log.d("Bluetooth", "Data sent: " + data);
        } catch (IOException e) {
            e.printStackTrace();
            Log.e("Bluetooth", "Error sending data: " + e.getMessage());
            runOnUiThread(new Runnable() {
            @Override
            public void run() {
                    Toast.makeText(MainActivity.this, "Error sending data: " + e.getMessage(),
Toast.LENGTH_SHORT).show();
                }
            });
        }
      } else {
        Log.e("Bluetooth", "OutputStream is null. Data: '" + data + "', not sent.");
```

```
            runOnUiThread(new Runnable() {
            @Override
            public void run() {
                    Toast.makeText(MainActivity.this, "Error: OutputStream is null. Data not sent.",
Toast.LENGTH_SHORT).show();
                }
            });
        }
    }

    // Function to read/recv from Raspberry Pi
    private void startReceivingData() {
        new Thread(new Runnable() {
            @Override
            public void run() {
                byte[] buffer = new byte[1024]; // Array to contain message and number of bytes
                int bytes;

                while (true) {
                    try {
                        bytes = inputStream.read(buffer);
                        String incomingMessage = new String(buffer, 0, bytes);
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
                                // WIP Can add feature to read the recving data from the Pi
                            }
                        });
                    } catch (IOException e) {
                        e.printStackTrace();
                        break;
                    }
                }
            }
        }).start();
    }

    @Override
    protected void onDestroy() { //Close Bluetooth Socket
        super.onDestroy();
        try {
            if (bluetoothSocket != null) {
                bluetoothSocket.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onRequestPermissionsResult(int requestCode, @NonNull String[] permissions,
@NonNull int[] grantResults) { // Method to handle permission request result
        super.onRequestPermissionsResult(requestCode, permissions, grantResults);
```

```
    if (requestCode == PERMISSION_REQUEST_CODE) {
                                if (grantResults.length > 0 && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
            // Permission granted, initialize Bluetooth
            initializeBluetooth();
        } else {
            // Permission denied, notify user
                    Toast.makeText(this, "Permission denied, Bluetooth functionality disabled",
Toast.LENGTH_SHORT).show();
        }
    }
}
```

> activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="30dp"
    tools:context="com.example.capstonerobottest1.MainActivity">

    <Button
        android:id="@+id/tmu"
        android:layout_width="84dp"
        android:layout_height="59dp"
        android:layout_gravity="center_horizontal"
        android:background="@drawable/tmu_logo"
        android:baselineAligned="true"
        android:rotation="0" />

    <TextView
        android:id="@+id/statusTextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Status: "
        android:textSize="18sp" />

    <LinearLayout
        android:layout_width="352dp"
        android:layout_height="180dp"
        android:gravity="center"
        android:orientation="horizontal">

        <com.example.capstonerobottest1.JoystickView
            android:id="@+id/joystickLeft"
            android:layout_width="0dp"
            android:layout_height="150dp"
```

```
        android:layout_weight="1"
        android:gravity="center" />

    <com.example.capstonerobottest1.JoystickView
        android:id="@+id/joystickRight"
        android:layout_width="0dp"
        android:layout_height="150dp"
        android:layout_weight="1"
        android:gravity="center" />
</LinearLayout>

<Button
    android:id="@+id/power_button"
    android:layout_width="55dp"
    android:layout_height="57dp"
    android:layout_gravity="center_horizontal"
    android:background="@drawable/power_on_off"
    android:rotation="0" />

<Button
    android:id="@+id/gait1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/gait1_robot" />

<Button
    android:id="@+id/gait2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/gait2_robot" />

<Button
    android:id="@+id/home"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/home_robot" />

<Button
    android:id="@+id/stop"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/stop_robot" />

</LinearLayout>
```

> JoystickView.java:

```
package com.example.capstonerobottest1;

//Required imports and libraries
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
```

```
import android.graphics.Paint;
import android.graphics.PorterDuff;
import android.util.AttributeSet;
import android.view.MotionEvent;
import android.view.SurfaceHolder;
import android.view.SurfaceView;
import android.view.View;

public class JoystickView extends SurfaceView implements SurfaceHolder.Callback, View.OnTouchListener
{
    public static final int JOYSTICK_LEFT_ID = R.id.joystickLeft; // Constants for joystick IDs (used in Main Activity)
   public static final int JOYSTICK_RIGHT_ID = R.id.joystickRight;
   private float centerX;
   private float centerY;
   private float baseRadius;
   private float hatRadius;
   private JoystickListener joystickCallback;
   private final int ratio = 10; //The smaller, the more shading will occur

   private void setupDimensions() //Circular dimensions
   {
      centerX = getWidth() / 2;
      centerY = getHeight() / 2;
      baseRadius = Math.min(getWidth(), getHeight()) / 3;
      hatRadius = Math.min(getWidth(), getHeight()) / 5;
   }

   public JoystickView(Context context)
   {
      super(context);
      getHolder().addCallback(this);
      setOnTouchListener(this);
      if(context instanceof JoystickListener)
         joystickCallback = (JoystickListener) context;
   }

   public JoystickView(Context context, AttributeSet attributes, int style)
   {
      super(context, attributes, style);
      getHolder().addCallback(this);
      setOnTouchListener(this);
      if(context instanceof JoystickListener)
         joystickCallback = (JoystickListener) context;
   }

   public JoystickView (Context context, AttributeSet attributes)
   {
      super(context, attributes);
      getHolder().addCallback(this);
      setOnTouchListener(this);
      if(context instanceof JoystickListener)
         joystickCallback = (JoystickListener) context;
```

```
    }

    private void drawJoystick(float newX, float newY)
    {
        if(getHolder().getSurface().isValid())
        {
            Canvas myCanvas = this.getHolder().lockCanvas(); //Stuff to draw
            Paint colors = new Paint();
            myCanvas.drawColor(Color.TRANSPARENT, PorterDuff.Mode.CLEAR); // Clear the BG

              //First determine the sin and cos of the angle that the touched point is at relative to the center of the
joystick
            float hypotenuse = (float) Math.sqrt(Math.pow(newX - centerX, 2) + Math.pow(newY - centerY, 2));
            float sin = (newY - centerY) / hypotenuse; //sin = o/h
            float cos = (newX - centerX) / hypotenuse; //cos = a/h

            //Draw the base first before shading
            colors.setARGB(255, 255, 0, 0); //Change color shades

            myCanvas.drawCircle(centerX, centerY, baseRadius, colors);
            for(int i = 1; i <= (int) (baseRadius / ratio); i++)
            {
                myCanvas.drawCircle(newX - cos * hypotenuse * (ratio/baseRadius) * i,
                        newY - sin * hypotenuse * (ratio/baseRadius) * i, i * (hatRadius * ratio / baseRadius), colors);
//Gradually increase the size of the shading effect
            }

            //Drawing the joystick hat
            for(int i = 0; i <= (int) (hatRadius / ratio); i++)
            {
                  colors.setARGB(255, (int) (i * (255 * ratio / hatRadius)), (int) (i * (255 * ratio / hatRadius)), 50);
//Change the joystick color for shading purposes
                myCanvas.drawCircle(newX, newY, hatRadius - (float) i * (ratio) / 2 , colors); //Draw the shading - 3
layer
            }

            getHolder().unlockCanvasAndPost(myCanvas); //Write the new drawing to the SurfaceView
        }
    }

    @Override
    public void surfaceCreated(SurfaceHolder holder)
    {
        setupDimensions();
        drawJoystick(centerX, centerY);
    }

    @Override
    public void surfaceChanged(SurfaceHolder holder, int format, int width, int height) {

    }

    @Override
```

```
    public void surfaceDestroyed(SurfaceHolder holder) {

    }

    public boolean onTouch(View v, MotionEvent e)
    {
        if(v.equals(this))
        {
            if(e.getAction() != e.ACTION_UP)
            {
                float displacement = (float) Math.sqrt((Math.pow(e.getX() - centerX, 2)) + Math.pow(e.getY() -
centerY, 2));
                if(displacement < baseRadius)
                {
                    drawJoystick(e.getX(), e.getY());
                                joystickCallback.onJoystickMoved((e.getX() - centerX)/baseRadius, (e.getY() -
centerY)/baseRadius, getId()); // Gets the normalised coordinates (-1 to 1)
                }
                else
                {
                    float ratio = baseRadius / displacement;
                    float constrainedX = centerX + (e.getX() - centerX) * ratio;
                    float constrainedY = centerY + (e.getY() - centerY) * ratio;
                    drawJoystick(constrainedX, constrainedY);
                                        joystickCallback.onJoystickMoved((constrainedX-centerX)/baseRadius,
(constrainedY-centerY)/baseRadius, getId());
                }
            }
            else
                drawJoystick(centerX, centerY);
            joystickCallback.onJoystickMoved(0,0,getId());
        }
        return true;
    }

    public interface JoystickListener
    {
        void onJoystickMoved(float xPercent, float yPercent, int id);
    }
}
```

> RaspberryPiServer.py:

```
import bluetooth


SERVICE_UUID = "00001101-0000-1000-8000-00805F9B34FB" # Pre-defined UUID
SERVICE_NAME = "BluetoothSerial" #Service/Socket name

def main():
        server_sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM) #Socket
init.
    server_sock.bind(("", bluetooth.PORT_ANY))
    server_sock.listen(1)
```

```
    bluetooth.advertise_service(server_sock, SERVICE_NAME,
                                service_id=SERVICE_UUID,

service_classes=[bluetooth.SERIAL_PORT_CLASS],
                                profiles=[bluetooth.SERIAL_PORT_PROFILE])

    print("Waiting for connection...")
    client_sock, client_info = server_sock.accept()
    print("Accepted connection from", client_info)

    try:
        while True:
            data = client_sock.recv(1024)
            if len(data) == 0:
                break

            #Integration Starts here:

               else: #Remove the else if it is causing an error or empty
terminal

                decodedData = data.decode()
                splitData = decodedData.split(',')
                whichJoystick = splitData[0].strip()

                if whichJoystick == "L":

                    xRotate = splitData[1]
                    print("Leftjoystick X Coordinate:",xRotate)
                 #   continue

                elif whichJoystick == "R":
                    xMove = float(splitData[1])
                    print("Right joystick X Coordinate:", xMove )
                    if "R" in splitData[2]:
                        yMove = float(splitData[2].replace("R", ""))
                        print("Right joystick Y Coordinate:", yMove )
                    else:
                        yMove = float(splitData[2])
                        print("Right joystick Y Coordinate:", yMove )

                if 'L' in str(decodedData) or 'R' in str(decodedData):
                    continue
                else:
                    print("Commands Received:", decodedData)

    except Exception as e:
        print("Error:", e)
    finally:
        client_sock.close() #Terminates the socket on exiting the app
        server_sock.close()

if __name__ == "__main__":
```

```
    main()
```

The most important code extracted from the final movement implementation:

```
def add_lists(list1, list2):
    if len(list1) != len(list2):
        raise ValueError("Lists must have the same size for element-wise addition.")
    result = [x + y for x, y in zip(list1, list2)]

    return result

def multiply_lists(list1, list2):
    # Determine the size of the result (maximum of the lengths of list1 and list2)
    result_size = max(len(list1), len(list2))

    # Perform element-wise multiplication for the shared size
    shared_size = min(len(list1), len(list2))
    result = [x * y for x, y in zip(list1[:shared_size], list2[:shared_size])]

    # Pad the result with zeroes to make it the size of the largest array
    if len(list1) > len(list2):
        result.extend([0] * (result_size - shared_size))
    elif len(list2) > len(list1):
        result.extend([0] * (result_size - shared_size))

    return result

def distance_within_wrapping_range(smaller, larger, value):
    if (smaller <= value and value <= larger):
            return (value - smaller) / (larger-smaller)
    elif smaller > larger and (value <= larger or value >= smaller):
            if value <= smaller:
                return (value - smaller + 1) / (1 + larger - smaller)
            else:
                return (value - smaller) / (1 + larger - smaller)
    else:
        return 0;

def linear_interpolation(points, times, target_time):

    if len(points) < 2:
        raise ValueError("At least two points are required for interpolation.")
```

```
    # Find the index of the point immediately before the target time
    for i in range(len(times) - 1):
        if times[i] <= target_time <= times[i+1]:
            t0 = times[i]
            t1 = times[i+1]
            v0 = points[i]
            v1 = points[i+1]
            interpolated_point = []
            for j in range(3):  # Assuming each point has 3 numbers
                interpolated_value = v0[j] + (target_time - t0) * (v1[j] - v0[j]) / (t1 - t0)
                interpolated_point.append(interpolated_value)
            return interpolated_point

    raise ValueError("Target time is outside the range of provided points.")

    def condition(self):
        while True:
            #other code for maintaining continuity in robot's operation
            #this code below integrates apurva’s controller
            elif RaspberryPiServer.xMove != 0 or RaspberryPiServer.yMove != 0 or
RaspberryPiServer.xRotate: #this is the new user app controller integration
                if (RaspberryPiServer.xRotate != 0):
                    c.rotateInPlace(RaspberryPiServer.xRotate * 10)
                else:
                    c.walk(RaspberryPiServer.xMove,RaspberryPiServer.yMove,5)
            #other code for maintaining continuity in robot's operation

    def restriction(self,var,v_min,v_max):
        if var < v_min:
            return v_min
        elif var > v_max:
            return v_max
        else:
            return var

    def map(self,value,fromLow,fromHigh,toLow,toHigh):
        return (toHigh-toLow)*(value-fromLow) / (fromHigh-fromLow) + toLow

    def walk(self,xAnalog,yAnalog,speed,angle=0,legClearHeight=40):
        x=self.restriction(float(xAnalog),-1,1)
        y=self.restriction(float(yAnalog),-1,1)
        totalFrames = 32 + (10 - int(speed)) * 8;
        delay=0.01
```

```
        point=copy.deepcopy(self.body_point)
        #if y < 0:
        #   angle=-angle
        if angle!=0:
            x=0
        feetVelVector=[[0,0],[0,0],[0,0],[0,0],[0,0],[0,0]]
        for i in range(6):

feetVelVector[i][0]=((point[i][0]*math.cos(angle/180*math.pi)+point[i][1]*math.sin(angle/180*
math.pi)-point[i][0])+x*35)/totalFrames

feetVelVector[i][1]=((-point[i][0]*math.sin(angle/180*math.pi)+point[i][1]*math.cos(angle/180
*math.pi)-point[i][1])+y*35)/totalFrames
        if x == 0 and y == 0 and angle==0:
            self.coordinateTransformation(point)
            self.setLegAngle()
        else:
            baseMovement = 7.45;
            times = [0,1/8,2/8,3/8,5/8,6/8,7/8,8/8]
            firstLegs = [0,8,8,4,-4,-8,-8,0]
            secondLegs = [0,-4,-8,-8,8,8,4,0]
            firstPoints = [];
            secondPoints = [];
            for i in range(len(firstLegs)):
                firstPoints.append([firstLegs[i]*-baseMovement,firstLegs[i]*-baseMovement,0]);

secondPoints.append([secondLegs[i]*-baseMovement,secondLegs[i]*-baseMovement,0]);
            leg_stagger_offset = [0.25,0.75]

            for j in range (totalFrames):
                for i in range(3):
                    #horiz positioning
                    point[i*2] =
add_lists(self.body_point[i*2],multiply_lists(feetVelVector[i*2],linear_interpolation(firstPoints,ti
mes,j / totalFrames)))
                    point[i*2+1] =
add_lists(self.body_point[i*2+1],multiply_lists(feetVelVector[i*2+1],linear_interpolation(second
Points,times,j / totalFrames)))
                    #lift
                    if (j <= leg_stagger_offset[0] * totalFrames or j >= (1-leg_stagger_offset[0]) *
totalFrames):
                        amnt1 =
distance_within_wrapping_range(1-leg_stagger_offset[0],leg_stagger_offset[0],j/totalFrames)
                        point[2*i+1][2] = math.sin(self.map(amnt1,0,1,0,3.14))*legClearHeight+self.height
```

```
            if (j <= leg_stagger_offset[1] * totalFrames and j >= (1-leg_stagger_offset[1]) *
totalFrames):
                amnt2 =
distance_within_wrapping_range(1-leg_stagger_offset[1],leg_stagger_offset[1],j/totalFrames)
                point[2*i][2] = math.sin(self.map(amnt2,0,1,0,3.14))*legClearHeight+self.height

        self.coordinateTransformation(point)
        self.setLegAngle()
        time.sleep(delay)

  def rotateInPlace(self, angle):
    self.walk(0,0,5,angle)
```

The important code specific to the simulation code, accounting for different robot setup, servo and time control:

```
def rearrange_array(numbers, array):
  # Check if the input list contains exactly 6 unique numbers from 1 to 6
  if len(numbers) != 6 or not all(1 <= num <= 6 for num in numbers) or len(set(numbers)) != 6:
    print("Invalid input. Please provide a list of 6 unique numbers from 1 to 6.")
    return None

  # Check if the input array contains exactly 18 elements
  if len(array) != 18:
    print("Invalid input. Please provide an array of 18 elements.")
    return None

  # Rearrange the array based on the input numbers
  index = 0
  tempArr = [None] * 18
  for num in numbers:
    for i in range(3):
      tempArr[index] = array[(num-1) * 3 + i]
      index += 1

  return tempArr

class Servo:
  def __init__(self):
```

```
    self.motor_names = [
      'shoulder2', 'shoulder8', 'foot2', 'shoulder6', 'shoulder12', 'foot6','shoulder3', 'shoulder9',
'foot3',
      'shoulder1', 'shoulder7','foot1', 'shoulder5', 'shoulder11', 'foot5', 'shoulder4',
'shoulder10', 'foot4'
    ]

    self.motors = [robot.getDevice(motor_name) for motor_name in self.motor_names]
    self.motorMapping = [9,8,31,12,11,10,15,14,13,22,23,27,19,20,21,16,17,18];
    self.reverseAngle = [1,-1,1] + [1,-1,1] + [1,-1,1] + [1,1,-1] + [1,1,-1] + [1,1,-1]
    self.angleOffset = [-90,90,0] + [-90,90,0] + [-90,90,0] + [-90,-90,0]  + [-90,-90,0] + [-90,-90,0]

    input_array = [1,2,3,6,5,4]
    self.motorMapping = rearrange_array(input_array, self.motorMapping )
    self.reverseAngle = rearrange_array(input_array, self.reverseAngle )
    self.angleOffset = rearrange_array(input_array, self.angleOffset )

    for i in range(3):
      self.angleOffset[i  *3 + 2] = 0
    for i in range(3):
      self.angleOffset[9 + i  *3 + 2] = 180

while robot.step(timestep) != -1:
  time = robot.getTime() #about 0.01 seconds per loop. (unit is seconds)

  if (simFrame == 50 + 60 + 120):
    print("move gait setup finished");
    c.walk(1,0,6);
  if simFrame < 500:
    print("\nTURN 20 DEGREES\n");
    c.rotateInPlace(5)
  elif simFrame < 800:
    c.walk(1,0,6);
  else:
    c.walk(0,1,8);

  if (simFrame > 50 + 60 + 120):
    c.gaitIteration()
```

Python code for the homing system

```
import RPi.GPIO as GPIO
import time
```

```
minVoltageThreshold=0.00488V
maxVoltageThreshold=2V

def auto_home():
  # Rotation function call to start the auto-homing process
    Rotation()
    # Function to read voltage from the resistor pin
    def readVoltage():
        sensorValue = 1023 - GPIO.input(resistorPin)  # Read the analog
value from the resistor pin
        voltage = sensorValue * (5.0 / 1023.0) * 1000  # Convert analog
value to voltage (assuming 5V supply)
        return voltage

    # Rotation function
    def Rotation():
        readVoltage()
        while readVoltage() < minVoltageThreshold:  # Rotate while voltage
is below threshold
            rotateInPlace(5)  # Rotate the robot by 5 degrees
            readVoltage()

        Move()  # If voltage rises above threshold, move

    # Move function
    def Move():
        readVoltage()
        while readVoltage() >= minVoltageThreshold and readVoltage() <
maxVoltageThreshold:  # Keep walking while voltage is above threshold
            walk()
            readVoltage()

        if readVoltage() >= maxVoltageThreshold:
            walk()  # The robot is very close to home. It will walk one
more unit and then stop.
        else:
            Rotation()  # If voltage drops below threshold, go back to
rotation
```

Note: The data sheets are attached after are part of this appendix section.