Table of Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we give a fair introduction about robots that exist and perform tasks that help humans in their everyday affairs. There are numerous robots which perform multiple tasks starting from production in factories, entertaining people, mining and even serve humans in restaurants. The next subsection gives an introduction about the different types of robots that coexist with us.

1.1 Robots in the Real World

Robots are designed to specifically tackle each task in a different way. Industrial robots are specifically designed only to perform one single task. One Industrial robot cannot perform task of another robot. They are highly task oriented. This leads to robot designs that are not reusable. The different robots that ease the human's task are shown below.

1. **Roomba** : It is a series of autonomous vacuum cleaning robot which are manufactured and sold by iRobot. It is capable of sensing the environment and clean it by autonomously navigate it. First introduced in 2002, it has become a famous home robot. This robot is specifically designed to vacuum the house and cannot perform any other job. The figure 1.1(a) shows a 700 series Roomba.
2. **Spirit and Opportunity Rovers** : These robots were created just to explore Mars and its terrain features. It was developed by NASA and were landed on the surface of Mars in January 2004. Spirit ran out of battery in 2010, and opportunity sent it's last message on June 10, 2018. These robots help scientists understand the red planet for possible habitats.
3. **General Atomics MQ-1 Predator** : This Remotely piloted aircraft was designed by General Atomics for the US Air force and CIA. It was first used for aerial reconnaissance during the early 90s and was also used for forward observation. The main sensing devices were cameras but it also had other sensors. It was even upgraded to be able to carry and fire

AGM-114 Hellfire missiles. The drone started its service in 1995 and helped it's nation in multiple wars and NATO interventions.

4. **ABB IRB 1660ID** : This robot is an industrial robot which are perform very accurate arc welding and machine tending. It's agile and hence reliable for machine tending tasks. The payload capacity is 6kgs and can support a wide variety of tools for achieving a variety of goals. This is one of the many advanced industrial robots used in the production line.

Figure 1.1(a)(b)(c)(d): 700 series Roomba(Top-Left), Spirit and Opportunity Rovers(Top Right), MQ-1 Predator Drone(Bottom-Left), ABB IRB 1660ID(Bottom-Right)

1.2 Existing Humanoid Robots

Coming to the humanoid robots there is a lot of research that is carried across various parts of the world. There are numerous attempts at mimicking the human walk, but rarely few have succeeded because it largely depends on the hardware used.

The robots listed below are some of the famous humanoid robots that are being developed in the world.

1. **ASIMO** : This robot developed by Honda R&D after researching for almost a decade, is one of the most advanced humanoid robot the world has seen. It has 28 Degrees of Freedom in total, with 12 in legs, 14 in arms and 2 for the head. Its 160 cm high and weighs about 130 kg[5]. It was made to help people in their domestic day-to-day tasks. The robot can walk straight, move sideways & diagonal and climb stairs at ease . It can sense obstacles, carry items of different structures and play football.
2. **HRP(Humanoid Robotics Platform)** : The HRP robots come in multiple versions. HRP-2 was designed and integrated by Kawada Industries, Inc. It has a height of 154 cm and weighs 58 kg with 30 DoF. Educational Universities use this robot to teach students about the concepts that go into making this robot. HRP-3 developed as a research and development of platform for the New Energy and Industrial Technology Development Organization (NEDO). It has a more improvised structure and system compared to HRP-2. It has a height of 160 cm and weighs 68 kg with 42 DoF. The latest entry in this series is the HRP-4 and is a combined effort of the salient features from HRP-2 and HRP-3
3. **Nao** : This robot is developed by SoftBank Robotics. It is a Kid-Size robot with a height of 58 cm and it weighs about 4 kg. The robot looks appealing and also has a sturdy structure, so its used as a platform to work on. It has 27 DoF which allows to perform yoga asanas. Nao is chosen as a standard robot to be given to participants of in RoboCup Standard Platform league.
4. **Marty** : This robot developed by ROBOTICAL, is a small humanoid robot. It is an easy to build robot with 8-DOF which is 3D printable.
5. **KHR Kondo series** : These robots are small sized, basically used for research purposes. There are 2 versions namely, KHR Kondo 2HV and KHR Kondo 3HV, the only difference being that the latter has 2 extra

degrees of freedom in its legs and can perform the turning operations more easily.

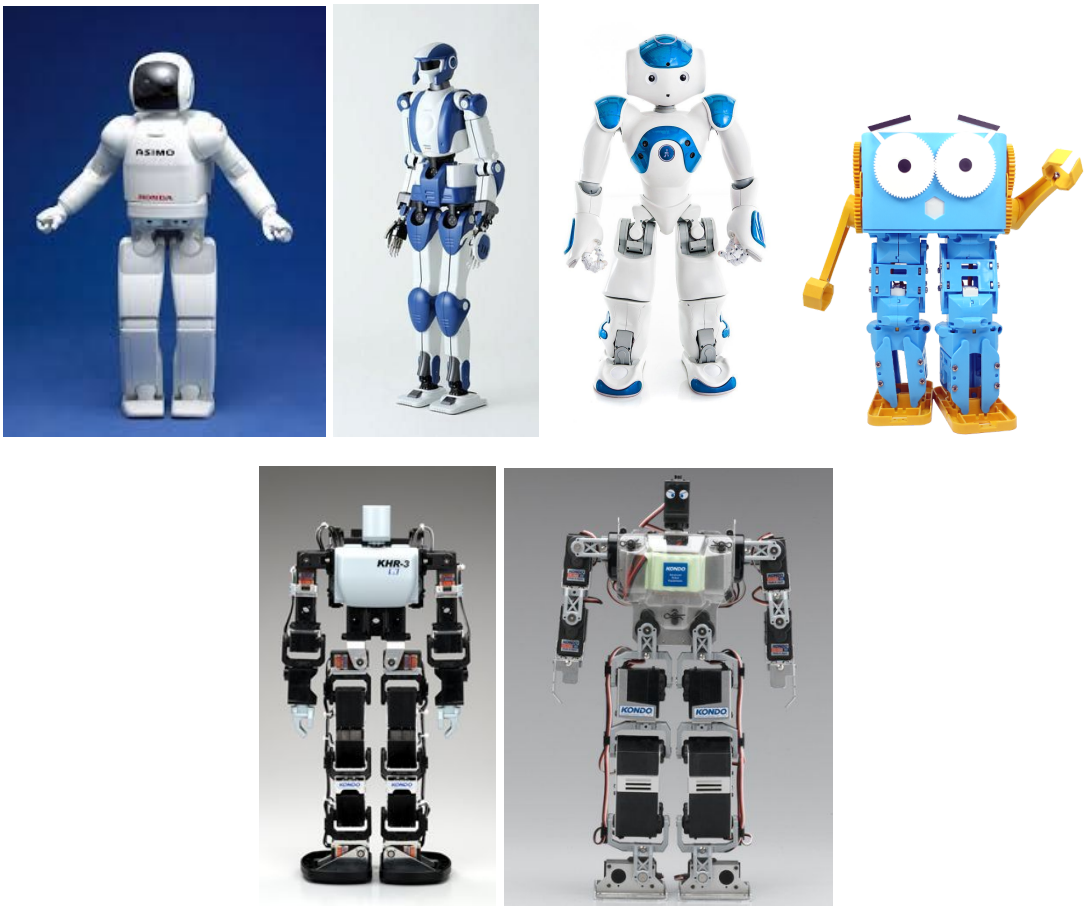


Figure 1.2(a)(b)(c)(d)(e)(f) : Pictures of ASIMO, HRP, Nao, Marty (Top Left- Top Right), Kondo KHR 3HV and Kondo KHR 3HV(Bottom left-Bottom Right)

2 Problem Statement

Even with a decade long intensive research, human like walking is not perfectly achieved. This leads to more and more opportunities in the field of gait generation. The main problems in this field is described below.

2.1 Humanoid Locomotion Requirements

Humans are the most intelligent creatures in the known living world. We can easily perform multiple tasks at once. Whether it be solving a simple mathematical problem or building a megastructure we can do it. It's hard to achieve human like walking because, the human body has an anatomy which is very complex to understand. We react to threats very quickly and act quickly. This is one advantage where machines cannot be fast enough. The main requirements to achieve stable walking motions are :

1. Motors that can produce enough Torque to actuate the arms/legs.
2. Postures that are kinematically balanced, COM is within the support polygon
3. The acceleration should not topple the robot or lead to inertial instability.

2.2 The Goal

Considering all the requirements the goal of this project can be defined as,

" Stable Gait Generation for a Small-Sized Humanoid Robot using 3D Linear Inverted Pendulum Model for walking motions"

The walking motion includes walking forward, turn left and turn right. This project also aims at achieving other motions like, Squats, Pushups, a welcome gesture. All of the motions are shown by proof that they are stable using ZMP measurement.

3 Literature Survey

This project came in response to achieve stable gait for small sized humanoid robots. This process involves Gait Planning (solving kinematic equations of biped robot and to generate ankle trajectory) and using important physical parameters of humanoid robot such as Zero Moment Point(ZMP).

3.1 Zero Moment Point(ZMP)

Achieving stability while locomotion is the challenging aspect in humanoid robotics.Zero Moment Point can be defined as the point on the surface of the foot where resultant is passed.

3.1.1 Support Polygon, Center of Mass

Support Polygon is an important concept related to ZMP. Support Polygon is the region formed by enclosing all the contacts points of robot and ground using some elastic cord. A point is called center of mass if it moves in the direction of the force without rotating. [4]

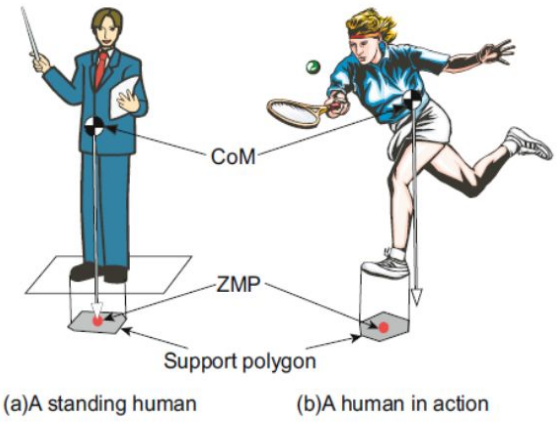


Fig 3.1: Center of Mass, Zero Moment Point and Support Polygon

Consider a human standing on the ground and a human in action. In first case the ground projection of the center of mass coincides with the ZMP while in second case the ground projections does not coincide with the ZMP. In both cases ZMP lies inside the Support Polygon.

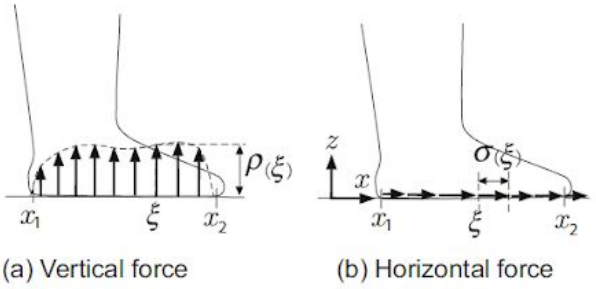


Fig 3.2: Ground reaction force of a 2D Model

The ZMP always exist inside of the support polygon [4]. The ZMP in case of 2D is the point where the moment of ground reaction forces sums up to zero. For 3D cases only the horizontal components of the moment of ground reaction forces are considered to calculate ZMP. To calculate the ZMP in both these cases, we use the following equations.

For 2D cases :

$$f_x = \int_{x_1}^{x_2} \sigma(\xi) d\xi \qquad .. 3.1.1$$

$$f_z = \int_{x_1}^{x_2} \rho(\xi) d\xi \qquad .. 3.1.2$$

$$\tau(p_x) = -\int_{x_1}^{x_2} (\xi - p_x)\rho(\xi) d\xi \qquad .. 3.1.3$$

Solving $\tau(p_x) = 0$, we obtain the ZMP in 2D case, that is

$$p_x = \frac{\int_{x_1}^{x_2} \xi\rho(\xi) d\xi}{\int_{x_1}^{x_2} \rho(\xi) d\xi} \qquad .. 3.1.4$$

In case of 3D, we split the entire forces into its components and only consider the horizontal components as only they are involved with gravity and balancing the foot.

3.1.2 Measuring the ZMP in practical cases

The ZMP can be measured using multiple sensors available which are described below. The sensors are generally attached to the robot feet. There can be a single sensor

or multiple sensors working together. ZMP of each foot is measured and then combined to actually calculate the overall Zero Moment Point. The different sensors are

1. **6 Axis Force/Torque Sensor** : These sensors have rubber bushes and dampers which calculate the forces acting on it in 6 directions, namely Force in 3 axes, and Torque in 3 axes. These sensors are bulky and are typically on the expensive side.

Figure 3.3 : 6-axis Force/Torque sensor

2. **Measurement using Multiple Sensors** : To have smaller and lighter sensors, we can use multiple small sensors such as Force Sensitive Resistors and combine their output to calculate the ZMP[3]. The FSR is a one-dimensional force sensor which varies resistance with force.

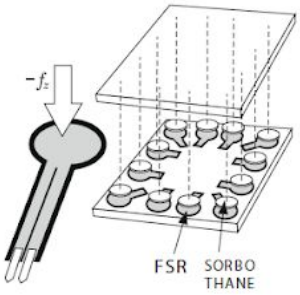


Figure 3.4 : Multiple FSR for ZMP calculation

3.2 3D Linear Inverted Pendulum Model

3D linear Inverted Pendulum Model is a type of walking pattern generator used to realise biped walking by generating trajectory.Walking pattern can be defined as a set of joint angles to achieve desired walking.

The assumptions made in 3D Inverted Pendulum Model are as follows:

1. Total mass of the robot is concentrated at its center of mass.

2. We assume that the robot has massless legs,whose tips contact the ground at single rotating joints.
3. The end of the joint only moves in a single constrained plane, which has to be parallel to the ground.

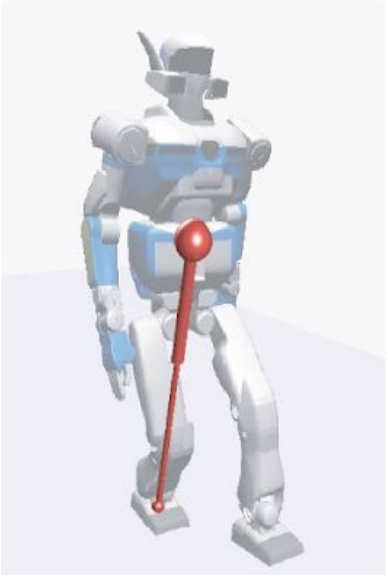
The important parameter in the 3D LIP model are r-supporting distance, $g-gravity$, $f-kick\,force$. The kick force f can be resolved into components as fx ,fy ,fz

$$fx = (x/r)f \quad .. 3.2.1$$

$$fy = (y/r)f \quad .. 3.2.2$$

$$fz = (z/r)f \quad .. 3.2.3$$

The robot's leg is assumed to be extensible using the kick force f which makes sure the CoM stays on the same plane.The only forces that act on CoM are kick force and gravity, thus we get the motion equations as follows.



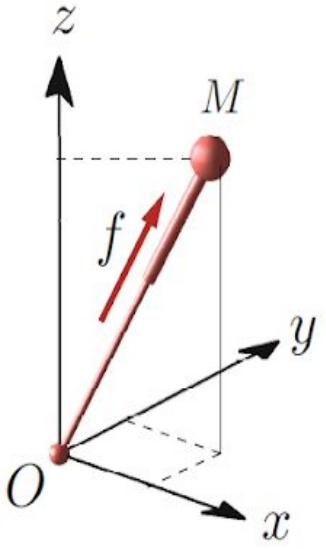


Figure 3.5 Approximation of a biped humanoid robot as 3D inverted pendulum

$$M\ddot{x} = \left(\frac{x}{r}\right)f \quad ...3.2.4$$

$$M\ddot{y} = \left(\frac{y}{r}\right)f \quad ... 3.2.5$$

$$M\ddot{z} = \left(\frac{z}{r}\right)f - Mg \quad ... 3.2.6$$

Introduction of a constraint plane defined as shown in figure 3.6. Where $k_{x,}\, k_y$ give slopes and z_c gives height of the constraint plane. Acceleration of the CoM has to be orthogonal to normal vector of the constraint for CoM to move along the constraint plane. The CoM is restricted to move only in the defined plane, and to maintain it, enough kick force must be applied. The slope of the plane can be manipulated to make the robot climb inclined plane or a stairway

$$z = k_x x + k_y y + z_c \qquad .. 3.2.7$$

Solving equations and making proper substitution, we get

$$f = \frac{Mgr}{z_c} \qquad .. 3.2.8$$

Using this kick force equation, we can easily tell that, the entire model depends on physical parameters of the robot and not on other parameters.

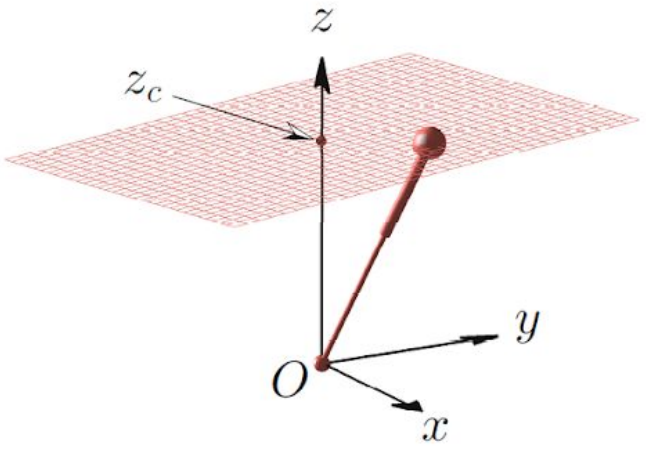


Figure 3.6 : The defined constraint plane for the 3D LIPM, using the kick force f the CoM is constrained to the particular plane.

Derivation of horizontal dynamic equations of the CoM is as follows

$$\ddot{x} = \frac{g}{z_c} x \qquad .. 3.2.9$$

$$\ddot{y} = \frac{g}{z_c} y \qquad .. 3.2.10$$

These equations give the acceleration required for the CoM in the plane. The above linear equations have z_c as the only parameter and inclination parameters k_x, k_y does not affect

CoM's horizontal motion. Hence we call it as 3D linear inverted pendulum as the acceleration is linear which can be inferred by the equation.

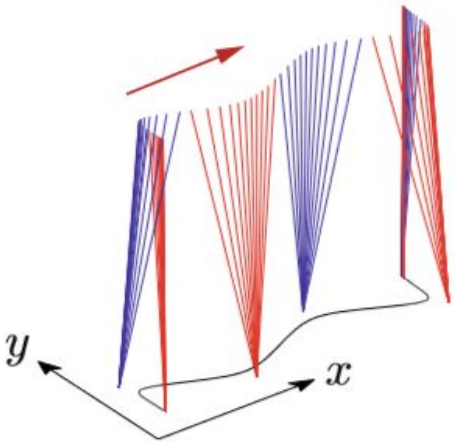


Figure 3.7 Walking pattern based on 3D LIPM

The geometric representation of 3D linear Inverted Pendulum Model is a hyperbola as shown in the above figure. The overall motion of the CoM is given by the equation 3.2.11. In the figure, it's evident that the CoM trajectory is continuous but the foot placements cannot be.

$$\frac{g}{2z_c E_x} x^2 + \frac{g}{2z_c E_y} y^2 + 1 = 0 \qquad \text{.. 3.2.11}$$

In the figure 3.7, the red line indicates the way left and right should interchangeably move to mimic a walking sequence. The algorithm basically shifts the CoM over either of the legs and make the other leg swing to the next position. This shift in CoM and ZMP have to carefully calculated using the actual robot's parameters.

In practical situations, we need to directly specify the foot placements which can be represented by using length of the step s_x and width of the step s_y .The foot placements can be represented (p_x, p_y) from which we can determine walk primitive of n_{th} step. The placement is given by

$$\begin{bmatrix} \bar{x}^{(n)} \\ \bar{y}^{(n)} \end{bmatrix} = \begin{bmatrix} \frac{s_x{}^{(n+1)}}{2} \\ (-1)^n \frac{s_y{}^{(n+1)}}{2} \end{bmatrix} \qquad \text{.. 3.2.12}$$

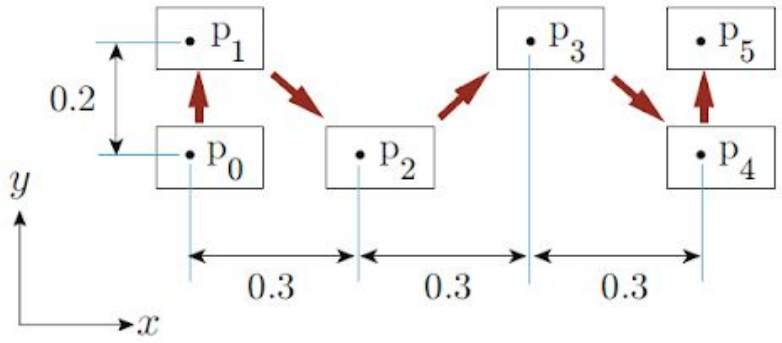


Figure 3.8 foot placements p0 . . . pN from which determine step length and the step width

By defining the foot placements, we directly have the end-effectors position, and from the equation 3.2.11 we can calculate the trajectory of the center of mass. Using these positions, we can calculate the positions of other links and then apply inverse kinematics to obtain the joint position over the entire range of the trajectory which will give us a sequence of angles. This sequence of angles guide the robot to perform the walking task. The 3D LIP model is stable walking algorithm in itself. It does not require calculating the ZMP to interfere in the algorithm. However other algorithms exist which use ZMP for better gaits. Coming to the inverse kinematics, the next section gives a introduction to it.

3.3 Inverse Kinematics

Kinematics is the theory which analyses the relationship between the position and attitude of a link and the joint angles of a mechanism. Kinematics lays a basic foundation for robotics. It includes mathematics which deals with representing a dynamic object in 3D space. If the joint angles are known, and the final position of the robot tool tip is calculated, it's known as Forward Kinematics. If the goal position is known and the joint angles need to be calculated, it's called Inverse Kinematics. Inverse kinematics is a field where the constraints define whether or not the equations are solvable. Hence, the approach in representation of links of a robot decides the effectiveness and solvability of the problem.

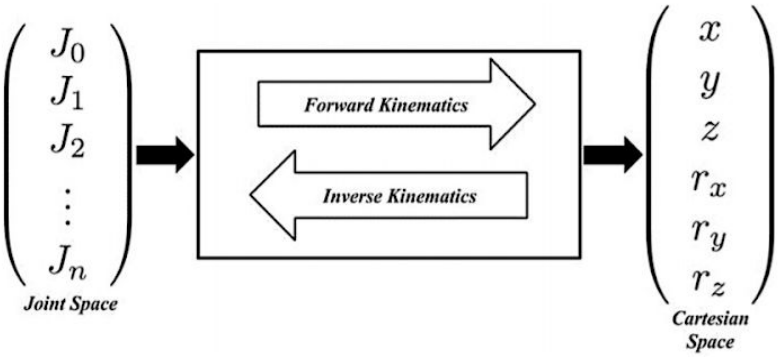


Figure 3.9 : Description of Forward and Inverse Kinematics

3.3.1 Inverse Kinematics approaches

The different approaches to the inverse kinematics problem will depend on the construction of the mechanism. Not all approaches have solutions to all the mechanisms. The choice of approach depends on the way the robot links are joined and constructed[1].

1. **Pseudo Inverse of Jacobian** : This method involves using a pseudo inverse of a Jacobian matrix of the homogeneous transformation matrix. The homogeneous matrix is a series of multiplied rotation matrices and translation matrices in the order all the links are placed. It's based on the assumption that the end-effector reaches the goal position in incremental steps. The Jacobian is rarely square and hence the pseudo inverse is used.
2. **Denavit-Hartenberg Method** : The Denavit-Hartenberg method uses 4 key parameters to represent each link. Using these parameters the entire homogeneous transformation matrix can be calculated. From the transformation matrix, the Jacobian can be calculated and using pseudo inverse method, the joint angles can be determined.
3. **Geometric Approach** : Sometimes when the joint structure is simple, it is possible to find out the joint angles using a geometric approach using positions of each links and their constraints. The constraints can be effectively used to determine joint angles with the help of trigonometric formulae. This approach doesn't always work as most times, the mechanism is complicated to be solved this way

4 Implementation

This section speaks about the design and construction of the humanoid robot and the implementation of 3D LIP algorithm and FSR stability on the robot. The major components such as chassis, motors and controllers were bought from robokits.co.in

4.1 Hardware Architecture

4.1.1 Components Used

1. **Actuation** : **Metal Gear Dual Shaft 16 Kg.cm Digital Servo Motor** : For the robot to be fully controllable, we need actuators that are precise and at the same time provide high torque. We choose a digital servo because it is accurate and the gears are made of metal which gives more strength to the motors. The exact specifications of the motor are shown below.

Specifications	4.8V	6.0V	7.4V
Idle current(at stopped)	4.85mA	7mA	8mA
No Load speed	0.18sec/ 60^o	0.16sec/ 60^o	0.14sec/ 60^o
Running Current(at no load)	160mA	190mA	230mA
Torque	14kg.cm	16Kg.cm	18.2Kg.cm
Stall Current	1200mA	1500mA	1900mA
Pulse width range	500 µsec - 2500 µsec		
Neutral Position	1500 µsec		

Table 4.1 : Specification of Servo Motor

Figure 4.1 : Dual Shaft Metal Gear Digital Servo

2. **Servo Controller** : **Arduino Uno R3 based 18 Servo controlle**r : The servo controller used is a Arduino based Servo Controller which uses an ATmega328p as its brain and a slave IC which works on I2C to control 18 servos with independent setting for each servo. It supports communication via UART with switchable USB and bluetooth channels. The advantage of this arrangement is 18 servos can be controlled using 2 pins and the other IO pins on the ATMega328p can be used for extra features. Various sensing devices can also be incorporated. The following figure shows the controller with all its parts described.

Figure 4.2 : 18-Channel Servo Controller

1	USB Connector for connecting USB Cable to PC
2	Power Connector 5 - 7.5 V DC
3	Switch for Switching Between Bluetooth and USB Mode.
4	Bluetooth Module
5	Bluetooth Status LED
6	LED on Pin 13 of Circuit Board as in Arduino
7	Reset Switch

3. **Arduino Nano**: Arduino Nano is a small sized microcontroller which is useful in integrating and controlling sensors. The role of the nano is to only act as a mediator between sensors and commanding machines, for example, a bridge between FSRs and CPU for stability control. The figure 4.3 explains the structure of Nano and the specifications are as shown.

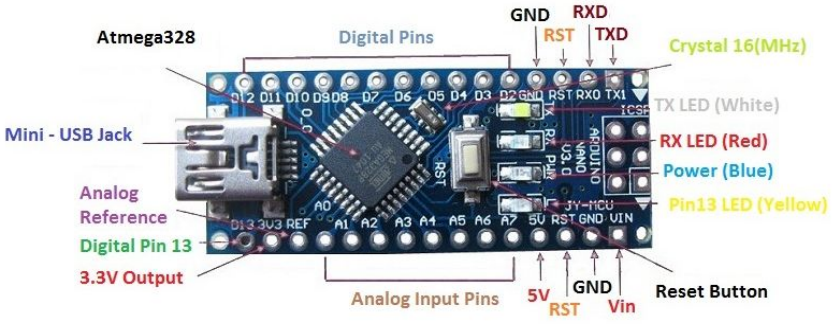


Figure 4.3 : Arduino Nano pinout diagram

Microcontroller	Atmega328p
Operating Voltage	5V
Input Voltage	7-12V
Digital I/O pins	14
PWM	6 out of 14 digital pins
Max. Current Rating	40mA
USB	Mini
Analog Pins	8
Flash Memory	16KB
SRAM	1KB
Crystal Oscillator	16MHz
EEPROM	512 bytes
USART	Yes

Table 4.2 : Specification of Arduino Nano

4. **<u>Bluetooth Module HC-05</u>**: HC-05 module is an easy to use module mainly used for serial port communication. It is designed for easy wireless serial communication. It is very compact and its dimensions are 12.7mm x 27mm. The specifications are

    • Typical -80dBm sensitivity

    • Up to +4dBm RF transmit power

- Low Power 1.8V Operation
- 1.8 to 3.6V I/O
- PIO control
- UART interface with programmable baud rate
- With integrated antenna and edge connector
- Supported baud rate: 9600, 19200, 38400, 57600, 115200,

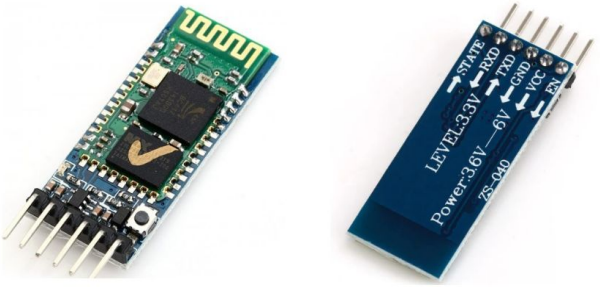
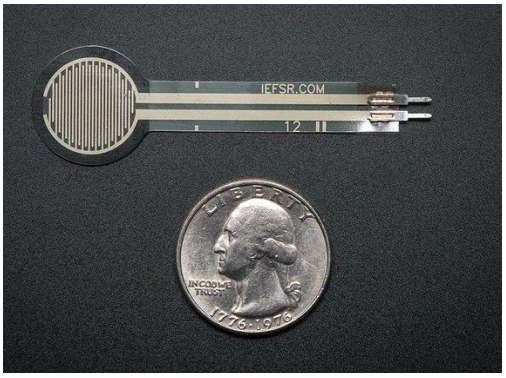


Figure 4.4 : HC05 Bluetooth Module

5. **Force Sensitive Resistor** : These sensors allow detection of physical pressure. They can detect squeezing and weight acting on it. They are comparatively cheap and easy to use. The figure 4.4 shows an FSR, specifically the Interlink 402 model. The 1/2" diameter round part is the sensitive bit. There are 2 layers in the FSR which are separated by a spacer. The active element makes contact with the semiconductor more and more as the pressure increases, and hence the resistance goes down. The figure 4.6 shows the variation of resistance with respect to pressure in an FSR.



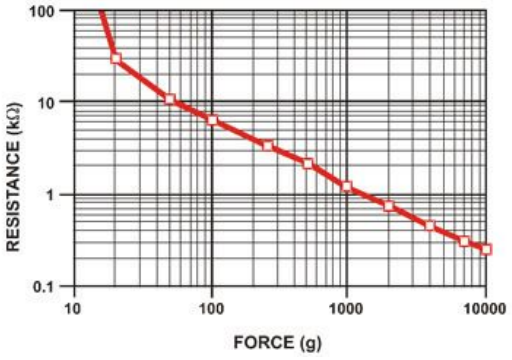


Figure 4.5 : A 0.5" FSR. **Figure 4.6** : Variation of Resistance w.r.t force in a FSR.

6. **Power Supply** : The battery used for powering up the entire robot is Lithium polymer rechargeable battery. The specification of the battery is 7.4V 2-cell 3300mAh battery. It is connected to the robot via a 16A Rocker switch. This battery can supply upto 40A of current in a single second. The figure 4.7 shows the battery and the rocker switch used.

Figure 4.7 : 7.4V Li-Po battery and 16A rocker switch.

4.1.2 Hardware Design

There are a number of design choices which are available for the humanoid robot. The robot needs to be designed based on the operations it will perform. A robot which has to only perform walking can be designed only with 2 legs without torso or hands. The legs of the humanoid robot need to have sufficient DOF to be able achieve all operations but not complicate the mechanism too much. The major types of humanoid robots have either 5 or 6 DoF per leg.

Humanoids with 6 DOF per leg have an advantage over 5 DOF. The gait generation for turning becomes easier in these cases. Though the 5 DOF can also achieve this, it depends on the SLIP generated by the floor, hence the amount of turn achieved is depends on the frictional force between the foot and the floor. Since, we aim at generating gait for not just walking, we need sufficient DOF in the body and arms. This need not be a very large, but a minimum number of DOF will do the job.

The robot's sensors make a large impact on the gait generation and refinement. The feedback it produces can be effectively used to improve upon the algorithms. They can also provide vital information about the surrounding environments for other uses. The materials used to build the robot also make a

difference. The robot has to be light to avoid excess stress on the motors. Lastly, the power source used must be able to completely cater to all the power requirements in the robot. The requirements for the small-sized humanoid robot to be designed are shown in the table below.

Functionalities	Mathematically Generated Gaits	Walking Forward
		Turn Left, Turn Right
	GUI generated Gaits	Squats, Pushups
		Welcome Gesture
Sensors	Left Foot	4 FSRs
	Right Foot	4 FSRs
Degrees of Freedom	Leg	5 per Leg
	Arms	3 per Arm
	Head	1
	Total	**10 + 6 + 1 = 17**
Dimensions	Height	365mm
	Weight	2 Kgs

Table 4.3 : Requirements for the Humanoid Robot.

The 17-DOF chassis as bought from robokits.co.in is shown below with all its degrees of freedom numbered from 1-17.

4.2 Construction of Humanoid

4.2.1 Design Specifications

For design of the physical robot we have to consider link parameters and rotation around particular axis(Roll, Pitch, Yaw).The rotation around x axis, y axis, and z axis,we call them **Roll**, **Pitch** and **Yaw** respectively. Each leg has 5 DoF and each arm has 3 DoF with 1 DoF for the head. The axis of the cylinder in the picture, is the axis of rotation of the joint.

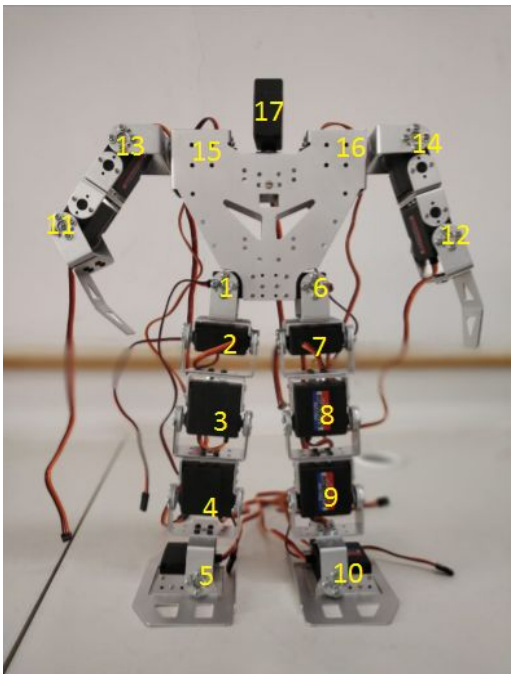


Figure 4.8 : 17-DOF Robot Chassis with all DOF numbered.

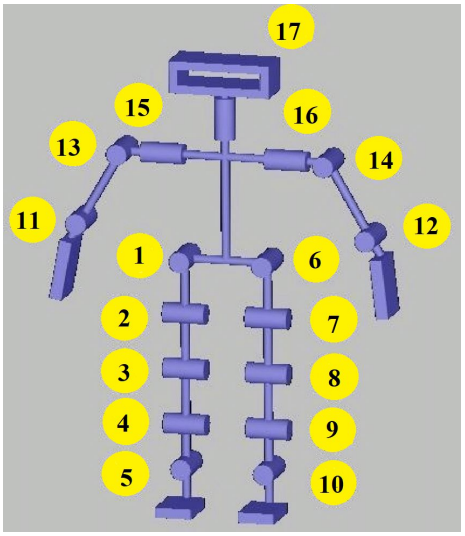


Figure 4.9 :The following depicts the position of the roll, pitch and yaw

Legs: 5 DOF each	**Arms: 3 DOF each**	**Neck : 1 DOF**
• Hip Roll • Hip Pitch • Ankle Pitch • Ankle Roll • Knee Pitch	• Shoulder Roll • Elbow Pitch • Wrist Pitch	• Neck Yaw

Link Parameters of the physical robot are very important parameters while generating motion sequences. The link lengths should be precisely known as they are needed in calculating the inverse kinematic equations. Figure 4.9 explains all the different lengths that are used in the robot, and the table 4.4 depicts all the specifications of these parameters.

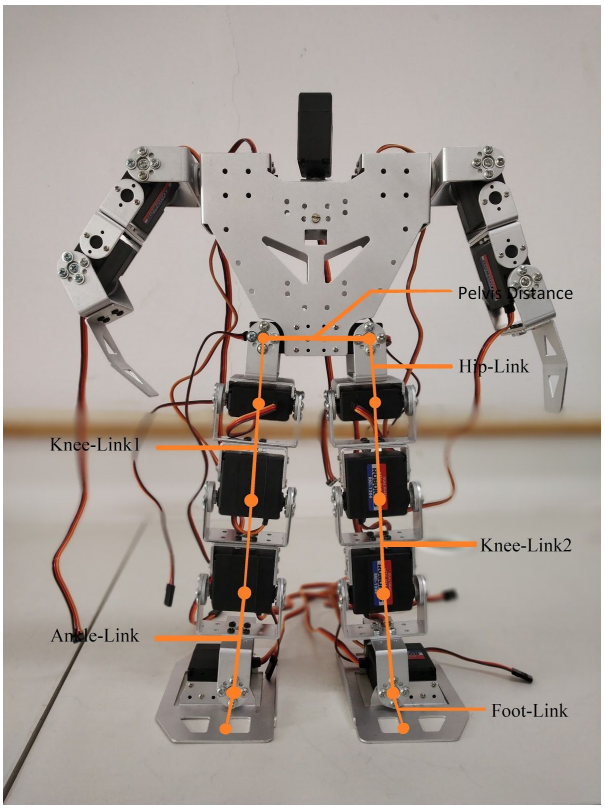


Figure 4.10 Humanoid Robot with Link Parameters

Link Parameters	**Lengths(in cm)**
Pelvis Distance	6.5
Hip-Link L1	4.5
Knee-Link1 L2	6.5
Knee-Link2 L3	6.5
Ankle-Link L4	5
Foot-Link L5	1.7
Centroid	21

Table 4.4 Link parameters of humanoid robot(TONY) from **figure 4.10**

The physical humanoid chassis consists of aluminium chassis weighting 1.650Kg shown in figure 4.8 and has following components bought from robokits.co.in

- 2 × Robot feet Aluminium Bracket
- 2 × Aluminium body frame
- 12 × Big U Servo Bracket
- 9 × Small U Servo Bracket
- 2 × Aluminium Frame

4.2.2 Construction of Physical Robot

Using, the components and the chassis mentioned above, a brief explanation about the construction of the robot is given below.

- The robot feet bracket are fixed with servo motor with screws. (Note Servo Motors are set to neutral position before attaching them). Figure 4.11 shows the foot of the robot.

Figure 4.11 Robot Feet Assembly

- The roll angle of the ankle is then converted to the pitch by combining one big U and a small U brackets, as shown in figure 4.12

Figure 4.12 U Bracket Assembly

- 4 Big U brackets are fixed as shown to get **U bracket Assembly** and that is fixed to the **Robot Feet Assembly** with the help of servo horns with threads inside and without threads inside.
- Now again U bracket assembly is done with 2 big u brackets and 2 small U brackets and those are fixed with servo motor to get **servo motor and bracket assembly** which is fixed to Robot Feet Assembly
- One more Servo Motor and Bracket Assembly is fixed as assembled in previous step and update robot feet assembly

Figure 4.13 : Updated Robot Feet Assembly with knee and ankle joints

- Now 2 Aluminium plates are fixed to 2 big U brackets and fixed to servo motors
- 4 Servo motors are fixed to one Aluminium Frame and 2 Big U brackets with servo horns with threads inside are fixed to top motors.
- U bracket assembly is done with 2 small U brackets and 4 servo motors are fixed to get **Servo Motor and Bracket Assembly.**

- Similar Servo Motor and Bracket Assembly is used as hip pitch joints. This completes the **Leg Assembly**
- Now 2 big U bracket and 2 hand brackets are fixed and again to **Servo Motor and Bracket Assembly** to get **hand assembly** as shown in fig 4.14

Figure 4.14 Updated Hand Assembly

- The above hand assembly is fixed to the body i.e aluminium frame through big U brackets
- Small U bracket is used as head with servo motor fixed.
- After both the Leg Assembly and the Hand Assembly are complete, the body assembly is started, by attaching 4 Servo to the body frame.
- The body frame is attached all the other assemblies to complete the bot, as shown in the figure below

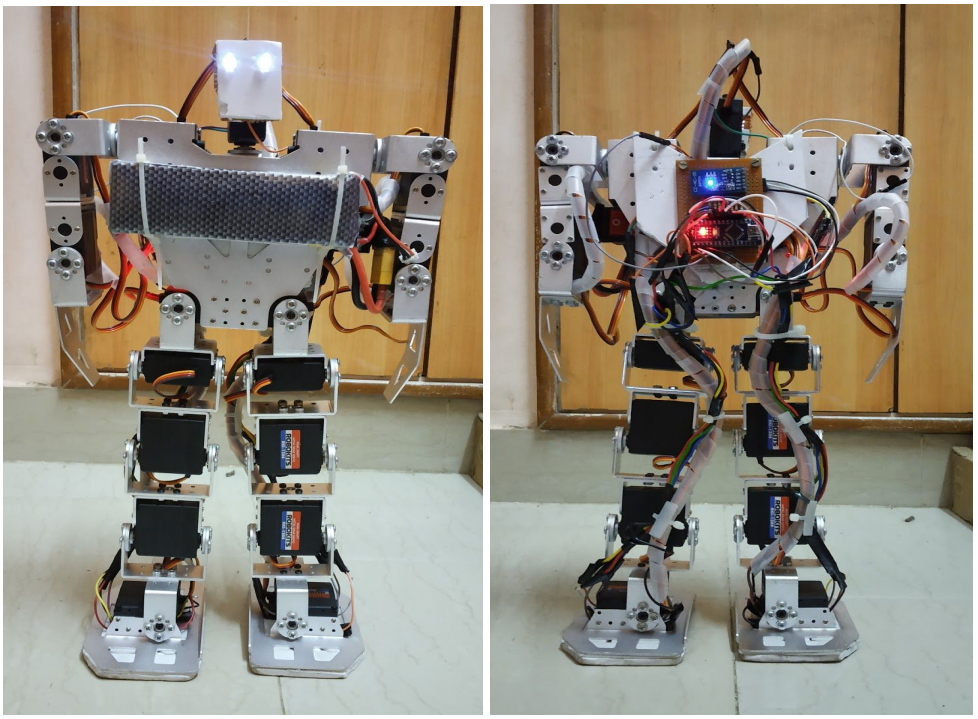


Figure 4.15 : Completed Robot - TONY (left-front view, right -back view)

There are few notable additions to the robot as shown in the figure 4.13. The robot's head has LED's added to it. They are remotely controllable and do not have any special function. The robot's foot has FSR's attached to it and they are controlled by an Arduino Nano and a second bluetooth module which are fit behind the robot. The lithium polymer is also visible in the front. This placement is to counteract the weight of the Arduino Nano at the back.

4.2.3 Electronic Design

There are two independent electronic systems in the robot. The block diagram of the both the systems are shown below. The first system is the main servo controller which actuates all the servo according to the commands received. There a mode switch(SW1) which switches between pc controlled and function controlled mode. The main microcontroller used is the Atmega328p. The block diagram is shown in figure 4.16.

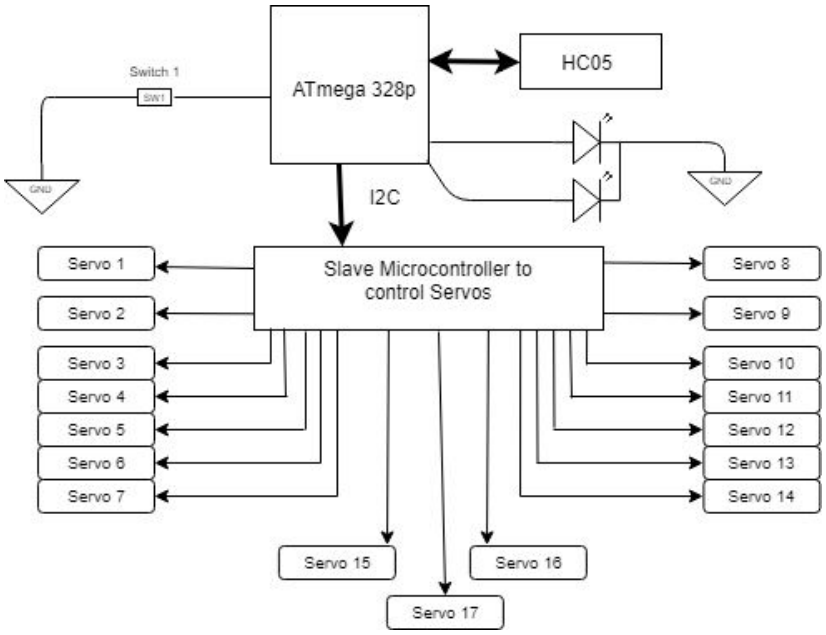


Figure 4.16 : Electronic system Block Diagram

The second electronic system that exists is the FSR system which gives feedback on the stability, there are 8 FSRs which are attached to the foot of the robot. Each of the FSR has a pull-up resistor attached. The Nano communicates with PC using HC-05 Bluetooth module. The resistance is measured through analog pins and later using the measured value, the force is calculated for later uses. The figure 4.17 shows the block diagram for FSR measurements.

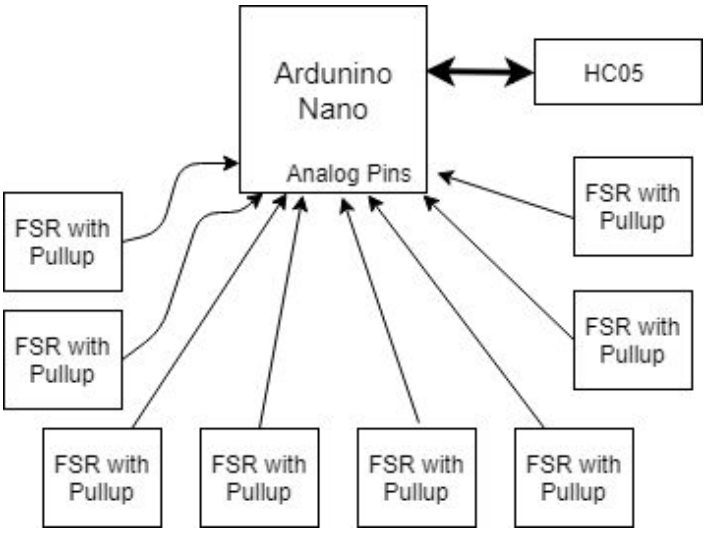


Figure 4.17 : FSR System block diagram

4.3 Software Architecture

4.3.1 Softwares Used

1. **Arduino IDE** : The Arduino IDE is a cross-platform application based on JAVA and Processing. The code editor is bundled with features such as syntax highlighting, brace matching, and automatic indentation. Compiling and uploading the code to a board can be done at single click. It eliminates the need for makefiles or run programs on a command-line interface.
2. **Python** : Python 3.7 is a high level programming language which emphasizes code readability.

   The libraries used are Matplotlib, SciPy, NumPy, PySerial

Matplotlib	a plotting library for python
SciPy	a library for scientific and technical computing
NumPy	Fundamental package for scientific computing
PySerial	Helps encapsulating the access to serial port

3. **Android App Controller** : Android App controller was downloaded from the Play store to control the bot in the 2nd mode. It has 10 buttons and the buttons were customizable.

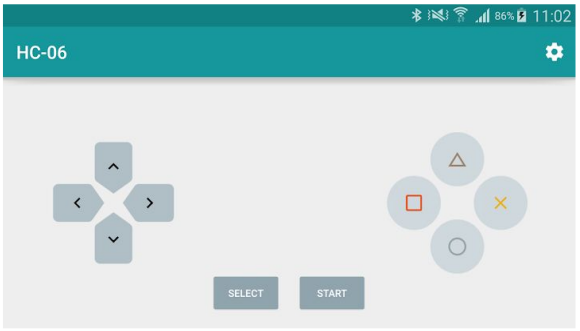


Figure 4.18 : Android App Remote Controller

4. **Robokits 18 Servo Controller Software** : Servo Controller software is an easy software to use as one can add servo sequences directly without actually programming the board and easy to generate Arduino code for the generated servo sequences which can be deployed to the Servo Controller board which makes the humanoid robot autonomous.

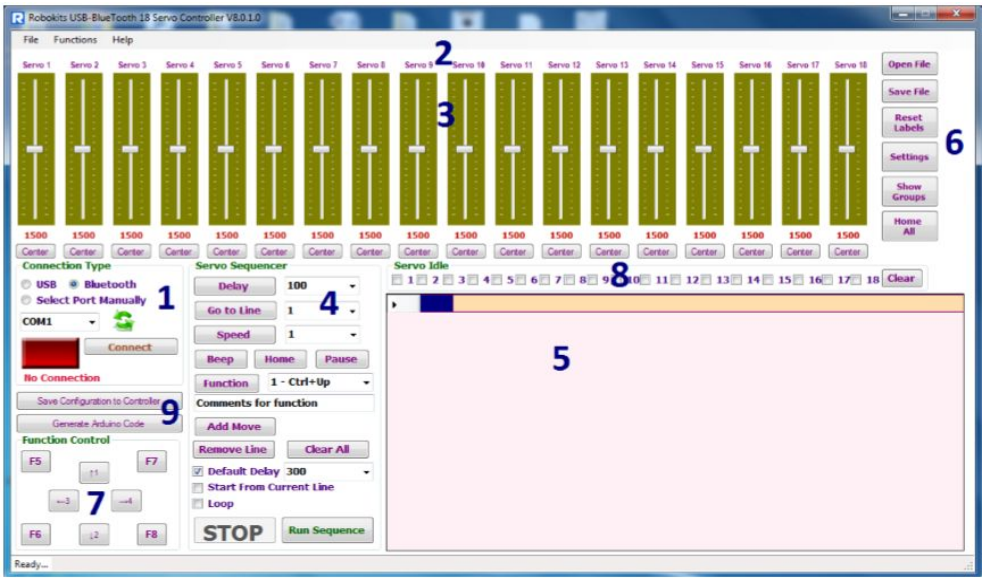


Figure 4.19 : Robokits 18 Servo Controller Software

No	Section with its Functions
1	Servo Controller Connection Section-This is for connecting servo controller with the software
2	Servo Label Section-Contains label for each servo motor
3	Servo Control Section-Controls the servos as there is a slider with pwm values from 500 µ sec to 2500 µ sec
4	Servo Sequencer-Section for creating the servo sequences and controlling the delay, speed etc
5	Sequence Window-Window for displaying the servo sequences
6	Servo Settings and Other Options-Changing the settings of the servo parameters for each servo motor

7	Function Control Section-To create the functions according the requirements while generating servo sequences
8	Servo Idle Section-To choose servos idles while adding move
9	Saving Configuration and Generating Arduino Code-For generation of Arduino code and saving configuration

4.3.2 Communication Model

The robot communicates continuously with several devices at the same time. The servo controller has to receive data from either the PC or the Android App to perform operations such as walking etc. This is done via serial communication.

1. **Communication in Servo controller** : The servo controller has 2 channels of communication, one using USB and other using bluetooth module HC-05. Both of these channels using UART as the communication protocol. The servo controller has a switch which can switch between these two modes. The servo controller runs at a baud rate of 115200 bits per second, as the data being sent is very large and requires faster update time. The servo controller has 2 different modes of operations. Both have different amounts of serial data incoming. It will be explained in next section.
2. **Communication in Arduino Nano** : The main goal of nano is to transfer the FSR data to a PC where the ZMP can be calculated and plotted. The HC-05 module is used for this purpose. The device operates at a baud rate of 9600 bits per second. This is sufficiently enough as only around 20 bytes of data is transferred per second. In this case, only the Arduino Nano sends the data and no data is being received.

4.3.3 Modes of Operation

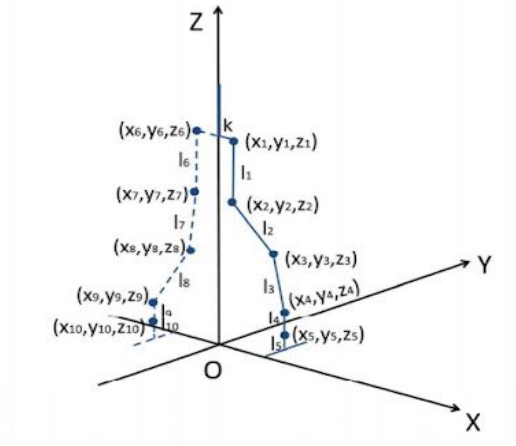
As mentioned before, the servo controller has 2 modes of operation, controlled by changing the input at D11 of the controller. Keeping D11 at 5V

switches the mode to PC controlled mode and if D11 is grounded the servo controller goes to function control mode. The description about each mode is given below.

1. **PC controlled mode** : In this mode, the servo controller accepts commands from the Robokits 18 servo controller software. Here, each and every servo can be controlled individually using a slider. This mode is used to create sequences which can be done intuitively without much calculation. It's the best preferred way to debug the controller in case of any issues. The secondary functionality like welcome gesture, pushups, squats were created using this mode. After a sequence has been created, it can be saved on the robot easily within few steps.
2. **Function controlled mode** : In this mode, the servo controller accepts input from an android app and executes sequences that are pre-loaded into the EEPROM. All the walking sequences are generated in python, so they are pre-stored in the controller's memory and executed using this mode. The mode is only to execute already store tasks. There are 8 functions defined in the memory. 3 speeds of walking forward, turn left and turn right. It also has pushups, squats and welcome gesture. Even the LED lights and the head can be controlled in this mode.

4.4 The 3D LIP Algorithm and parameters

The 3D Linear Inverted pendulum model was briefly explained in section 3.2. It was shown that the CoM trajectory represents a hyperbola and the foot positions have to planned along with it. In this section we clearly explain all the equations and parameters that are used to plan the gait. The major notations are explained in the following figures.



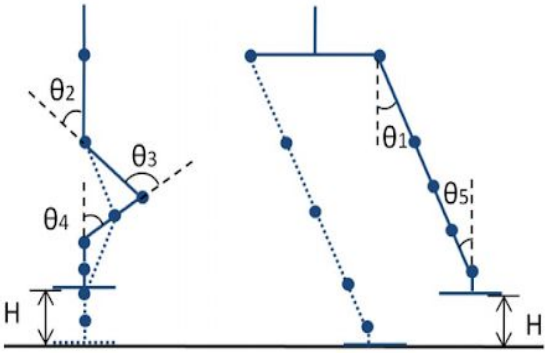


Figure 4.20 :Coordinate system of robot leg(left).

Figure 4.21 :The definition of angle variable at leg joints(right).

The first figure defines the coordinate frames for the robot's legs. The Y axis is assumed to be the direction in which the robot is walking, the Z axis is perpendicular to the floor. The X axis is completed with the help of Right Hand Rule. The point k is the point where the CoM lies, it is the point through which the fixed plane passes through. The right legs position are indicated using (x_1, y_1, z_1) to (x_5, y_5, z_5) while the left legs use (x_6, y_6, z_6) to (x_{10}, y_{10}, z_{10}). The joint angles for right leg are from θ_1 to θ_5 and the left leg are θ_6 to θ_{10} . These angles are in 3D-space and the sign of these depend on the mechanism construction.

4.4.1 Gait planning to Generate Forward Walking Motion

The entire walking motion is assumed to be a cyclic motion of time period T. This time period defines the number of stances in between. The entire cycle is divided into 4 stages. Each stage has its own equations. Each leg can either be in support phase or swing phase. Swing Phase is the stage where the foot moves from the previous step position to its next position. In support phase, the foot is held constant while other leg moves. The 4 stages of the sequence are :

1. At time t = 0, the centroid is located in between the legs. The robot squats down and prepares for the motion. The height of the centroid does not change afterwards and is kept constant. The right leg is ahead and the left leg is behind.

2. At time t = T/4, the right leg goes to support phase and the left leg is in swing phase. The centroid shifts towards the right and the left leg swings forward while attaining the maximum height.
3. At time t = T/2, the centroid comes back in the middle of two legs, but now the left leg is forward and right leg is behind.
4. At time t = 3T/4, the left leg goes into support phase as the centroid shifts left and the right legs swings forward to reach the maximum height in the swing phase.
5. At time t = T, the centroid comes back to the centre of two legs with the right leg forward and left leg backward.

This cycle is repeated continuously to obtain a walking sequence. For walking backward, this algorithm is followed in reverse. The entire walking is performed while the bot is slightly in squat position. This increases stability of the walk. This sequence is shown in the below figure.

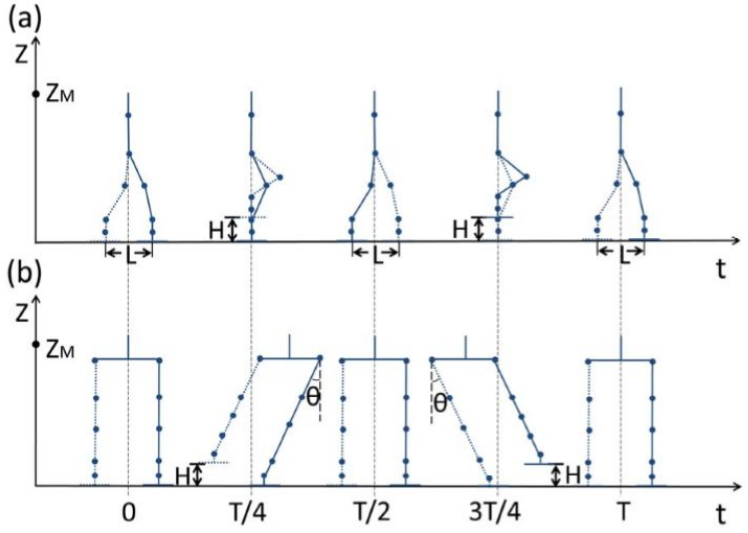


Figure 4.22 : Gait Planning using 3D LIP Model

The equations of motions of the hip and ankle positions as per the planned gait are mentioned below. Phase 1 has right leg in support and left leg in swing phase. Phase 2 has right leg in swing and left leg in support phase.

	Phase 1 { 0 - T/2 }

Right Leg : Support Phase	**Hip Roll**	$x_1 = K[sin(2\pi t/T) + 1]$ $y_1 = C_1 e^{\sqrt{K}t} + C_2 e^{-\sqrt{K}t}$ $z_1 = Z_m$
	Ankle Roll	$x_5 = K$ $y_5 = z_5 = 0$
Left Leg : Swing Phase	**Hip Roll**	$x_1 = K[sin(2\pi t/T) - 1]$ $y_1 = C_1 e^{\sqrt{K}t} + C_2 e^{-\sqrt{K}t}$ $z_1 = Z_m$
	Ankle Roll	$x_5 = K$ $y_5 = L/2[sin((2\pi(t + T/2)/T) + \pi/2]$ $z_5 = H[1 + sin((4\pi(t + T/2)/T - \pi/2)]/2$

	Phase 2 {T/2 - T}	
Right Leg : Swing Phase	**Hip Roll**	$x_1 = K[sin(2\pi t/T) + 1]$ $y_1 = C_1 e^{\sqrt{K(t-T/2)}} + C_2 e^{-\sqrt{K(t-T/2)}} + L/2$ $z_1 = Z_m$
	Ankle Roll	$x_5 = K$ $y_5 = L/2[sin(2\pi t/T + \pi/2] + L/2$ $z_5 = H[1 + sin(4\pi t/T - \pi/2)]/2$
Left Leg : Support Phase	**Hip Roll**	$x_1 = K[sin(2\pi t/T) - 1]$ $y_1 = C_1 e^{\sqrt{K(t-T/2)}} + C_2 e^{-\sqrt{K(t-T/2)}} + L/2$ $z_1 = Z_m$
	Ankle Roll	$x_5 = K$ $y_5 = L/2$ $z_5 = 0$

Note that here **K** is the half the distance between legs, **L** is the stride distance and **H** is the maximum foot lift height. The constants C_1 and C_2 are defined as follows.

$$C_1 = Lsech(\sqrt{K}T/2)/4 \qquad C_2 = -Le^{\sqrt{K}t}\, sech(\sqrt{K}T/2)/4 \quad .. (4.3)$$

These equations only give the trajectories of hip and ankle roll joints. Hence to calculate the the other positions of the links we have certain constraints. These constraints are based on few assumptions. Each link moves in a circle. Hence we get the 4 equations as

$$(x_i - x_{i+1})^2 + (y_i - y_{i+1})^2 + (z_i - z_{i+1})^2 = l_i^2 \quad \text{where } i = 1 \sim 4 \quad .. (4.4)$$

The pitch joint and roll joint are always perpendicular to the ground and when we can always assume that $y_1 = y_2,\ y_4 = y_5$. The similar assumptions are also applicable to the left leg($i = i + 5$). The next assumption is that the projections of the two legs are always parallel in the XOZ plane. This leads us to equations shown below.

$$\frac{z_j - z_{j+1}}{x_j - x_{j+1}} = \frac{z_{j+1} - z_{j+2}}{x_{j+1} - x_{j+2}} \quad \text{where } j = 1 \sim 3 \quad .. (4.5)$$

Using all these equations, we can obtain all the positions of each and every link in the 3D space. These positions have to be converted into joint angles, hence we next apply inverse kinematics on this.

4.4.2 Application of Inverse Kinematics

In the previous method we found out the way to derive all the trajectories of the walking sequence. This data is in the cartesian space which has to be converted into joint space. Hence inverse kinematics is applied here. There are 2 different issues involved here. We first use geometric approach to find the inverse kinematics. After we get the angles, as defined in the figure 4.21. The angles are defined in the real 3D space. The equations to calculate them are shown below.

θ_1	$tan^{-1}(x_1 - x_2/z_1 - z_2)$
θ_2	$abs(tan^{-1}(y_3 - y_2/z_2 - z_3))$
θ_3	$abs(\theta_4) + abs(\theta_2)$
θ_4	$abs(tan^{-1}(y_3 - y_4/z_3 - z_4))$
θ_5	$-\theta_1$

θ_6	$tan^{-1}(x_6 - x_7/z_6 - z_7)$
θ_7	$abs(tan^{-1}(y_8 - y_7/z_7 - z_8))$
θ_8	$abs(\theta_9) + abs(\theta_7)$

θ_9	$abs(tan^{-1}(y_8 - y_9/z_8 - z_9))$
θ_{10}	$-\theta_6$

The issue with these angles are that they can't be directly be sent to the robot. The servos angle range is between 0 and 180 degrees, and the way they are connect with 3D space depends on the orientation of the servo as it is attached on the robot. We had to map the real world angles into servo usable angles. The figure shows how the leg's servo angles are oriented. These are then carefully mapped from 3D angles to servo angles.

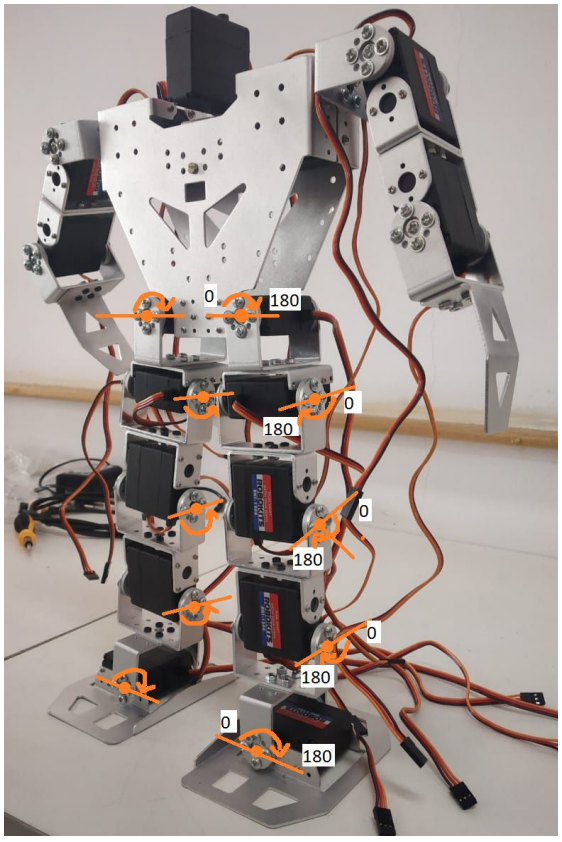


Figure 4.23 : Servo angles on the robot.

Hence, using these as a reference, the mapping is done as shown below.

θ_1'	$\pi/2 - \theta_1$
θ_2'	$\pi/2 + \theta_2$

θ_3'	$5\pi/6 - \theta_3$
θ_4'	$\pi/2 + \theta_4$
θ_5'	$\pi/2 + \theta_5$
θ_6'	$\pi/2 - \theta_6$
θ_7'	$\pi/2 - \theta_7$
θ_8'	$\pi/6 + \theta_8$
θ_9'	$\pi/2 - \theta_9$
θ_{10}'	$\pi/2 + \theta_{10}$

The obtained new angles can directly be sent to the robot which will set the posture of the robot as was calculated.

4.4.3 Controlling Speeds and other Parameters.

There are multiple parameter which can control the speed of walking of the robot. Here we discuss those parameters to improve the walking speed of the robot. The major parameter which controls the speed is the stride length. The stride length decides how much the robot travels in each step. The robot sequences were generated for 3 different stride lengths 4 cm, 6 cm and 8cm. The speeds were different but not very significant.

The next parameter which affects the walking pattern is the centroid height. Having a low centroid height makes the robot squat little too much, this makes the robot unstable as the CoM is lower and toppling becomes easier. If the centroid is too high, the robot does not squat and the gait generated is not a correct walking sequence. Hence the centroid must be optimal. We used a centroid height of 21 cm after testing various different once and arriving at this optimal one.

The step height is another such parameter as improper values cause the robot to topple, as it decides the shift of CoM to side ways. The step height used by was 2cm and

was sufficient to get good walking gait. The Time Period decides number of intermediate steps in the gait. Having a short time period leads to robot moving in a discontinuous jerky motion. The parameters used for generation are shown below.

Stride Lengths	4 cm, 6cm and 8cm
Time Period	1 sec(20 incremental steps)
Centroid Height	21 cm
Step Height	2 cm

Table 4.5 : Parameters used for Gait generation.

4.4.4 Achieving Turning Motion

We have seen in the previous sections about the gaits for forward walking. But to achieve a turning gait, the most widely used method to have an hip-yaw DoF. As we do not have this we exploit the surface's friction to turn. This is known as SLIP motion. This involves the robot following the same gait sequence for only half the time period, and then goes back to initial position. This leads to robot's feet slipping and leading to a turn.

The amount of turn is not predictable with this model as it depends on the friction between the ground and the foot. Anyway, the sequence can be repeated until the bot reaches a desired angle which can be measured using an IMU and its feedback can help use this model of turning in a precise manner.

Repeating only Right support - Left swing leads to a left turn and similarly, Left Support and Right Swing leads to a right turn.

4.5 Stability Calculations

Real environments contain unmodelled errors, terrain unevenness, the humanoid following a given pattern suffers a rapid evolution of errors between the actual state and the reference state and falls down in few steps, so we need a stabilizer. For

stabilising control we use ZMP controller. The principle of ZMP controller is that calculating ZMP with the help of Force Sensitive Resistors (FSRs).

4.5.1 FSR Placements

Four FSRs are attached to the base of each foot. **Figure 4.24** shows the FSR sensor placements. The forces acting on the foot get distributed around the ZMP. If the position of FSR sensors not being near to the core of the foot, the FSR data will not be good enough to be detected. That's what we experienced here. Hence, the FSR placements was changed as shown in **Figure 4.25**. Here, the FSRs are close to the center of the foot and hence, the ZMP measured will be more precise and useful.

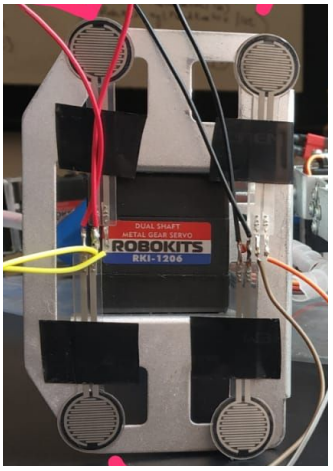


Figure 4.24 Old FSR Sensor Placements

Figure 4.25 New FSR Sensor Placements

4.5.2 Calculating Stability

Stability is calculated using ZMP data which we get from FSR sensors data and quiver plot is used to analyse the stability of the humanoid during different phases of gait. Coordinates of ZMP can be calculated as mentioned in [3].Consider a robot foot as shown in figure 4.26. The foot has 4 FSR sensors at each corner located at an offset from each corner. All the distances are measured and stored.

The ZMP of each foot is calculated using the following equations[3].

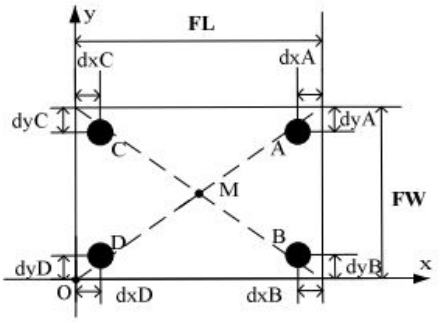


Figure 4.26 : FSR placements at point A, B, C and D

$$x_{ZMP} = \frac{f_A x_{P_A} + f_B x_{P_B} + f_C x_{P_C} + f_D x_{P_D}}{f_A + f_B + f_C + f_D} \quad .. (4.6)$$

$$y_{ZMP} = \frac{f_A y_{P_A} + f_B y_{P_B} + f_C y_{P_C} + f_D y_{P_D}}{f_A + f_B + f_C + f_D} \quad .. (4.7)$$

Where f_A , f_B , f_C, f_D are the forces at point A,B,C,D respectively.

4.5.3 Data Flow

The given circuit in **Figure 4.27** shows a simple way to connect FSR sensors using pull-up method.The analog voltage is fed to the analog pins of the Arduino Nano. The pullup used for 10k ohms. The Nano reads these analog data and sends it to PC using a HC-05 module via serial communication.

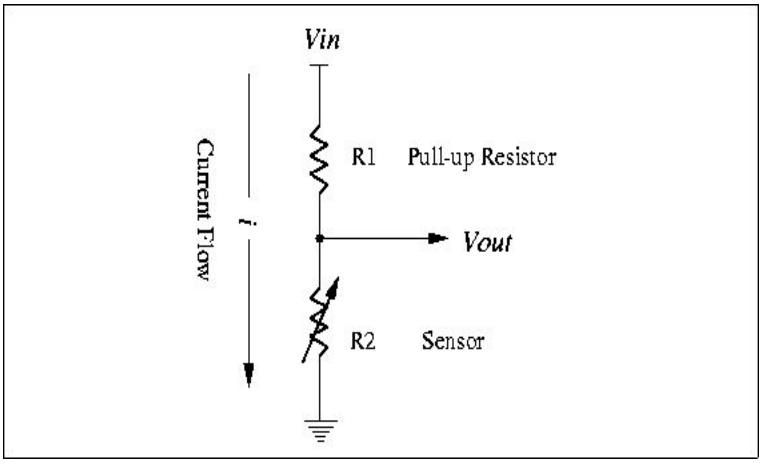


Figure 4.27: Connecting FSR sensors using pull-up method

The analog data received is then converted to voltage value using basic mapping techniques. The voltage is then used to find out the resistance and conductance of the FSR using voltage divider rule. This resistance can be mapped to force using the plot shown in figure 4.6 This forces are individually plotted on a quiver graph and then the ZMP is calculated and plotted. If the major arrows are towards the CoM of the robot, it indicates that the robot is stable. Figure 4.26 shows the data flow from FSR.

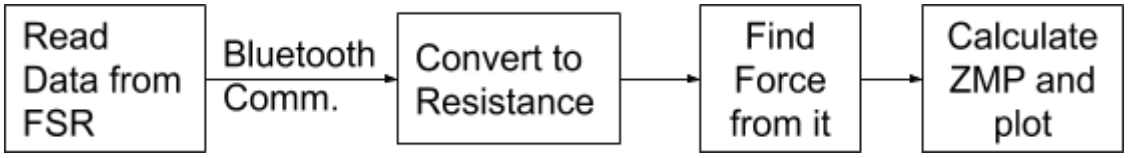


Figure 4.28 : Data Flow to calculate ZMP

5 Results

The generated gaits were tested on the built humanoid robot. The robot performed upto the expectation. The results obtained are described below. The main operations that were performed on the robot were, walking forward, turn left and right. The secondary operations tested were pushups and squats. The main discussion will be about walking forward as the gait was generated primarily from the 3D LIP model and it needs to be validated in the real world.

5.1 Simulation Outputs

The 3D LIP model was first simulated on python and the trajectories were plotted using matplotlib. The below figures show the simulated plots obtained using python. The figure 5.1 shows the gait generated using 3D LIP model. The right leg in support phase is indicated in blue, while the left leg in support is shown with green. Red indicates the right leg in swing phase while, yellow is left leg in swing phase. The figure 5.1 is the entire gait in 3D view. The figures 5.2, 5.3, 5.4 show the gaits from different views namely front, side and top view respectively. The next figure 5.5 indicates the plot of every link angle that right leg has over the entire time period T.

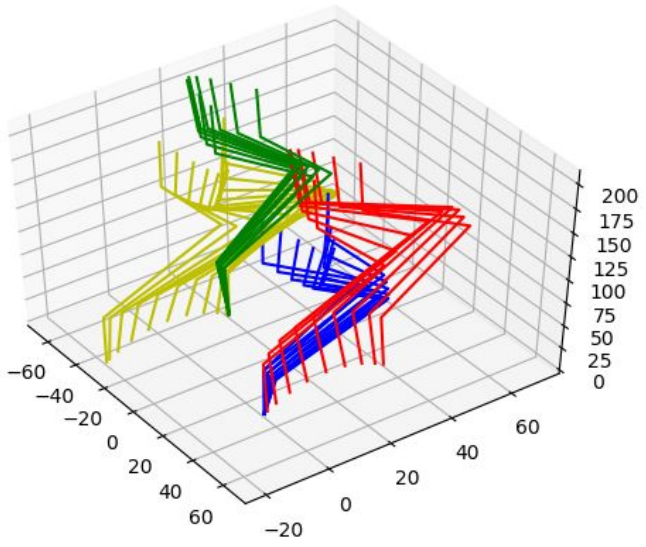


Figure 5.1 :3D LIP trajectory

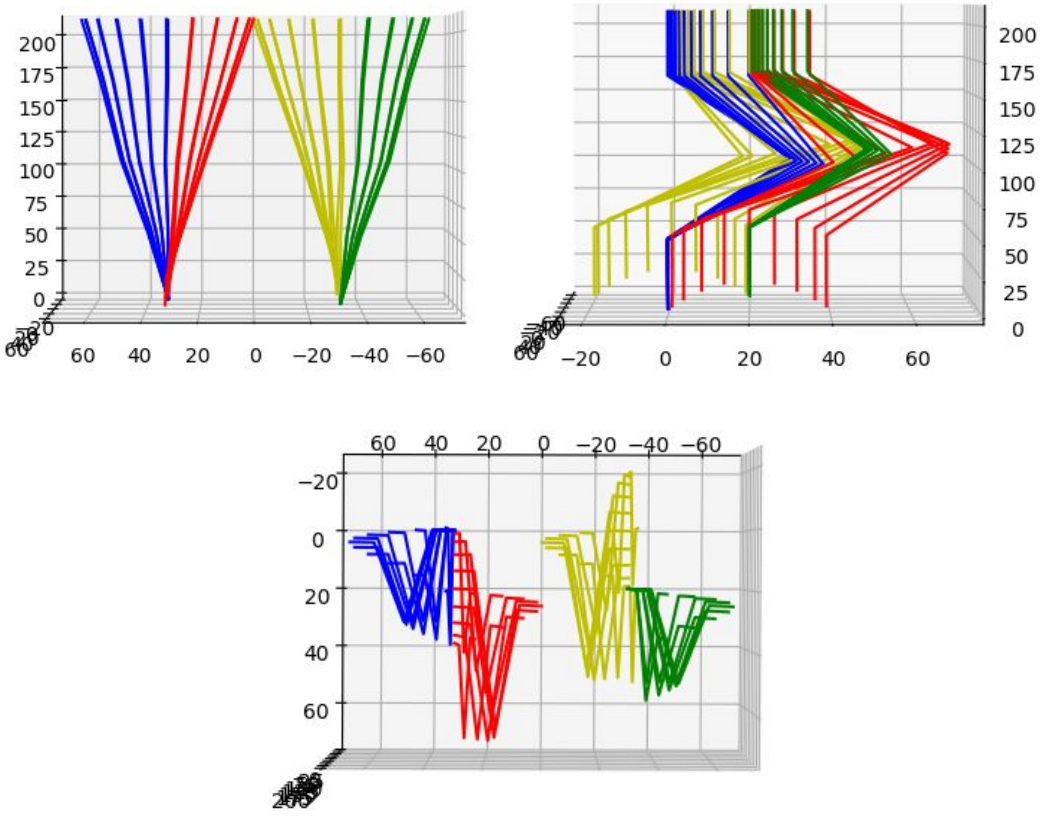


Figure 5.2,5.3,5.4 : 3D LIP front, side and top view

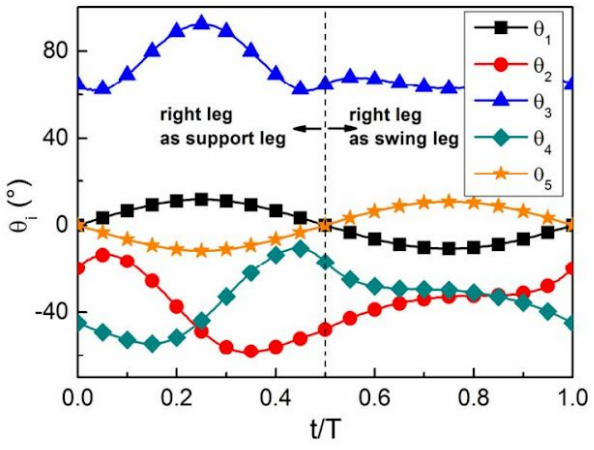
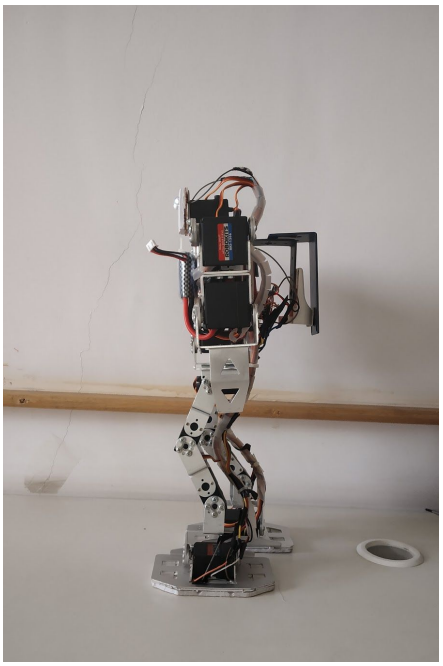
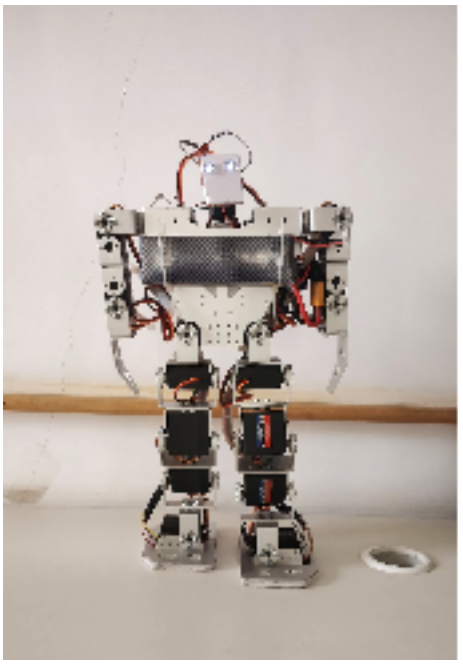
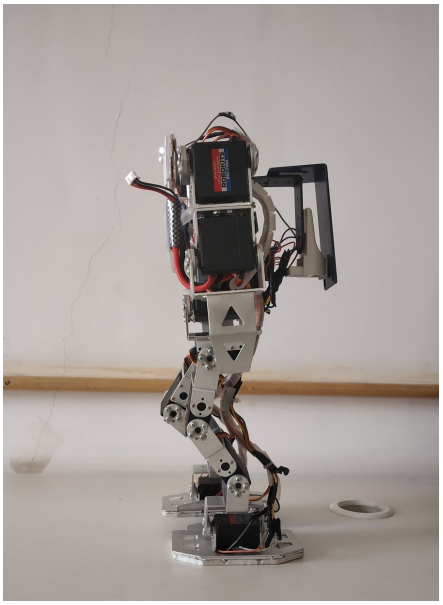
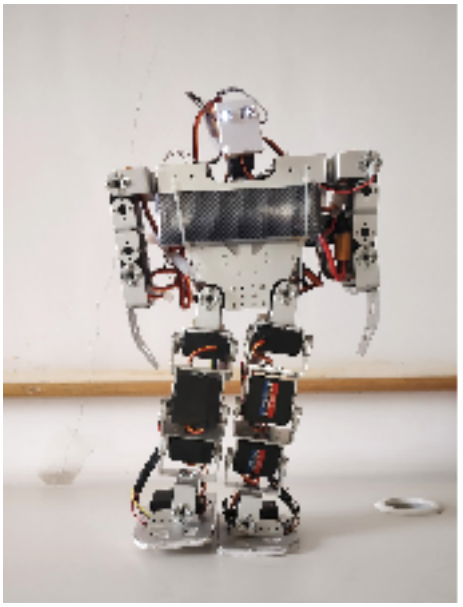


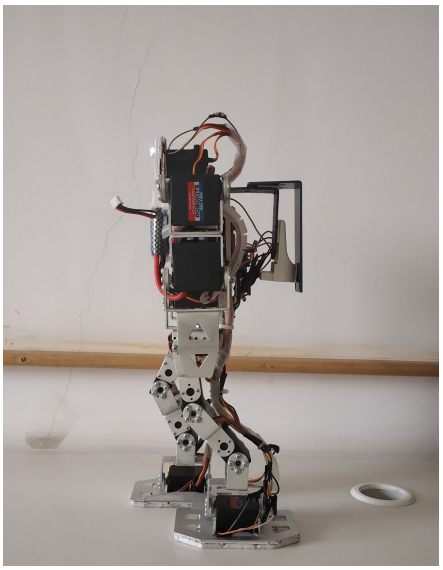
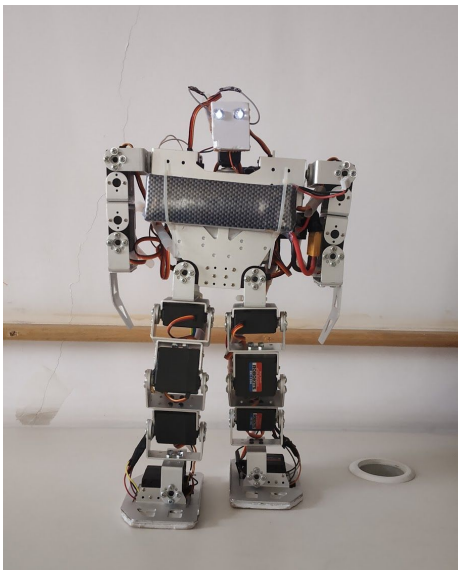
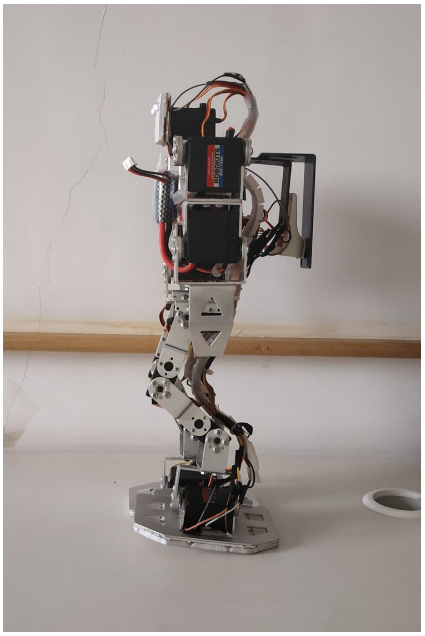
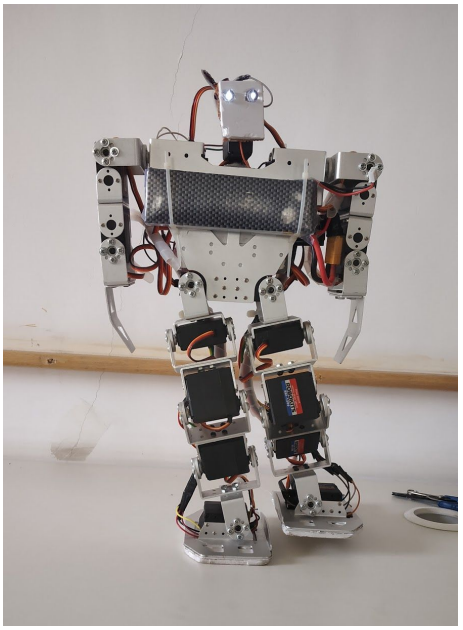
Figure 5.5 : Plot of all the link angles of right leg during entire T

5.2 Walking Achieved by Humanoid

The generated sequences were run on the humanoid robot, and the robot is able to walk similar to a human. There are a few issues with the weight of the robot which led to tiny inertial movement which did not affect the walking motion of the robot. The

table 5.1 indicated below, shows all the stances of the robot at the key time points during the walking cycle.

Time	Side View	Front View
t = 0		
t = T/4		

t = T/2		
t = 3T/4		

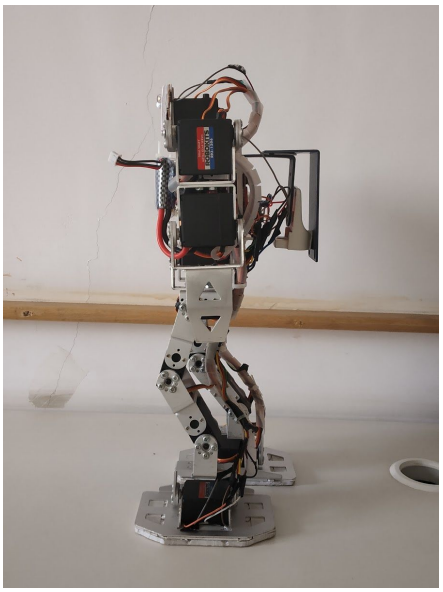
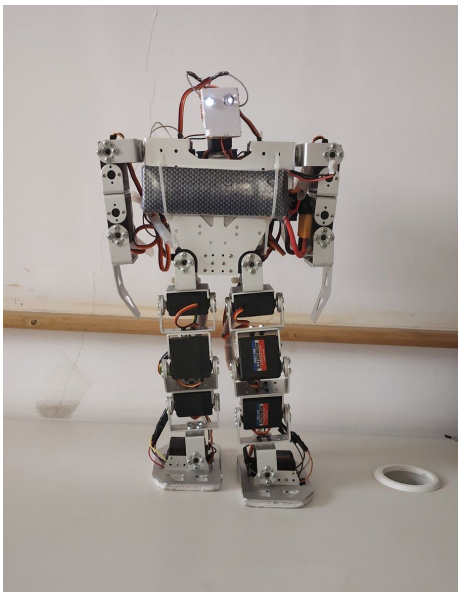
t = T		

Table 5.1 : Stances of robot at t=0,T/4,T/2,3T/4 and T

5.3 Stability Plots

The values obtained from the FSR were plotted using a matplotlib quiver plot. The left white square represents the left foot and similarly for the right foot. The forces from the FSR are indicated by the blue arrow. The taller the arrow, the more force is being applied. The below figures show the plots obtained from FSRs.

The first figure 5.6 shows the FSR plot for the robot when its standing still with both feet on the ground. It is clearly visible that the forces on both feet are almost distributed. And the average seems to sum up to the center. This indicates that gait is stable.

The figure 5.7 shows FSR plot for left leg in support phase and right leg in swing phase. It is clearly evident that there is more force being applied on the left foot as it supports the entire structure during the support phase. The right leg has negligible forces acting on it.

The figure 5.7 shows FSR plot for right leg in support phase and left leg in swing phase. It is clearly evident that there is more force being applied on the right foot as it supports the entire structure during the support phase. The left leg has negligible forces acting on it.

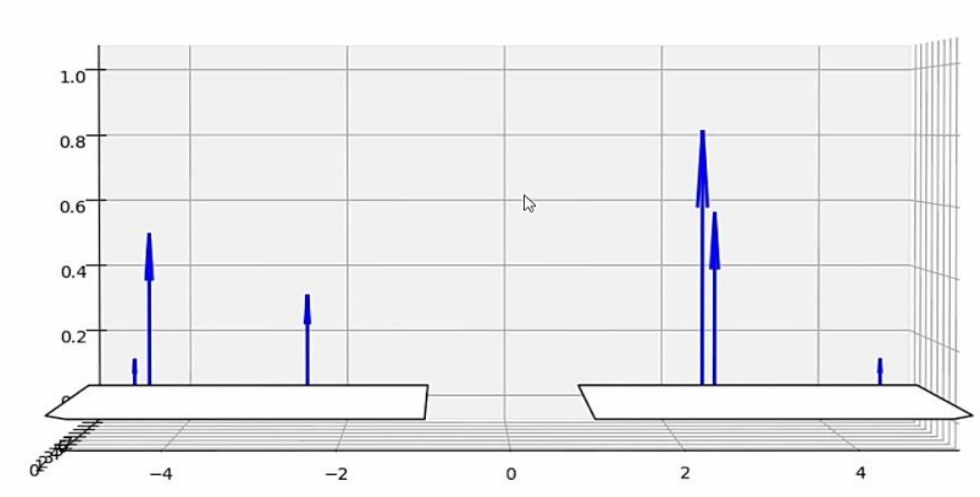


Figure 5.6 : FSR plot for robot standing still

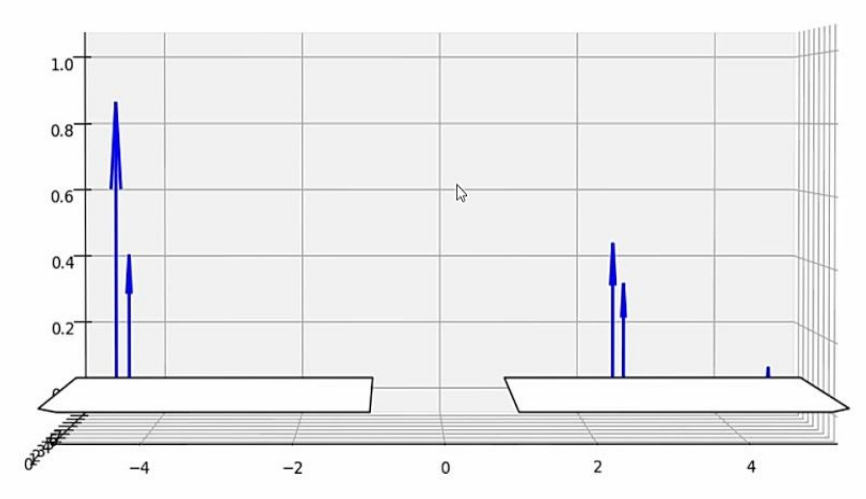


Figure 5.7 : FSR plot for Left Support and Right swing.

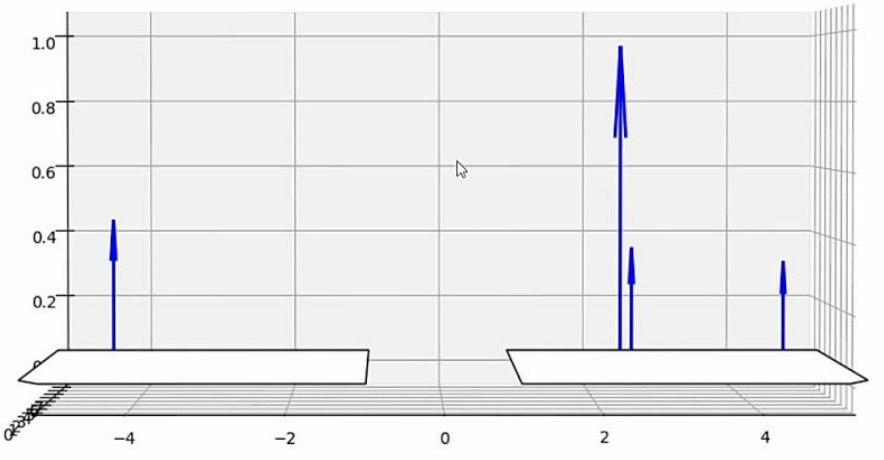


Figure 5.8 : FSR plot for Right Support and Left swing.

5.4 Secondary Gaits

The humanoid was also tested with gaits which were generated using the Robokits 18 servo controller gui software. The two gaits that were generated was pushups and squats. These gaits are hard to perform even for humans. Hence the capabilities of robot is tested using these gaits. The figure 5.10 shows the sequence of stances performed by the bot to perform squats. The figure 5.9 shows the sequence of stances performed by the robot for pushups.

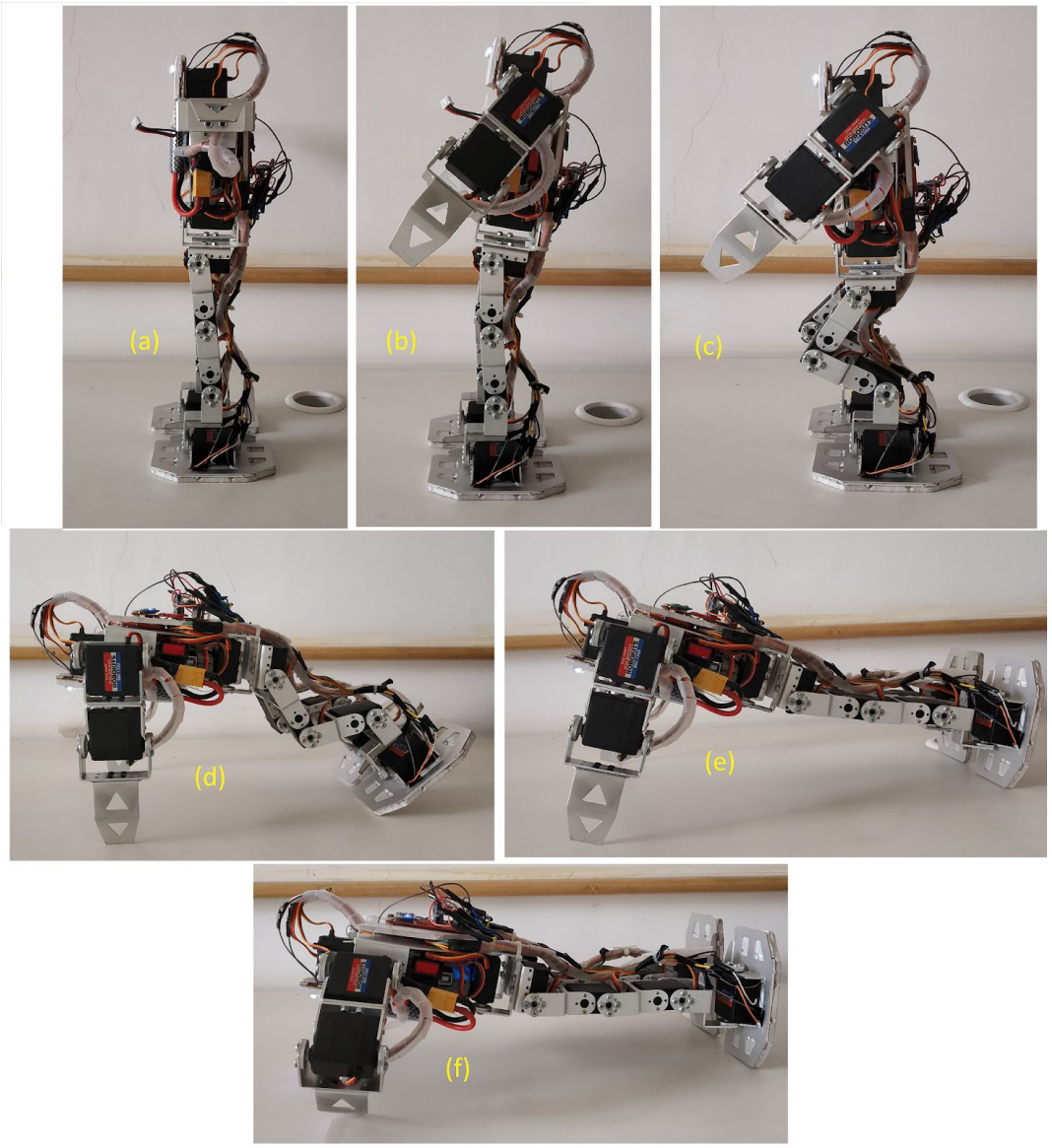


Figure 5.9(a)(b)(c)(d)(e)(f) : Stances while performing push ups (In-order)

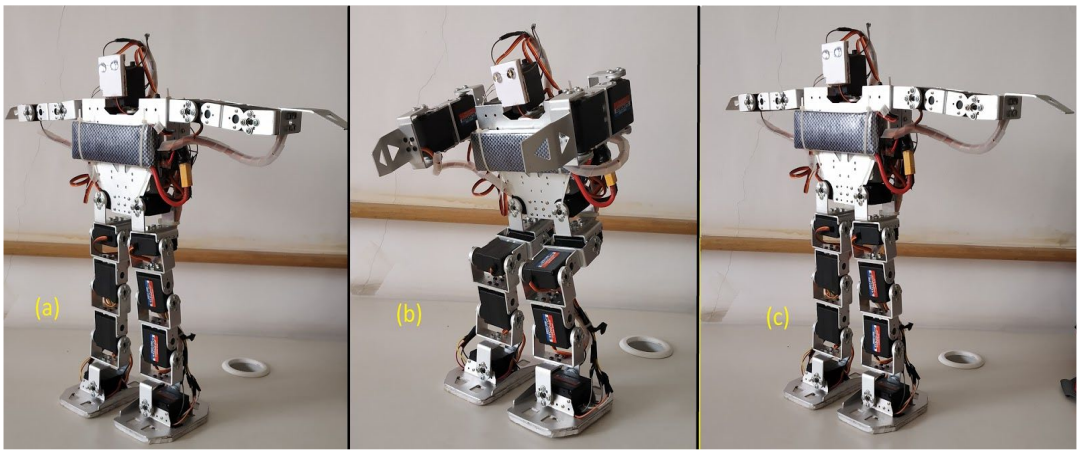


Figure 5.10(a)(b)(c) : Stances while performing Squats (In-order)

6 Conclusions and Future Scope

6.1 Conclusions

In this project, controlled gait generation is achieved for a 17 DOF humanoid robot with stability by solving inverse kinematic equations. The humanoid is approximated to be 3D Linear Inverted Pendulum Model to generate walking patterns.A simulated model has been established with the help of Python. The trajectory of the swing foot and the centroid was matching the hyperbola curve. FSR sensor data was used to plot stability plots and were verified on humanoid during walking. The humanoid is completely navigable from an android app.

The robot is also able to perform several other tasks other than walking forward. It is able to turn left and right. It can also perform gaits such as push ups and squats. All these gaits are stable and do not topple the robot. The robot also has two modes of operations for testing and developing new algorithms. The robot is also able to plot the data coming from the FSR using a bluetooth module. Overall, the goal of the project has been achieved.

6.2 Future Scope

There are several regions of study where this robot can be expanded to. The robot is able to walk without any feedback from sensors. Using the feedback of sensors for stabilizing the gait is one option which can significantly increase the gait stability. The robot can be developed as a research platform by upgrading the hardware so that it can be used by future teams for further development.

Further study will focus on addition of toes to robot's foot and make it counteract the disturbances and use of an on-board computer for real time generation of sequences to get more realistic walk. Establishing coordination between hands and legs for a wide range of motions. The scope of possibilities for future work on the humanoid is endless and there are multiple ways of improvement in the robot whether it be improving the gait algorithm to upgrading the hardware.

7 References

[1]J.M. Ibarra Zannatha, R. Cisneros Limon - Forward and Inverse Kinematics fora Small-Sized Humanoid Robot - 2009 International Conference on Electrical, Communications, and Computers,Cholula, Puebla Mexico.

[2] Ying Zhang, Shuanghong Li, Boyu Han, and Qiaoling Du -Research on Gait Planning and Inverse Kinematics Solving of Biped Walking Robots - 2015 8th International Symposium on Computational Intelligence and Design, Hangzhou, China.

[3]Bi Sheng, Min Huaqing, Zhuang Zhongjie et.al. - Walking Control Method of Humanoid Robot Based on FSR Sensors and Inverted Pendulum Model - TAROS 2012, Bristol, UK.

[4]S. Kajita, H. Hirukawa, and K. Yokoi, K. Harada - Introduction to Humanoid Robots - Springer Tracts in Advanced Robotics,Tsinghua University Press,2007.

[5]Honda Motor Co., Ltd. Public Relations Division, “ASIMO Technical Information”, September 2007

8 Appendix

8.1 Code to generate sequences using 3-D LIP model

```
# Author : Prasanna Venkatesan
# The motion is generated for a single time period T, The repeated
# 0 : Robot Squats, Centroid in the middle, Left leg back & Right leg front
# T/4 : Centroid towards right, Right leg - support phase, Left leg - swing
phase
# T/2  : Centroid in the middle, Left leg front & Right leg back
# 3T/4: Centroid towards left, Left Leg - support phase, Right leg - swing
phase
# T : Same as T=0
import math
import time
import copy
import robot as rb
import motion as mn
import numpy as np
import scipy
import scipy.optimize
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = fig.add_subplot(1,1,1, projection='3d')
plt.ion()
# Right leg -(x1,y1,z1),(x2,y2,z2),(x3,y3,z3),(x4,y4,z4),(x5,y5,z5)
# Left leg -(x6,y6,z6),(x7,y7,z7),(x8,y8,z8),(x9,y9,z9),(x10,y10,z10)
RIGHT_LEG_JOINT_POS =
np.array([[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0
.0]])
LEFT_LEG_JOINT_POS =
np.array([[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,0
.0]])
PREVIOUS_RIGHT_JOINT_ANGLES = np.array([0.0,0.0,0.0,0.0,0.0])
PREVIOUS_LEFT_JOINT_ANGLES = np.array([0.0,0.0,0.0,0.0,0.0])
RIGHT_LEG_JOINT_ANGLES = np.array([0.1,0.1,0.1,0.1,0.1])
LEFT_LEG_JOINT_ANGLES = np.array([0.1,0.1,0.1,0.1,0.1])
# Current walk time, in seconds
CURRENT_TIME = 0
CONSTANT_C1 = 0
CONSTANT_C2 = 0

def sech(x):
    return math.cosh(x)**(-1)
def checkParametersSet():
       return rb.checkRobotParametersSet() or mn.checkMotionParametersSet()
# Set all the required parameters
```

```
def setAllParameters(pelvisDist, linkLengths, timePeriod, centroidHeight,
maxLiftHeight, strideLength):
	rb.setRobotParameters(pelvisDist, linkLengths, linkLengths)
	mn.setMotionParameters(timePeriod, centroidHeight, maxLiftHeight,
strideLength)

def calSupport(type, legPos):
	global CONSTANT_C1, CONSTANT_C2, CURRENT_TIME
	if(type == 'right'):
		#x1,y1,z1 - RIGHT HIP
		legPos[0][0] = rb.HALF_DIST_BW_LEGS *
(math.sin(2*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD)) + 1)
		rootKt = math.sqrt(rb.HALF_DIST_BW_LEGS)*CURRENT_TIME
		legPos[0][1] = CONSTANT_C1 * math.exp(rootKt) + CONSTANT_C2 *
math.exp(-rootKt)
		legPos[0][2] = mn.CENTROID_HEIGHT

		#x5,y5,z5 - RIGHT ANKEL
		legPos[4][0] = rb.HALF_DIST_BW_LEGS
		legPos[4][1] = legPos[4][2] = 0
	else:
		#x6,y6,z6 - LEFT HIP
		legPos[0][0] = rb.HALF_DIST_BW_LEGS *
(math.sin(2*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD)) - 1)
		rootKt = math.sqrt(rb.HALF_DIST_BW_LEGS)*(CURRENT_TIME -
float(mn.TIME_PERIOD)/2)
		legPos[0][1] = CONSTANT_C1 * math.exp(rootKt) + CONSTANT_C2 *
math.exp(-rootKt) + float(mn.STRIDE_LENGTH)/2
		legPos[0][2] = mn.CENTROID_HEIGHT

		#x10,y10,z10 - LEFT ANKEL
		legPos[4][0] = - rb.HALF_DIST_BW_LEGS
		legPos[4][1] = float(mn.STRIDE_LENGTH)/2
		legPos[4][2] = 0

def calSwing(type, legPos):
	global CONSTANT_C1,CONSTANT_C2
	if(type == 'right'):
		#x1,y1,z1 - RIGHT HIP
		legPos[0][0] = rb.HALF_DIST_BW_LEGS *
(math.sin(2*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD)) + 1)
		rootKt = math.sqrt(rb.HALF_DIST_BW_LEGS)*(CURRENT_TIME -
float(mn.TIME_PERIOD)/2)
		legPos[0][1] = CONSTANT_C1 * math.exp(rootKt) + CONSTANT_C2 *
math.exp(-rootKt) + float(mn.STRIDE_LENGTH)/2
		legPos[0][2] = mn.CENTROID_HEIGHT

		#x5,y5,z5 - RIGHT ANKEL
```

```
            legPos[4][0] = rb.HALF_DIST_BW_LEGS
            legPos[4][1] = float(mn.STRIDE_LENGTH)/2 *
(math.sin(2*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD) + math.pi/2))
+ float(mn.STRIDE_LENGTH)/2
            legPos[4][2] = mn.LIFT_HEIGHT * (1 +
math.sin(4*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD) - math.pi/2)) /
2
      else:
            #x6,y6,z6 - LEFT HIP
            legPos[0][0] = rb.HALF_DIST_BW_LEGS *
(math.sin(2*math.pi*float(CURRENT_TIME)/float(mn.TIME_PERIOD)) - 1)
            rootKt = math.sqrt(rb.HALF_DIST_BW_LEGS)*CURRENT_TIME
            legPos[0][1] = CONSTANT_C1 * math.exp(rootKt) + CONSTANT_C2 *
math.exp(-rootKt)
            legPos[0][2] = mn.CENTROID_HEIGHT

            #x10,y10,z10 - LEFT ANKEL
            legPos[4][0] = - rb.HALF_DIST_BW_LEGS
            legPos[4][1] = float(mn.STRIDE_LENGTH)/2 *
(math.sin(2*math.pi*(CURRENT_TIME +
float(mn.TIME_PERIOD)/2)/float(mn.TIME_PERIOD) + math.pi/2))
            legPos[4][2] = mn.LIFT_HEIGHT * (1 +
math.sin(4*math.pi*(CURRENT_TIME +
float(mn.TIME_PERIOD)/2)/float(mn.TIME_PERIOD) - math.pi/2)) / 2

def getSlope(point1, point2):
      if (point1[0] - point2[0]) == 0:
            return 'inf'
      else:
            slope = (point1[1] - point2[1]) / (point1[0] - point2[0])
            if slope > 1000 or slope < -1000:
                  return 'inf'
            else:
                  return slope
def calAllPos():
      global RIGHT_LEG_JOINT_POS, LEFT_LEG_JOINT_POS
      #-----------------------Roll Joint and Pitch joint are always
perpendicular-----------------------

      RIGHT_LEG_JOINT_POS[1][1] = RIGHT_LEG_JOINT_POS[0][1]
      RIGHT_LEG_JOINT_POS[3][1] = RIGHT_LEG_JOINT_POS[4][1]

      LEFT_LEG_JOINT_POS[1][1] = LEFT_LEG_JOINT_POS[0][1]
      LEFT_LEG_JOINT_POS[3][1] = LEFT_LEG_JOINT_POS[4][1]

      #----------------------Fixed Length Constraints and Same Line
Constraints-------------------------
      #RIGHT
```

```
	slopeRight = getSlope(RIGHT_LEG_JOINT_POS[4][::2],
RIGHT_LEG_JOINT_POS[0][::2])
	print("Right Slope :", slopeRight)
	if(slopeRight == 'inf'):
		RIGHT_LEG_JOINT_POS[1][0] = RIGHT_LEG_JOINT_POS[2][0] =
RIGHT_LEG_JOINT_POS[3][0] = RIGHT_LEG_JOINT_POS[0][0]
		#For z2
		sqrDiff = rb.RIGHT_LEG_LINK_LEN[0]**2 -
(RIGHT_LEG_JOINT_POS[0][0] - RIGHT_LEG_JOINT_POS[1][0])**2 -
(RIGHT_LEG_JOINT_POS[0][1] - RIGHT_LEG_JOINT_POS[1][1])**2
		sqrRootDiff = math.sqrt(sqrDiff)
		RIGHT_LEG_JOINT_POS[1][2] = RIGHT_LEG_JOINT_POS[0][2] -
sqrRootDiff

		#For z4
		sqrDiff = rb.RIGHT_LEG_LINK_LEN[3]**2 -
(RIGHT_LEG_JOINT_POS[3][0] - RIGHT_LEG_JOINT_POS[4][0])**2 -
(RIGHT_LEG_JOINT_POS[3][1] - RIGHT_LEG_JOINT_POS[4][1])**2
		sqrRootDiff = math.sqrt(sqrDiff)
		RIGHT_LEG_JOINT_POS[3][2] = RIGHT_LEG_JOINT_POS[4][2] +
sqrRootDiff
		def thighLink(x):
			global RIGHT_LEG_JOINT_POS
			value = (RIGHT_LEG_JOINT_POS[1][1] - x[0])**2 +
(RIGHT_LEG_JOINT_POS[1][2] - x[1])**2 - rb.RIGHT_LEG_LINK_LEN[1]**2
			return value
		def kneeLink(x):
			global RIGHT_LEG_JOINT_POS
			value = (x[0] - RIGHT_LEG_JOINT_POS[3][1])**2 + (x[1] -
RIGHT_LEG_JOINT_POS[3][2])**2 - rb.RIGHT_LEG_LINK_LEN[2]**2
			return value
		def eqnSet(x):
			return [thighLink(x), kneeLink(x)]

		solution = scipy.optimize.fsolve(eqnSet,
[RIGHT_LEG_JOINT_POS[1][1]+mn.STRIDE_LENGTH,RIGHT_LEG_JOINT_POS[1][2]-rb.RIG
HT_LEG_LINK_LEN[1]])
		RIGHT_LEG_JOINT_POS[2][1] = solution[0]
		RIGHT_LEG_JOINT_POS[2][2] = solution[1]
		print(RIGHT_LEG_JOINT_POS)

print([RIGHT_LEG_JOINT_POS[1][1]+mn.STRIDE_LENGTH,RIGHT_LEG_JOINT_POS[1][2]-
rb.RIGHT_LEG_LINK_LEN[1]])
	else:
		intercept = RIGHT_LEG_JOINT_POS[0][2] - slopeRight *
RIGHT_LEG_JOINT_POS[0][0]
		def inLine(x):
			return (x[1] - slopeRight * x[0] - intercept)
```

```
            def hipLink(x):
                  global RIGHT_LEG_JOINT_POS
                  value = (RIGHT_LEG_JOINT_POS[0][0] - x[0])**2 +
(RIGHT_LEG_JOINT_POS[0][2] - x[1])**2 - rb.RIGHT_LEG_LINK_LEN[0]**2
                  return value
            def ankleLink(x):
                  global RIGHT_LEG_JOINT_POS
                  value = (x[0] - RIGHT_LEG_JOINT_POS[4][0])**2 + (x[1] -
RIGHT_LEG_JOINT_POS[4][2])**2 - rb.RIGHT_LEG_LINK_LEN[3]**2
                  return value
            def solveHipPitch(x):
                  return [hipLink(x), inLine(x)]
            def solveAnkelPitch(x):
                  return [ankleLink(x), inLine(x)]

            solutionAnkel = solutionHip = [0,0,0]
            solutionHip = scipy.optimize.fsolve(solveHipPitch,
[RIGHT_LEG_JOINT_POS[0    ][0],RIGHT_LEG_JOINT_POS[0][2]-
rb.RIGHT_LEG_LINK_LEN[0]])
            solutionAnkel = scipy.optimize.fsolve(solveAnkelPitch,
[rb.HALF_DIST_BW_LEGS ,solutionHip[1]])
            # print("Solution Hip :: ", solutionHip)
            # print("Solution Ankel :: ", solutionAnkel)
            RIGHT_LEG_JOINT_POS[1][0] = solutionHip[0]
            RIGHT_LEG_JOINT_POS[1][2] = solutionHip[1]

            RIGHT_LEG_JOINT_POS[3][0] = solutionAnkel[0]
            RIGHT_LEG_JOINT_POS[3][2] = solutionAnkel[1]
            def thighLink(x):
                  global RIGHT_LEG_JOINT_POS
                  value = (RIGHT_LEG_JOINT_POS[1][0] - x[0])**2 +
(RIGHT_LEG_JOINT_POS[1][1] - x[2])**2 + (RIGHT_LEG_JOINT_POS[1][2] -
x[1])**2 - rb.RIGHT_LEG_LINK_LEN[1]**2
                  return value
            def kneeLink(x):
                  global RIGHT_LEG_JOINT_POS
                  value = (x[0] - RIGHT_LEG_JOINT_POS[3][0]) + (x[2] -
RIGHT_LEG_JOINT_POS[3][1])**2 + (x[1] - RIGHT_LEG_JOINT_POS[3][2])**2 -
rb.RIGHT_LEG_LINK_LEN[2]**2
                  return value
            def solveKneePitch(x):
                  return [inLine(x), thighLink(x), kneeLink(x)]

            solutionKnee = [0,0,0]
            solutionKnee = scipy.optimize.fsolve(solveKneePitch,
[RIGHT_LEG_JOINT_POS[1][0], RIGHT_LEG_JOINT_POS[1][2],
RIGHT_LEG_JOINT_POS[1][1]+ mn.STRIDE_LENGTH])
            # print("Solution Knee :: ", solutionKnee)
```

```
            RIGHT_LEG_JOINT_POS[2][0] = solutionKnee[0]
            RIGHT_LEG_JOINT_POS[2][1] = solutionKnee[2]
            RIGHT_LEG_JOINT_POS[2][2] = solutionKnee[1]


#-------------------------------------------------------------------------
-------------------------
      #LEFT
      slopeLeft = getSlope(LEFT_LEG_JOINT_POS[4][::2],
LEFT_LEG_JOINT_POS[0][::2])
      print("Left Slope :", slopeLeft)
      if(slopeLeft == 'inf'):
            LEFT_LEG_JOINT_POS[1][0] = LEFT_LEG_JOINT_POS[2][0] =
LEFT_LEG_JOINT_POS[3][0] = LEFT_LEG_JOINT_POS[0][0]

            #For z7
            sqrDiff = rb.LEFT_LEG_LINK_LEN[0]**2 -
(LEFT_LEG_JOINT_POS[0][0] - LEFT_LEG_JOINT_POS[1][0])**2 -
(LEFT_LEG_JOINT_POS[0][1] - LEFT_LEG_JOINT_POS[1][1])**2
            sqrRootDiff = math.sqrt(sqrDiff)
            LEFT_LEG_JOINT_POS[1][2] = LEFT_LEG_JOINT_POS[0][2] -
sqrRootDiff

            #For z9
            sqrDiff = rb.LEFT_LEG_LINK_LEN[3]**2 -
(LEFT_LEG_JOINT_POS[3][0] - LEFT_LEG_JOINT_POS[4][0])**2 -
(LEFT_LEG_JOINT_POS[3][1] - LEFT_LEG_JOINT_POS[4][1])**2
            sqrRootDiff = math.sqrt(sqrDiff)
            LEFT_LEG_JOINT_POS[3][2] = LEFT_LEG_JOINT_POS[4][2] +
sqrRootDiff
            def thighLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (LEFT_LEG_JOINT_POS[1][1] - x[0])**2 +
(LEFT_LEG_JOINT_POS[1][2] - x[1])**2 - rb.LEFT_LEG_LINK_LEN[1]**2
                  return value

            def kneeLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (x[0] - LEFT_LEG_JOINT_POS[3][1])**2 + (x[1] -
LEFT_LEG_JOINT_POS[3][2])**2 - rb.LEFT_LEG_LINK_LEN[2]**2
                  return value

            def eqnSet(x):
                  return [thighLink(x), kneeLink(x)]

            solution = scipy.optimize.fsolve(eqnSet,
[LEFT_LEG_JOINT_POS[1][1]+mn.STRIDE_LENGTH,LEFT_LEG_JOINT_POS[1][2]-rb.LEFT_
LEG_LINK_LEN[1]])
```

```
            # print("Left Leg :: ",solution)
            LEFT_LEG_JOINT_POS[2][1] = solution[0]
            LEFT_LEG_JOINT_POS[2][2] = solution[1]
            print(LEFT_LEG_JOINT_POS)

print([LEFT_LEG_JOINT_POS[1][1]+mn.STRIDE_LENGTH,LEFT_LEG_JOINT_POS[1][2]-rb
.LEFT_LEG_LINK_LEN[1]])
      else:
            intercept = LEFT_LEG_JOINT_POS[0][2] - slopeLeft *
LEFT_LEG_JOINT_POS[0][0]

            def inLine(x):
                  return (x[1] - slopeLeft * x[0] - intercept)

            def hipLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (LEFT_LEG_JOINT_POS[0][0] - x[0])**2 +
(LEFT_LEG_JOINT_POS[0][2] - x[1])**2 - rb.LEFT_LEG_LINK_LEN[0]**2
                  return value

            def ankleLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (x[0] - LEFT_LEG_JOINT_POS[4][0])**2 + (x[1] -
LEFT_LEG_JOINT_POS[4][2])**2 - rb.LEFT_LEG_LINK_LEN[3]**2
                  return value

            def solveHipPitch(x):
                  return [hipLink(x), inLine(x)]
            def solveAnkelPitch(x):
                  return [ankleLink(x), inLine(x)]

            solutionHip = scipy.optimize.fsolve(solveHipPitch,
[LEFT_LEG_JOINT_POS[0][0],LEFT_LEG_JOINT_POS[0][2]-rb.LEFT_LEG_LINK_LEN[0]])
            solutionAnkel = scipy.optimize.fsolve(solveAnkelPitch,
[-rb.HALF_DIST_BW_LEGS,solutionHip[1]])

            # print("Solution Hip :: ", solutionHip)
            # print("Solution Ankel :: ", solutionAnkel)
            LEFT_LEG_JOINT_POS[1][0] = solutionHip[0]
            LEFT_LEG_JOINT_POS[1][2] = solutionHip[1]

            LEFT_LEG_JOINT_POS[3][0] = solutionAnkel[0]
            LEFT_LEG_JOINT_POS[3][2] = solutionAnkel[1]
            def thighLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (LEFT_LEG_JOINT_POS[1][0] - x[0])**2 +
(LEFT_LEG_JOINT_POS[1][1] - x[2])**2 + (LEFT_LEG_JOINT_POS[1][2] - x[1])**2
- rb.LEFT_LEG_LINK_LEN[1]**2
```

```
                  return value
            def kneeLink(x):
                  global LEFT_LEG_JOINT_POS
                  value = (x[0] - LEFT_LEG_JOINT_POS[3][0]) + (x[2] -
LEFT_LEG_JOINT_POS[3][1])**2 + (x[1] - LEFT_LEG_JOINT_POS[3][2])**2 -
rb.LEFT_LEG_LINK_LEN[2]**2
                  return value
            def solveKneePitch(x):
                  return [inLine(x), thighLink(x), kneeLink(x)]
            solutionKnee = [0,0,0]
            solutionKnee = scipy.optimize.fsolve(solveKneePitch,
[LEFT_LEG_JOINT_POS[1][0], LEFT_LEG_JOINT_POS[1][2],
LEFT_LEG_JOINT_POS[1][1]+mn.STRIDE_LENGTH])
            # print("Solution Knee :: ", solutionKnee)
            LEFT_LEG_JOINT_POS[2][0] = solutionKnee[0]
            LEFT_LEG_JOINT_POS[2][1] = solutionKnee[2]
            LEFT_LEG_JOINT_POS[2][2] = solutionKnee[1]
def calJointAngles():
      global RIGHT_LEG_JOINT_POS, LEFT_LEG_JOINT_POS
      # ALL the Roll Joint Angles
      rightRoll =
math.atan(float(RIGHT_LEG_JOINT_POS[0][0]-RIGHT_LEG_JOINT_POS[1][0])/(RIGHT_
LEG_JOINT_POS[0][2]-RIGHT_LEG_JOINT_POS[1][2]))
      leftRoll =
math.atan(float(LEFT_LEG_JOINT_POS[0][0]-LEFT_LEG_JOINT_POS[1][0])/(LEFT_LEG
_JOINT_POS[0][2]-LEFT_LEG_JOINT_POS[1][2]))
      avgRoll = float(rightRoll + leftRoll)/2
      RIGHT_LEG_JOINT_ANGLES[0] = LEFT_LEG_JOINT_ANGLES[0] = avgRoll
      RIGHT_LEG_JOINT_ANGLES[4] = LEFT_LEG_JOINT_ANGLES[4] = -avgRoll
      # HIP Joint angles
      rightHipPitch = math.atan(float(RIGHT_LEG_JOINT_POS[2][1] -
RIGHT_LEG_JOINT_POS[1][1])/(RIGHT_LEG_JOINT_POS[1][2] -
RIGHT_LEG_JOINT_POS[2][2]))
      leftHipPitch = math.atan(float(LEFT_LEG_JOINT_POS[2][1] -
LEFT_LEG_JOINT_POS[1][1])/(LEFT_LEG_JOINT_POS[1][2] -
LEFT_LEG_JOINT_POS[2][2]))
      RIGHT_LEG_JOINT_ANGLES[1] = abs(rightHipPitch)
      LEFT_LEG_JOINT_ANGLES[1] = abs(leftHipPitch)
      # ANKEL Pitch angles
      rightAnkelPitch = math.atan(float(RIGHT_LEG_JOINT_POS[2][1] -
RIGHT_LEG_JOINT_POS[3][1])/(RIGHT_LEG_JOINT_POS[2][2] -
RIGHT_LEG_JOINT_POS[3][2]))
      print("_____",float(RIGHT_LEG_JOINT_POS[2][1] -
RIGHT_LEG_JOINT_POS[3][1])/(RIGHT_LEG_JOINT_POS[2][2] -
RIGHT_LEG_JOINT_POS[3][2]))
      leftAnkelPitch = math.atan(float(LEFT_LEG_JOINT_POS[2][1] -
LEFT_LEG_JOINT_POS[3][1])/(LEFT_LEG_JOINT_POS[2][2] -
LEFT_LEG_JOINT_POS[3][2]))
```

```
	RIGHT_LEG_JOINT_ANGLES[3] = abs(rightAnkelPitch)
	LEFT_LEG_JOINT_ANGLES[3] = abs(leftAnkelPitch)
	# KNEE Pitch angles
	rightKneePitch = abs(rightHipPitch) + abs(rightAnkelPitch)
	leftKneePitch = abs(leftHipPitch) + abs(leftAnkelPitch)
	RIGHT_LEG_JOINT_ANGLES[2] = rightKneePitch
	LEFT_LEG_JOINT_ANGLES[2] = leftKneePitch
def bipedNewPosition(plot):
	global RIGHT_LEG_JOINT_POS, LEFT_LEG_JOINT_POS, CURRENT_TIME,
pointHz,pointHy,pointA1z,pointA1y,pointA2z,pointA2y

	if(CURRENT_TIME < float(mn.TIME_PERIOD)/2):
		# Right Leg Support Phase, Left Leg Swing Phase
		print("RIGHT Support, LEFT Swing")
		calSupport('right', RIGHT_LEG_JOINT_POS)
		calSwing('left', LEFT_LEG_JOINT_POS)
		calAllPos()
		if(plot):
			ax.plot([x[0] for x in RIGHT_LEG_JOINT_POS], [x[1] for x
in RIGHT_LEG_JOINT_POS], [x[2] for x in RIGHT_LEG_JOINT_POS], 'b')

			ax.plot([x[0] for x in LEFT_LEG_JOINT_POS], [x[1] for x
in LEFT_LEG_JOINT_POS], [x[2] for x in LEFT_LEG_JOINT_POS], 'y')
	else:
		#Left leg Support Phase, Right Leg Swing Phase
		print("LEFT Support, RIGHT Swing")
		calSupport('left' ,LEFT_LEG_JOINT_POS)
		calSwing('right', RIGHT_LEG_JOINT_POS)
		calAllPos()
		if(plot):
			ax.plot([x[0] for x in RIGHT_LEG_JOINT_POS], [x[1] for x
in RIGHT_LEG_JOINT_POS], [x[2] for x in RIGHT_LEG_JOINT_POS], 'r')

			ax.plot([x[0] for x in LEFT_LEG_JOINT_POS], [x[1] for x
in LEFT_LEG_JOINT_POS], [x[2] for x in LEFT_LEG_JOINT_POS], 'g')
	if(plot):
		plt.pause(0.05)
	calJointAngles()
def run(currentTime, plot):
	global CONSTANT_C1, CONSTANT_C2, CURRENT_TIME, RIGHT_LEG_JOINT_POS,
LEFT_LEG_JOINT_POS
	CURRENT_TIME = currentTime % mn.TIME_PERIOD
	if checkParametersSet():
		CONSTANT_C1 = mn.STRIDE_LENGTH *
sech(math.sqrt(rb.HALF_DIST_BW_LEGS) * mn.TIME_PERIOD / 2) / 4
		CONSTANT_C2 = - CONSTANT_C1
		print("TIME t :: ", CURRENT_TIME , mn.TIME_PERIOD)
		bipedNewPosition(plot)
```

```
        else:
                print("Parameters of the Robot or the Motion Parameters are not
SET!")
def walkForward(currentTime, plot=True):
        run(currentTime, plot)
def walkBackward(currentTime, plot=True):
        currentTime = mn.TIME_PERIOD - currentTime
        run(currentTime, plot)
def rightLegPosition(hipRoll,ankelRoll):
        global RIGHT_LEG_JOINT_POS
        RIGHT_LEG_JOINT_POS[0] = hipRoll
        RIGHT_LEG_JOINT_POS[4] = ankelRoll
def leftLegPosition(hipRoll, ankelRoll):
        global LEFT_LEG_JOINT_POS
        LEFT_LEG_JOINT_POS[0] = hipRoll
        LEFT_LEG_JOINT_POS[4] = ankelRoll
def updateAllPositions(rightLegPos, leftLegPos):
        rightLegPosition(rightLegPos[0], rightLegPos[1])
        leftLegPosition(leftLegPos[0], leftLegPos[1])
        calAllPos()
        calJointAngles()
def standardPosition(front='right'):
        global CURRENT_TIME
        if (front == 'right'):
                positions = [[[
[rb.HALF_DIST_BW_LEGS, 0, mn.CENTROID_HEIGHT],[rb.HALF_DIST_BW_LEGS, 0,
0]],[[-rb.HALF_DIST_BW_LEGS, 0, mn.CENTROID_HEIGHT],
[-rb.HALF_DIST_BW_LEGS, 0, 0]]],[[
[rb.DISTANCE_BW_LEGS, 0, mn.CENTROID_HEIGHT],
[rb.HALF_DIST_BW_LEGS, 0, 0]],[[0, 0, mn.CENTROID_HEIGHT],
[-rb.HALF_DIST_BW_LEGS, 0, mn.LIFT_HEIGHT]]]]
                CURRENT_TIME = 0
                calSupport('right', RIGHT_LEG_JOINT_POS)
                calSwing('left', LEFT_LEG_JOINT_POS)

positions.append(copy.deepcopy([[RIGHT_LEG_JOINT_POS[0],RIGHT_LEG_JOINT_POS[
4]], [LEFT_LEG_JOINT_POS[0],LEFT_LEG_JOINT_POS[4]]]))
        else:
                positions = [[[[rb.HALF_DIST_BW_LEGS, 0, mn.CENTROID_HEIGHT],
[rb.HALF_DIST_BW_LEGS, 0, 0]],[[-rb.HALF_DIST_BW_LEGS, 0,
mn.CENTROID_HEIGHT],[-rb.HALF_DIST_BW_LEGS, 0, 0]]], [[[0, 0,
mn.CENTROID_HEIGHT],[rb.HALF_DIST_BW_LEGS, 0,
mn.LIFT_HEIGHT]],[[-rb.DISTANCE_BW_LEGS, 0,
mn.CENTROID_HEIGHT],[-rb.HALF_DIST_BW_LEGS, 0, 0]]]]
                CURRENT_TIME = float(mn.TIME_PERIOD)/2
                calSupport('left', LEFT_LEG_JOINT_POS)
                calSwing('right', RIGHT_LEG_JOINT_POS)
positions.append(copy.deepcopy([[RIGHT_LEG_JOINT_POS[0],RIGHT_LEG_JOINT_POS[
```

```
4]], [LEFT_LEG_JOINT_POS[0],LEFT_LEG_JOINT_POS[4]]]))
	# print(positions)
	rightJointAng = []
	leftJointAng = []
	for x in positions:
		updateAllPositions(x[0], x[1])
		# print(x[0], x[1])
		rightJointAng.append(copy.deepcopy(RIGHT_LEG_JOINT_ANGLES))
		leftJointAng.append(copy.deepcopy(LEFT_LEG_JOINT_ANGLES))
	return rightJointAng, leftJointAng

if __name__ == '__main__':
	#setAllParameters(90, [85,80,65,50,15], 3, 230, 30, 50)
	setAllParameters(65, [43,64,64,50,19],1, 210, 20, 60)
	f1=open('new_walk_anlges.txt','w')
	while CURRENT_TIME < mn.TIME_PERIOD:
		walkForward(CURRENT_TIME)
		left_angles_print=[math.degrees(x) for x in
LEFT_LEG_JOINT_ANGLES]
		right_angles_print=[math.degrees(x) for x in
RIGHT_LEG_JOINT_ANGLES]
		for abc in left_angles_print:
			f1.write(str(int(abc))+" ")
		for abc in right_angles_print:
			f1.write(str(int(abc))+" ")
		print("LEFT :", left_angles_print)
		print("Right :", right_angles_print)
		f1.write("\n")
		CURRENT_TIME += 0.05
		time.sleep(2)

	while True:
		plt.pause(5)

	f1.close()
```

8.2 Code to generate Servo mapped angels using 3-D LIP angles

```
# Author : Prajwal R M,  Pavitra N
# Receievs walking sequence angles from 3D LIP Algorithm
# Converts it to SERVO mapped angles which can be sent to the bot.

def calculate_PWM(angle):
    pwm= 500.0 + (((2500.0-500.0)/(180-0))*(angle-0))
    return pwm

f=open('new_walk_anlges.txt');
```

```
f2=open('pwm.sc3','w');
angles=f.readlines();
newangles=[0]*10;
finalangles=[]
f2.write("Max: 2500 2500 2500 2500 2500 2500 2500 2500 2500 2500 2500 2500
2500 2500 2500 2500 2500 2500\nMin: 500 500 500 500 500 500 500 500 500 500
500 500 500 500 500 500 500 500\nCen: 1500 1500 1500 1500 1500 1500 1500
1500 1500 1500 1500 1500 1500 1500 1500 1500 1500 1500\nHom: 1500 1500 1500
1500 1500 1500 1500 1500 1500 1500 1500 1500 1500 1500 1500 1500 1500
1500\nDir: True True True True True True True True True True True True True
True True True True True\nLab:Servo 1:Servo 2:Servo 3:Servo 4:Servo 5:Servo
6:Servo 7:Servo 8:Servo 9:Servo 10:Servo 11:Servo 12:Servo 13:Servo 14:Servo
15:Servo 16:Servo 17:Servo 18:\nPrm: 0 2 1 0 1\nGrp: 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0\n")

print(len(angles))
for step in angles:
    current_step=step.split();
    newangles[0]=90-(int(current_step[0])*1.4) #theta1
    newangles[1]=90+int(current_step[1])-7  #theta2
    newangles[2]=180-(30+int(current_step[2])) #theta3
    newangles[3]=90+int(current_step[3]) #theta4
    newangles[4]=90+(1.4*int(current_step[4])) #theta5
    newangles[5]=90-(1.4*int(current_step[5]))#theta6
    newangles[6]=180-(90+int(current_step[6])-7) #theta7
    newangles[7]=30+int(current_step[7]) #theta8
    newangles[8]=180-(90+int(current_step[8])) #theta9
    newangles[9]=90+(int(current_step[9])*1.4) #theta10

#print(current_step[0],current_step[1],current_step[2],current_step[3],curre
nt_step[4],current_step[5],current_step[6],current_step[7],current_step[8],c
urrent_step[9])
    print(newangles)
    current_pwm=[]
    f2.write("SERVO: ")
    for i in newangles:
        pwm=calculate_PWM(i);
        current_pwm.append(int(pwm))
        f2.write(str(int(pwm))+" ")
    current_pwm.append(1500);
    current_pwm.append(1500);
    current_pwm.append(1500);
    current_pwm.append(1500);
    current_pwm.append(1500);
    current_pwm.append(1500);
    current_pwm.append(1500);
    f2.write("1500 ")
    f2.write("1500 ")
```

```
    f2.write("500 ")
    f2.write("2500 ")
    f2.write("1500 ")
    f2.write("1500 ")
    f2.write("1500 ")
    f2.write("1500\n")
    finalangles.append(current_pwm);
f2.close()
f.close()
for j in finalangles:
    print(j)
```

8.2 Code to Plot FSR stability plots through bluetooth

```
# Author : Prasanna Venkatesan K S
# Receives data from Arduino Nano and plots FSR values.

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
import numpy as np
from mpl_toolkits.mplot3d.art3d import Poly3DCollection
import serial
import time

try:
       ser= serial.Serial('COM12')
       ser.readline()
       ser.readline()
       ser.readline()
       ser.readline()
except:
       print("Could not Connect to HC-05 module")
       exit()

R_foot_locations=[[1,5.2,5.5,5.5,1],[1,1,2,11.7,11.7],[0,0,0,0,0]]
L_foot_locations=[[-1,-5.2,-5.5,-5.5,-1],[1,1,2,11.7,11.7],[0,0,0,0,0]]

R_FSR_Locations=[[2.5, 4.5, 2.5, 4.5],[3.5, 3.5, 7.5, 7.5],[0,0,0,0]]
L_FSR_Locations=[[-2.5, -4.5, -2.5, -4.5],[3.5, 3.5, 7.5, 7.5],[0,0,0,0]]

fig=plt.figure()
ax=fig.gca(projection='3d')


def initialize():
       global verts_R,verts_L,R_center,L_center
       fig=plt.figure()
       ax=plt.gca(projection='3d')
```

```
verts_R=[list(zip(R_foot_locations[0],R_foot_locations[1],R_foot_locations[2
]))]

verts_L=[list(zip(L_foot_locations[0],L_foot_locations[1],L_foot_locations[2
]))]


      R_center=[sum(R_foot_locations[0])/5,sum(R_foot_locations[1])/5]
      L_center=[sum(L_foot_locations[0])/5,sum(L_foot_locations[1])/5]


def convertADvalue(values):
      forces =[]
      for i in values:
            Voltage = 5/1024 * int(i.decode())
            if (not(Voltage >= 4.9)):
                  Resistance = 10000 * Voltage / (5 - Voltage)
            else:
                  Resistance = 10000000
            fsrConductance = 1000/Resistance

            if(fsrConductance <=1000):
                  force = fsrConductance/80
            else:
                  force = (fsrConductance -1000)/30
            forces.append(force*1000)
      return forces

def receive_data():
      global R_forces,L_forces
      line = ser.readline()
      values = line.split()
      forces = convertADvalue(values)

      R_forces = forces[0:4]
      L_forces = forces[4:8]
      print (R_forces,L_forces)


def plot_all():

      plt.cla()

ax.add_collection3d(Poly3DCollection(verts_R,facecolors='w',linewidth=1,edge
color='k'))

ax.add_collection3d(Poly3DCollection(verts_L,facecolors='w',linewidth=1,edge
```

```
color='k'))

        ax.scatter(R_center[0],R_center[1], color='green',marker='+')
        ax.scatter(L_center[0],L_center[1], color='green',marker='+')


ax.quiver(R_FSR_Locations[0],R_FSR_Locations[1],R_FSR_Locations[2],[0,0,0,0]
,[0,0,0,0],R_forces, [2,2,2,2],normalize=False,color='blue')

ax.quiver(L_FSR_Locations[0],L_FSR_Locations[1],L_FSR_Locations[2],[0,0,0,0]
,[0,0,0,0],L_forces, [2,2,2,2],normalize=False,color='blue')

        plt.draw()
        plt.pause(0.05)

if __name__ == '__main__':
        initialize()
        while(1):
                receive_data()
                plot_all()
        ser.close()
```