technische universität
dortmund

Final Report

PG WS 2022/23 - Upper body control for the soccer playing robot NAO

Maryam Asghar, Mirzaagha Guliyev, Kumar Harsh,
Lokeshwaran Manohar, Dhiral Anilbhai Panchal,
Rahul Panchal, Mirsadhyder Shah

Reviewer:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Supervisors:
M. Sc. Aaron Larisch
M. Sc. Arne Moos

Technische Universität Dortmund
Fakultät für Elektrotechnik und Informationstechnik
Institut für Roboterforschung
https://ds.etit.tu-dortmund.de/

Contents

1 Introduction **1**

1.1 Objective . . . 2

1.2 Overview . . . 2

2 Technical Background **3**

2.1 Model Predictive Control . . . 3

2.2 Reinforcement Learning . . . 4

3 Model Predictive Control **7**

3.1 Methodology . . . 7

3.1.1 Model . . . 7

3.2 Evaluation . . . 9

3.2.1 Testing Parameters . . . 9

3.2.2 Ideal Case . . . 12

3.2.3 Error Dynamics . . . 13

3.3 Testing on Actual Robot . . . 21

3.4 Use of libmpc++ for System Optimization . . . 22

3.5 Conclusion . . . 23

4 Reinforcement Learning Walking **24**

4.1 Methodology . . . 24

4.1.1 Selection of Environment . . . 24

4.1.2 Selection of RL Algorithm . . . 26

4.1.3 Joint Angles extraction using walk request . . . 27

4.1.4 Selection of Joint Angles . . . 28

4.1.5 Selection of Observation Parameters . . . 29

4.1.6 Selection of Reward function . . . 30

4.1.7 Imitation Learning . . . 31

4.1.8 Inverse Reinforcement Learning . . . 32

4.1.9 Layers . . . 33

4.2 Hyper Parameter Tuning . . . 34

4.2.1 Neural Network Parameterizations . . . 34

4.2.2 Agent Options 34
4.2.3 Actor and Critic Optimizer Options 35
4.2.4 Exploration Model 36
4.2.5 Target Policy Smoothing Model 36
4.3 Experimental Results 36
4.3.1 Reinforcement Learning 37
4.3.2 Using External Action 42
4.4 Conclusion 44

5 Discussion and Outlook **45**

List of Figures **48**

Bibliography **49**

Chapter 1

Introduction

Robots are becoming increasingly common in modern society. They have been used for various purposes, including manufacturing, construction, and transportation. However, robots can also be used more interactively. Some current applications of robots are self-driving cars, robotic assistants in operating rooms, and service robots in hotels and hospitals. The use of robots in the home has been prevalent in recent years. Several companies are developing robotic assistants for the home that will be capable of performing a variety of tasks. One of the most interesting uses for a robot is soccer.

Robot soccer is a growing sport that involves teams of humanoid robots. These robots are equipped with various sensors and actuators that enable them to perform different tasks, such as controlling the ball, making passes, and shooting at goals. Researchers are currently working to improve the performance of robots in this sport and create advanced robots that behave increasingly similarly to humans by participating in a RoboCup Standard Platform League (SPL). The SPL is a RoboCup league in which all teams compete with identical robots that operate fully autonomously. Each team consists of 5 robots, including 1 goalkeeper and 4 field players, who can communicate with each other and send up to 1200 messages per match. Matches are 30 minutes in total, divided into two 10-minute halves and a 10-minute break. The robots are prohibited from any manual interaction and damaging the field.

The focus of the project group is the upper-body control of the robot while walking. The group is split into two groups to try out with two different approaches. One group implemented Model Predictive Control (MPC) to achieve the objective while the other group experimented with Reinforcement Learning (RL) techniques. MPC uses predictive model to estimate the future states of the robot based on current input and state information. The controller then uses this information to control the robot by providing an optimal control input. The RL group goal is to develop a new walking technique for the NAO robot that prioritizes speed and stability. Currently, the robot has a stable walking technique, but we want to improve it. To achieve this, we plan to use Reinforcement Learning (RL) methods. By using RL, the robot will be trained through trial and error, allowing it to

learn a new walking pattern. The advantage of RL is that it can learn from experience, and with the right algorithm, it can continually improve over time. Walking is an uncertain process, but RL can handle uncertainty and adjust the walking pattern accordingly, leading to more efficient walking behavior. Overall, RL will be instrumental in developing a new walking technique for the NAO robot that prioritizes speed and stability[2].

1.1 Objective

The objective of the project group is to develop a control system for the upper-body of the soccer playing robot Nao that enables it to perform precise and coordinated movements required for effective gameplay, like walking, while maintaining stability and balance. The control system should optimize the robot's agility, speed, and accuracy, while minimizing errors and the risk of damage.

1.2 Overview

The following report consists of four chapters that detail the development of a control system for the upper body of a soccer-playing robot Nao, using Model Predictive Control (MPC) and Reinforcement Learning (RL) techniques. Chapter 2, Technical Background, provides an overview of the necessary technical concepts and principles for understanding control system development. Chapter 3, Model Predictive Control, explains the basics of MPC, its advantages, and its implementation in the control system development. Chapter 4, Reinforcement Learning Walking covers algorithms,selection of observation and concepts related to hyperparameter tuning, evolution of reward function , and inverse reinforcement learning. Finally, Chapter 5, Discussion and Outlook, evaluates the developed control system's performance, discusses its limitations, proposes future improvements, and highlights areas for further research. This chapter also provides a summary of the project's results and their relevance to the project's objective

Chapter 2

Technical Background

This section provides a background on the concepts used to achieve the objective. The first sub-section explains MPC and the next sub-section RL basics.

2.1 Model Predictive Control

The current state of NaoDevils framework uses a Flexible Linear Inverted Pendulum (FLIP) Model [23]. The model posits that the centre of mass (CoM) of the robot can be modelled as an inverted pendulum that is connected to a flexible link that represents the leg. The FLIP model accounts for the compliance and flexibility of the leg. This allows for more realistic modelling of the robot's dynamics. The CoM trajectory is generated by a Linear Quadratic Regulator (LQR). The LQR approach entails modelling the robot's dynamics as a collection of linear equations, followed by the use of an optimal control algorithm to create a feedback controller that minimises a quadratic cost function based on the robot's state variables. For a biped robot, the LQR controller seeks to control the CoM's motion to maintain stability while walking. The position, velocity, and acceleration of the CoM are some examples of the current system state that the LQR controller uses to determine the best control input for the robot's actuators. Further, a PID controller is used for error regulation.

In this project, the task of upper-body control of the robot is formulated as a 3D linear inverted pendulum (3D LIP) model [10], [12], [14] and solved using Model Predictive Control [4].

The 3D LIP model [13], [11] represents the robot as a rigid body with a single point mass at its CoM, causing it to be unstable and requiring control to maintain its balance. Three dimensions of motion are considered which are forward-backward, side-to-side, and up-and-down. Further, it assumes that the robot moves in a straight line and that the legs move in a straight line as well. 3D LIP model uses two important variables: the state vector and the control input. The state vector describes the position and velocity of the walker in each dimension, while the control input describes the forces and torques applied

to the walker to control its motion. The dynamics of this model are described by a set of second-order differential equations. These equations incorporate the effects of gravity, the control input, and the internal dynamics of the walker.

MPC is a popular control strategy that uses a mathematical model of a system to predict its future behavior and optimize its control inputs over a finite time horizon. The control problem is formulated as an optimization problem that seeks to minimize a cost function that reflects the control objectives and constraints. An MPC problem can be solved either numerically or analytically, and the resulting optimal control inputs are applied to the system. The optimization problem is solved at each frame for obtaining the numerical solution. However, an analytical approach has been employed here.

Solving the MPC problem analytically has some significant advantages over its numerical counterpart. Firstly, an analytical solution can be computed offline, so only the control input must then be computed online [8]. This significantly increases the computational efficiency. Secondly, sensitivity to the initial conditions is greatly reduced because analytical solutions can provide a more robust control policy that is less sensitive to small variations in the system state.

The ZMP Preview Control system [14] suggests creating a trajectory for a humanoid robot's CoM while imposing the restriction that the footsteps are stationary and immovable. The constraint is consequently that the trajectory of the ZMP defined by equation (2.1) always stays within the convex hull of these fixed footprints. Another simplifying premise is that the CoM's altitude, h_{CoM}, remains constant. This signifies that the model is linearized around it. Further, assuming that the CoM does not move vertically, we get the position z of the ZMP on the ground:

$$z = x - \frac{h_{CoM}}{g}\ddot{x} \tag{2.1}$$

where x is the horizontal position of the CoM, $\ddot{x}$ its horizontal acceleration, h_{CoM} its altitude and g is the gravity force. In the examination of the robot's ZMP, this approximation decouples the lateral and forward movements of the robot. As a result, throughout the work of the project, only the case of lateral motions is considered, knowing that the case of forward motions is exactly identical.

2.2 Reinforcement Learning

Reinforcement learning(RL) is another approach for solving upper-body control [16]. RL is a type of machine learning that involves training an agent to make decisions based on feedback received from its environment. It is a popular approach for training robots to perform complex tasks, such as grasping objects, navigating through environments, or walking.

The goal of the control problem in RL is to learn a policy $pi(\texttt{a\_t|s\_t})$ that maps states to actions in a way that maximizes the expected cumulative reward. The RL agent learns the optimal policy by interacting with the environment, observing the current state and the reward received, and updating its policy based on this information.

There are several types of RL algorithms, each with their own strengths and weaknesses. Here are some of the most common types of RL:

1. Q-Learning: Q-Learning is a simple and effective RL algorithm that works by iteratively updating a Q-table, which stores the expected rewards for each state-action pair. Q-Learning is particularly useful for problems where the state and action spaces are small [25].

2. Policy Gradient Methods: Policy Gradient methods directly optimize the policy that maps states to actions, rather than learning a value function like Q-Learning. They are often used in problems with continuous action spaces and have shown great success in a variety of applications, including robotics and game playing[20].

3. Actor-Critic Methods: Actor-Critic methods combine elements of both value-based and policy-based methods. They have an actor that learns the policy and a critic that estimates the value function. This approach can be particularly useful when dealing with problems that have high-dimensional state and action spaces[7].

4. Model-based RL: Model-based RL algorithms learn a model of the environment and then use that model to plan actions. This approach can be particularly useful when dealing with problems that have complex dynamics, but it requires a lot of data to build an accurate model[22].

5. Multi-agent RL: Multi-agent RL involves training multiple agents to work together to achieve a common goal. This approach can be particularly useful in scenarios where multiple agents must collaborate to complete a task, such as in robotics or in multiplayer games.

In this project, we implemented three different approaches: DDPG, PPO and TD3. Their definitions are as follows:

1. **DDPG**: DDPG(Deep Deterministic Policy Gradient) is a reinforcement learning algorithm for problems with continuous action spaces and sparse rewards. It uses a neural network to represent the policy and value function, updates them iteratively using gradient descent, and employs experience replay to reduce correlation between experiences [9].

2. **PPO**: PPO(Proximal Policy Optimization) is a reinforcement learning algorithm that works for both continuous and discrete action spaces, and can handle both sparse and dense rewards. It updates the policy iteratively using a gradient method

while constraining the updates to be close to the current policy, and uses a clipped surrogate objective to limit the size of the policy update [3].

3. **TD3**: TD3 (Twin Delayed DDPG) is a deep reinforcement learning algorithm that improves upon DDPG by using twin critics to reduce variance, an exploration strategy to explore the state space effectively, and has shown to outperform other algorithms on various benchmark tasks, particularly in continuous control and robotic manipulation [6].

Throughout the project, TD3 was better than DDPG and PPO in teaching a robot to walk because of its improved stability through twin critics and better exploration with added noise to the target policy. TD3's use of twin critics helps to reduce the variance in the critic updates, which led to more stable learning. Additionally, TD3's exploration strategy helped the robot to explore the state space more effectively and learn more efficient walking patterns. All things considered, TD3 was selected as the main RL algorithm for this project.

Chapter 3

Model Predictive Control

The optimal control method that was employed to achieve the objective was Model Predictive Control. This chapter covers the methodology of the project and model used. It also explains the setting under which the experiment was conducted, the results obtained and, finally, the conclusion.

3.1 Methodology

The task of upper-body control of the robot has been formulated as a Model Predictive Control problem, which globally amounts to solving online a sequence of Optimal Control problems [26] over a limited amount of time, indicated by its prediction horizon. Additionally, a ZMP Preview Control scheme is used to generate dynamically stable motions through a 3D LIP approximation of the dynamics of the Center of Mass (CoM) of the robot 2.

3.1.1 Model

The matrices and the equations presented below can be referred from [26]. With the assumptions made in 2, the CoM and ZMP trajectories are discretized as piecewise cubic polynomials, with constant jerks $\dddot{x}$ and $\dddot{z}$ throughout time intervals of constant lengths T. In NaoDevils framework, the refresh rate is 83.34 Hz, hence the value of $T = 12ms$ With the notations shown in equation (2) of [26], the easy integration of the constant jerk $\dddot{x}$ across time intervals of length T leads to the recursive relationship as shown below, which also gives the state-space 3D LIP Model of the system:

$$\hat{x}_{k+1} = \begin{bmatrix} 1 & T & \frac{T^2}{2} \\ 0 & 1 & T \\ 0 & 0 & 1 \end{bmatrix} \hat{x}_k + \begin{bmatrix} \frac{T^3}{6} \\ \frac{T^2}{2} \\ T \end{bmatrix} \dddot{x}_k \tag{3.1}$$

whereas equation (2.1) becomes

$$\hat{z}_k = \begin{bmatrix} 1 & 0 & -\frac{h_{CoM}}{g} \end{bmatrix} \hat{x}_{k+1} \tag{3.2}$$

for $t = kT$ with $k = 1, 2,...$

The purpose of using a ZMP Control Scheme proposed in [14] is to minimize the jerk while maintaining a position z_k of the ZMP as close as possible to some prescribed reference positions z_k^{ref}.

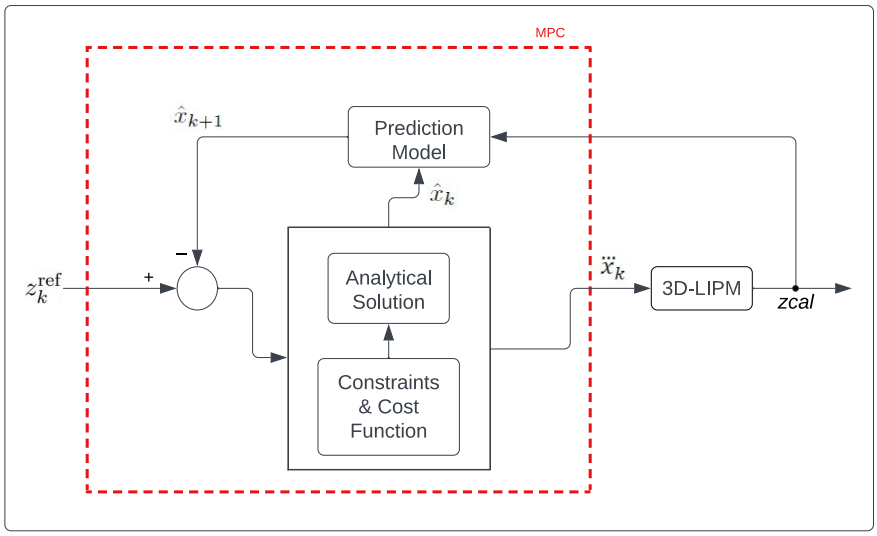


Figure 3.1: The MPC scheme diagram

The idea of using MPC here is to execute only the first interval $[kT, (k+1)T]$ of the trajectory, then measure the actual state of the system, i.e., position, velocity and acceleration of the CoM, and then recompute a new trajectory using the Quadratic Program (QP) mentioned in equation (7) of [26], allowing for some feedback of the state through ratio R/Q, which allows to balance the minimization of jerks $\dddot{x}_k$ with the tracking of reference positions z_k^{ref} in k^{th} frame .

The recursive relation in equation (3.1) can be iterated N times and combined with N versions of the relation (3.2) in order to relate at once N values of the jerk $\dddot{x}_k$ of the CoM with N values of the position z_k of the ZMP, which gives a simplified representation:

$$Z_{k+1} = P_x \hat{x}_k + P_u \dddot{X}_k \tag{3.3}$$

where P_x and P_u are as in equation (8) of [26]. Z and X represent the vectors of all calculated ZMP and the optimal inputs, respectively, throughout the horizon.

From this, the QP in equation (7) can be written as shown in equation (10) of [26]. Finally, solving this QP analytically leads to:

$$\dddot{x}_k = -e^T (P_u^T P_u + \frac{R}{Q} I_{N\times N}^{-1} P_u^T (P_x \hat{x}_k - Z_k^{ref}) \tag{3.4}$$

where $I_{N\times N}$ is an identity matrix. The jerk $\dddot{x}_k$ is the control input applied to the system.

It can be observed that solving the QP analytically reduces the entire calculation to simple matrix manipulations. If numerical approach had been chosen, then more complex algebraic Riccati equation [19] would have had to be solved.

A stability check can be performed for the system at particular values of R/Q ratio and length NT of the horizon over which the trajectories are pre-computed. If the norms of the three eigenvalues of the stability matrix shown in equation (14) of [26] turn out to be less than 1, then the control scheme can be said to be stable .

The control scheme is applied as follows: The next predicted state $\hat{x}_{k+1}$ gives the values of position, velocity and acceleration of CoM of the robot which should be achieved in the next frame. Moreover, $\hat{z}_k$ gives the calculated position of the ZMP in the current frame.

3.2 Evaluation

The evaluation section presents the experimental conditions and testing parameters used to assess the performance of the proposed model. Various scenarios and conditions were evaluated to demonstrate the effectiveness and robustness of the system. The results section presents the evaluation of the control laws under various experimental conditions, for both Software-in-the-Loop (SiL) and Hardware-in-the-Loop (HiL) testing.

3.2.1 Testing Parameters

To determine the best values of the parameters that primarily influence the stability of the control scheme, i.e., R/Q ratio and prediction horizon NT, test scenarios were run for all of the above cases, in SimRobot as well as on the actual robot. In the simulation, a force $F = [f_x\ f_y\ f_z]$ was applied in the $X-Y$ plane to the robot by using an in-built command in the simulator, whereas, the distance travelled was measured for the actual robot.

Tuning R/Q ratio

In the cost function, the R/Q ratio establishes the relative weights of control effort (R) and deviation from the setpoint (Q). When the R/Q ratio is higher, the controller will prioritise decreasing control effort, whereas when the R/Q ratio is lower, the controller will prioritise correctly tracking the setpoint. Observed effect of R/Q ratio upon the ZMP movement is as follows: With a higher value of R/Q ratio, a smoother curve was obtained. On the other hand, with a lower value of this ratio, the calculated ZMP z_k tracked the reference ZMP z_k^{ref} more accurately. This can be observed in Table 3.2.1

With the number of slices of prediction horizon, N, fixed at 100, the values mentioned in Table 3.2.1 were tested. The system was stable for the values below 10^{-2}. For values below 10^{-9}, no significant difference in the tracking was observed.

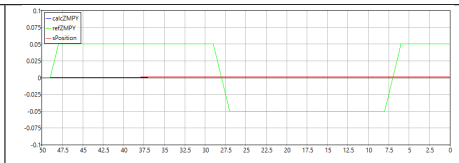
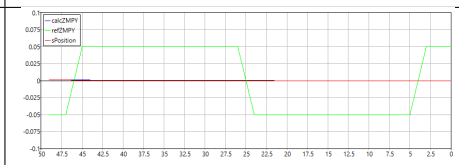
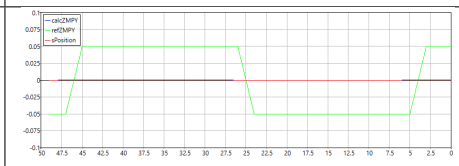
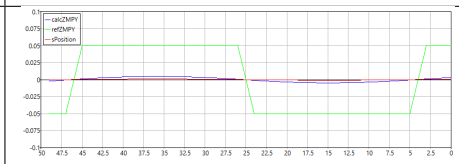
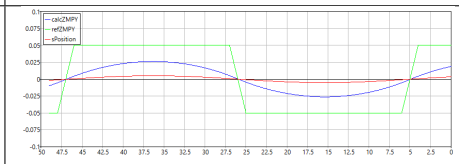
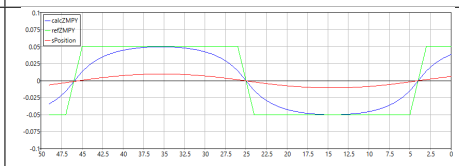
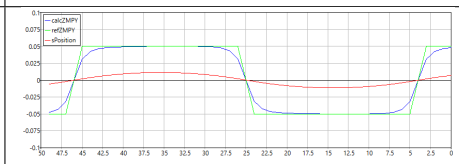
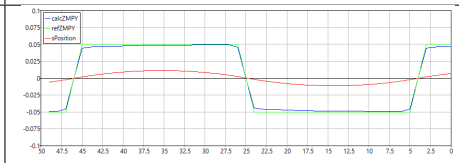
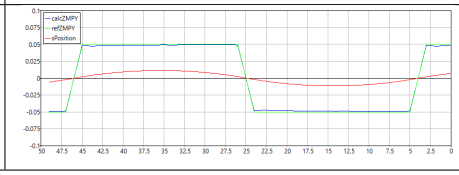
R/Q Ratio	Graph	Stability
10^{-1}		Unstable
10^{-2}		Stable
10^{-3}		Stable
10^{-4}		Stable
10^{-5}		Stable
10^{-6}		Stable
10^{-7}		Stable
10^{-8}		Stable
10^{-9}		Stable

Table 3.1: Test results for various values of R/Q ratio

When $R/Q = 10^{-9}$ and prediction horizon $NT = 100$, computation time for each frame was approximately 11 ms. However, in NaoDevils framework, each frame is 12 ms, during which all the information must be exchanged. Hence, utilizing 11 ms for just this purpose

is not possible. Therefore, as a solution, the value of N was reduced to speed up the calculations and decrease the time consumed.

Tuning Prediction Horizon NT

At $R/Q = 10^{-9}$, the stability check demonstrated in equation (14) of [26] declares the system unstable. Thus, further tests were conducted for $R/Q = 10^{-8}$, where it was observed that the system was stable for all $N > 75$. For an MPC problem, a finite prediction horizon $N_{min}T$ exists that is optimal for the closed-loop system [5]. For all values of prediction horizon greater than this minimum value, the performance of the system does not improve[4]. For values less than $N_{min}T$, the closed loop can still be close to optimality due to receding horizon implementation; this can be seen in Table 3.2 when the number of frames N was 75. It was observed that 84 is N_{min} because the tracking did not improve for greater values of N. This was concluded by calculating the error between the calculated and the reference positions of ZMP.

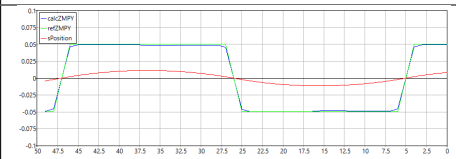
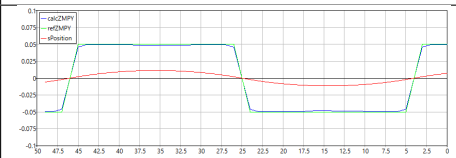
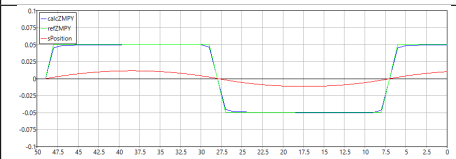
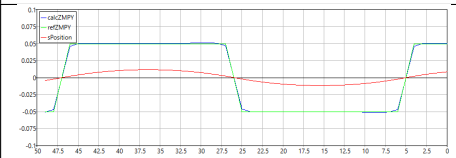
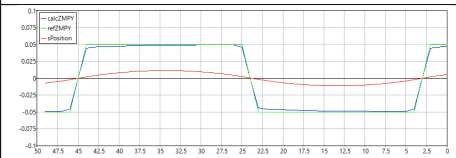
Prediction horizon	Graph	Stability
50	-	Unstable
75		Stable
77		Stable
84		Stable
90		Stable
100		Stable

Table 3.2: Test results for various values of prediction horizon

The problem in (3.1) and (3.2) was simulated in the simulation environment, SimRobot, within the NaoDevils Framework and later, on the robot itself, with the following scenarios for $N = 84$ and $R/Q = 10^{-8}$:

3.2.2 Ideal Case

It is assumed that the system runs without the existence of any error, as in the case of an ideal world. There is no error feedback in this case.

$$\hat{x}_{k+1} = A\hat{x}_k + B\,\ddot{x}_k \tag{3.5}$$

$$\hat{z}_k = C\hat{x}_{k+1} \tag{3.6}$$

In figure 3.2, it can be observed that the calculated position of ZMP tracks the reference position of ZMP with high accuracy, and the movement of CoM also seems quite stable.

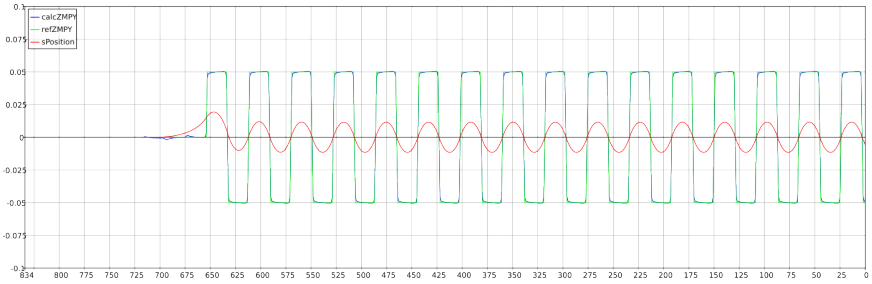


Figure 3.2: Calculated ZMP position, reference ZMP position and actual CoM position.

Figure 3.3 shows the control input. It noteworthy that the control input remains constant throughout the 10 seconds because the system assumes ideal world, without any disturbances, and hence, without the need to compensate them.

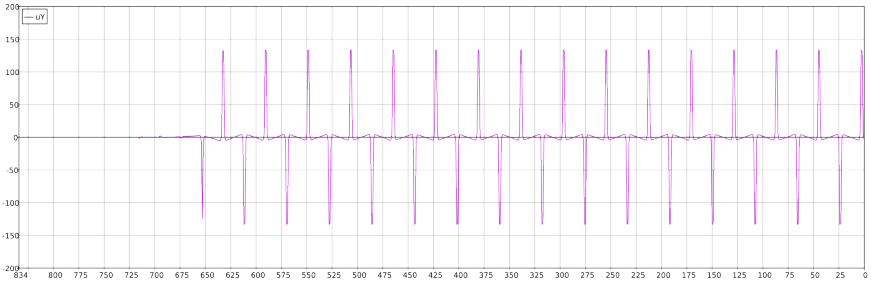


Figure 3.3: Applied control input.

Figure 3.4 indicates the error between estimated and measured positions of CoM. It can be observed that the error remains more or less constant. This is, again, due to the fact that an ideal scenario is assumed.

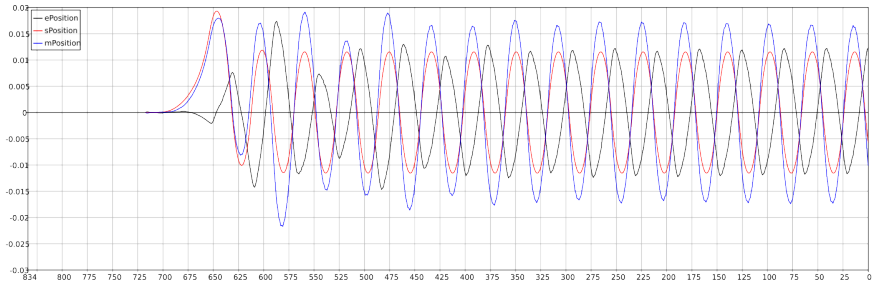


Figure 3.4: Error in estimated and measured positions of CoM.

3.2.3 Error Dynamics

The analysis and design of control systems that can successfully minimise or eliminate the error between a system's expected output and actual output are done using error dynamics. Error dynamics have been implemented in five different scenarios. The primary difference between these scenarios can be observed where the error has been fed within the control loop. The error E can be defined as:

$$E = \hat{x}_k - x_{mk} \tag{3.7}$$

where x_{mk} and $\hat{x}_k$ are the matrices of measured and estimated state, respectively.

Frame Lag Compensation

The time consumed for requesting and receiving the measured values from the sensors introduces an inherent lag of three frames. In other words, by the time the measured values at frame k are received, the robot is already at frame $(k + 3)$. This discrepancy causes false error calculations and hence, incorrect values being fed into the controller. This issue had overcome by using a Ring Buffer to temporarily store the state values from the previous three frames until they were no longer required. Hence, the obtained values for error are not from the same frame but from the antepenultimate frame.

Case 1. Error enters the state-space model linearly

The error is introduced into the system linearly, with K as its gain and added to state equation directly.

$$\hat{x}_{k+1} = A\hat{x}_k + B\,\ddot{x}_k - K \cdot E \tag{3.8}$$

In this case, different gains for each of the three states were selected and multiplied element-wise:

$$K = \begin{bmatrix} k_p \\ k_v \\ k_a \end{bmatrix}; E = \begin{bmatrix} E_p \\ E_v \\ E_a \end{bmatrix}; K \cdot E = \begin{bmatrix} k_p * E_p \\ k_v * E_v \\ k_a * E_a \end{bmatrix} \tag{3.9}$$

Figure 3.5 shows the calculated position of ZMP and the reference position of ZMP, when error dynamics are applied as shown in equation (3.8). After trying multiple values, comparatively better results were obtained for the following values of gain: $K = [0.015\ 0.0015\ 0.0015]^T$

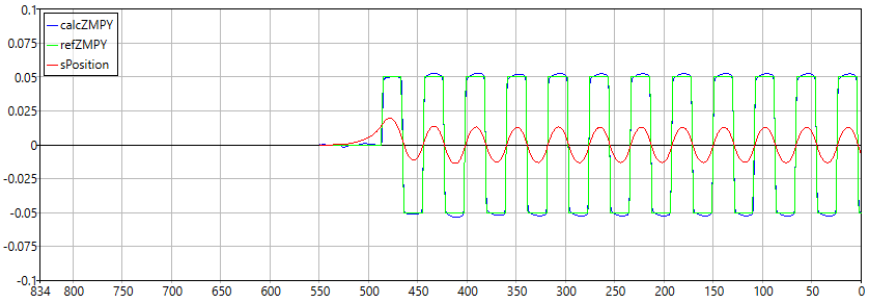


Figure 3.5: Calculated ZMP position, reference ZMP position and actual CoM position for Case 1 of error dynamics 3.2.3.

Marginal error compensation can be seen in figure 3.6 at around 400 ms.

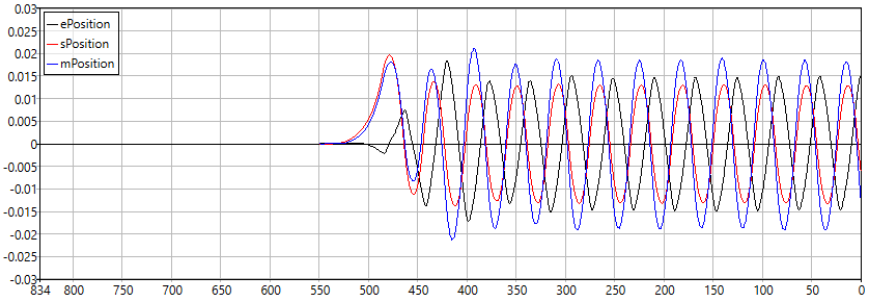


Figure 3.6: Error in estimated and measured positions of CoM.

An external force $F = [25\ 75\ 0]$ was applied to the robot, which can be observed at around 500 ms in figures 3.5 and 3.8 showing the movement of ZMP and positional error of CoM, respectively.

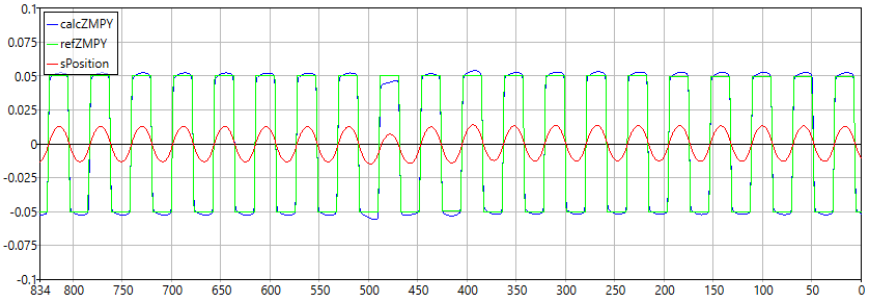


Figure 3.7: Calculated ZMP position, reference ZMP position and actual CoM position for Case 1 of error dynamics 3.2.3 when a force is applied.

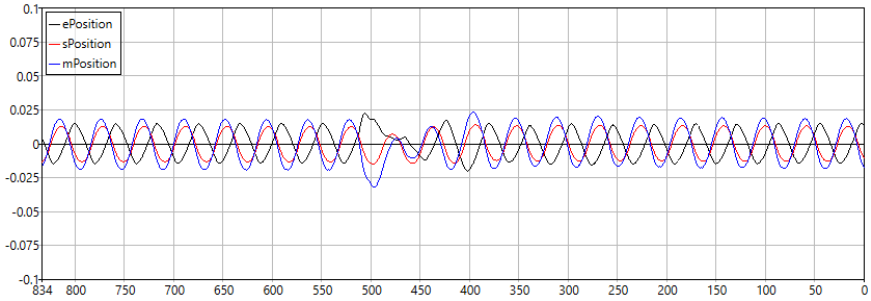


Figure 3.8: Error in estimated and measured positions of CoM when a force is applied.

Case 2. Error incorporated into the input matrix linearly

Here, the gain matrix K is multiplied with the error matrix E by matrix multiplication. The resulting input u and the state space equation is as follows:

$$u = \dddot{x}_k - K^T E \tag{3.10}$$

$$\hat{x}_{k+1} = A\hat{x}_k + B(\dddot{x}_k - K^T E) \tag{3.11}$$

Overall, the jerk is calculated analytically as in equation (3.4), but the input that goes into the system model contains an error component. The output can be calculated as in equation (3.7). Figure 3.9 shows the calculated position of ZMP and the reference position of ZMP, when error dynamics are applied as shown in equation (3.11). After trying multiple values, comparatively better results were obtained for the following values of gain: $K = [100\ 15\ 0.5]^T$.

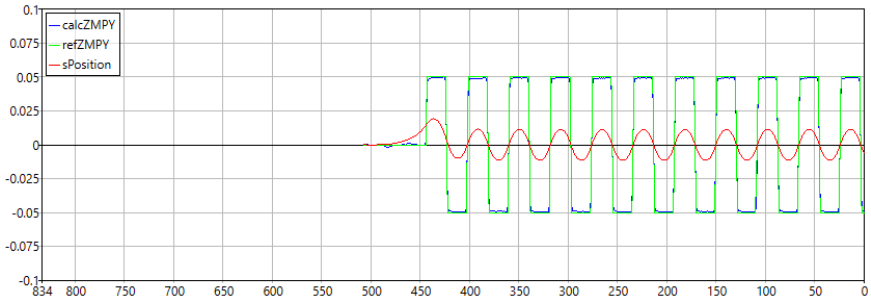


Figure 3.9: Calculated ZMP position, reference ZMP position and actual CoM position for Case 2 of error dynamics 3.2.3.

For the above-mentioned values of gain K, the tracking of calculated position of ZMP to the reference position of ZMP improved, but the improvement in error compensation (Figure 3.10) is not as evident.

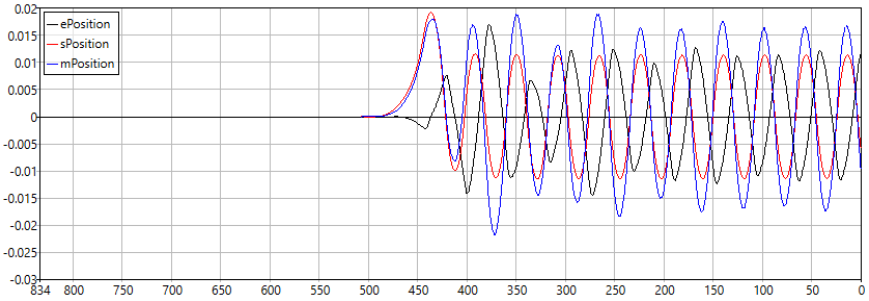


Figure 3.10: Error in estimated and measured positions of CoM.

An external force $F = [25\ 75\ 0]$ was applied at around 400 ms, whose effects on ZMP movement and positional error of CoM can be observed in figures 3.11 and 3.12, respectively.

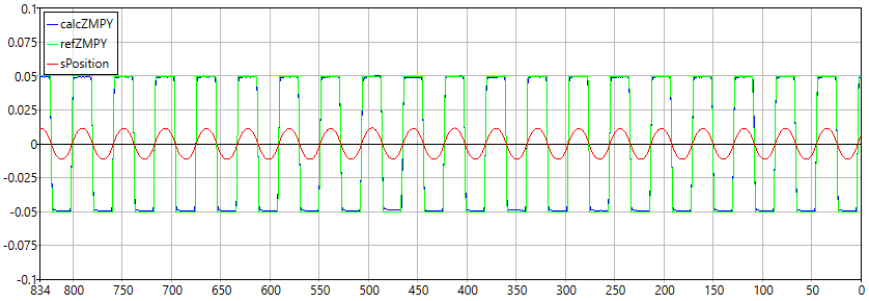


Figure 3.11: Calculated ZMP position, reference ZMP position and actual CoM position for Case 2 of error dynamics 3.2.3 when a force is applied.

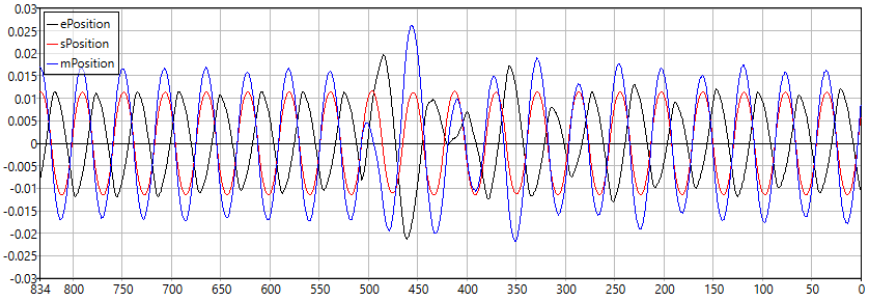


Figure 3.12: Error in estimated and measured positions of CoM when a force is applied.

Case 3. State estimation error fed into jerk calculation with gain

Here, the state estimation error is fed directly to the input jerk calculation, which gives an analytically computed input that already had an error correction term. Thus, input equation (3.4) changes as shown:

$$\dddot{x}_k = -e^T(P_u^T P_u + \frac{R}{Q} I_{N\times N})^{-1} P_u^T (P_x(\hat{x}_k - K \cdot E) - Z_k^{ref}) \tag{3.12}$$

Figure 3.13 shows the calculated position of ZMP and the reference position of ZMP, when error dynamics are applied as shown in equation (3.12). After trying multiple values, comparatively better results were obtained for the following values of gain: $K = [0.15\ 0.015\ \ -0.0015]^T$.

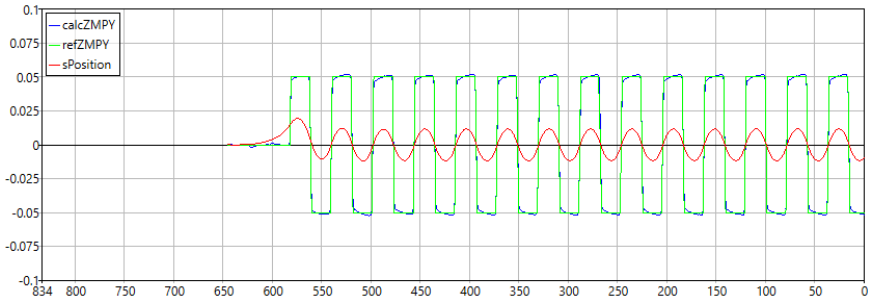


Figure 3.13: Calculated ZMP position, reference ZMP position and actual CoM position for Case 3 of error dynamics 3.2.3.

The error of estimated and measured positions of CoM can be observed in figure 3.14

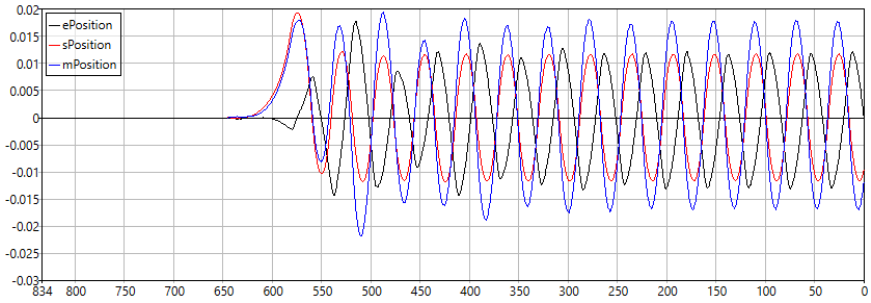


Figure 3.14: Error in estimated and measured positions of CoM.

An external force $F = [25\ 75\ 0]$ was applied at around 500 ms, whose effects on ZMP movement and positional error of CoM can be observed in figures 3.15 and 3.16, respectively. It can be observed that despite application of force, there is no overshoot in the calculated position of ZMP and reasonably good tracking was achieved.

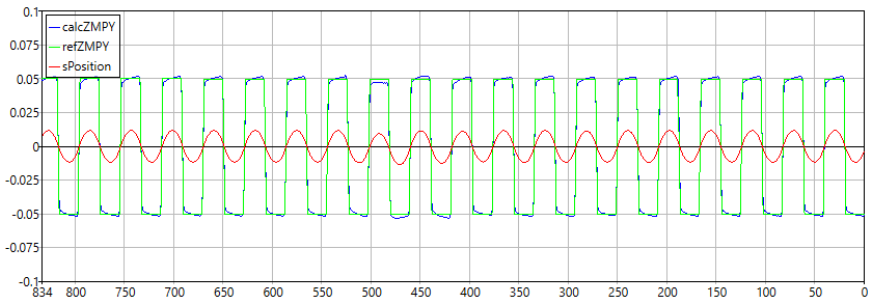


Figure 3.15: Calculated ZMP position, reference ZMP position and actual CoM position for Case 3 of error dynamics 3.2.3 when a force is applied.

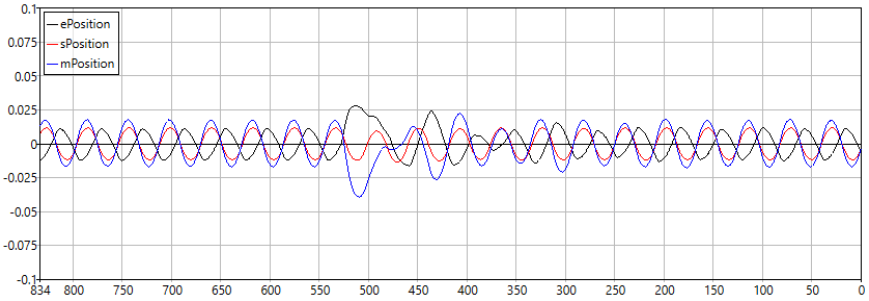


Figure 3.16: Error in estimated and measured positions of CoM when a force is applied.

Case 4. Complementary state estimation

This method depends upon the confidence we place in the obtained measurements and estimated states.

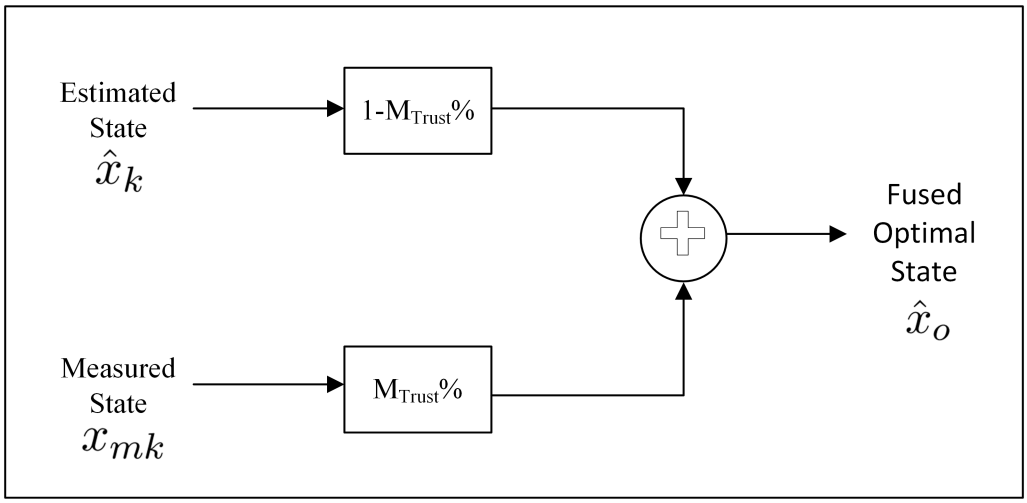


Figure 3.17: Complementary filter for fusion of states (3.13)

Therefore, the fused state can be obtained as follows:

$$\hat{x}_o = (1 - M_{Trust})\hat{x}_k + M_{Trust}x_{mk} \tag{3.13}$$

where $\hat{x}_o$ is the fused optimal state and M_{Trust} represents the trust or confidence placed in the measurements. If M_{Trust}>50%, then it indicates that more confidence is placed on the measured state as opposed to estimated state, and vice-versa. The resulting new state-space equation is shown below:

$$\hat{x}_{k+1} = A\hat{x}_o + B\,\ddot{x}_k \tag{3.14}$$

Figure 3.18 shows the calculated position of ZMP and the reference position of ZMP, when error dynamics are applied as shown in equation (3.14). After trying multiple values, it

was observed that for values of M_{Trust} between 0% and 2.5%. Moreover, comparatively better results were obtained for $M_{Trust} = 2\%$.

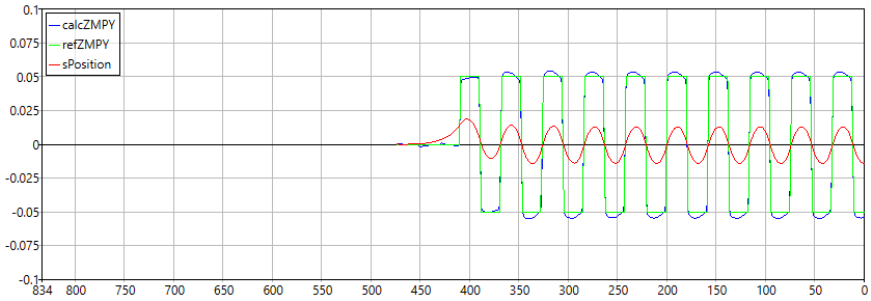


Figure 3.18: Calculated ZMP position, reference ZMP position and actual CoM position for Case 4 of error dynamics 3.2.3.

Here, ZMP tracking and error compensation (Figure 3.19) were worse than the previous scenarios.

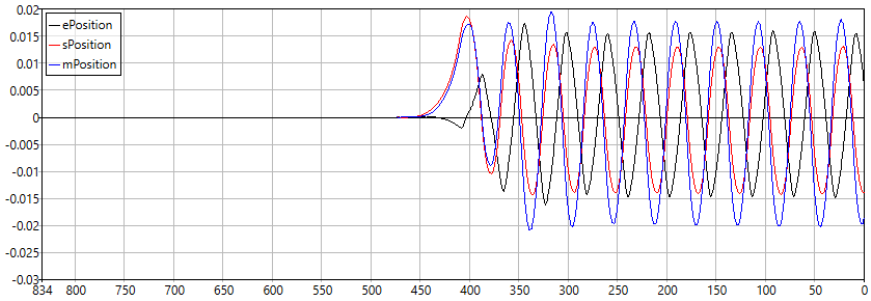


Figure 3.19: Error in estimated and measured positions of CoM.

An external force $F = [25\ 75\ 0]$ was applied at around 550 ms, whose effects on ZMP movement and positional error of CoM can be observed in figures 3.20 and 3.21, respectively. Better regulation can be observed as it only took one walk cycle to return back to normal walking pattern.

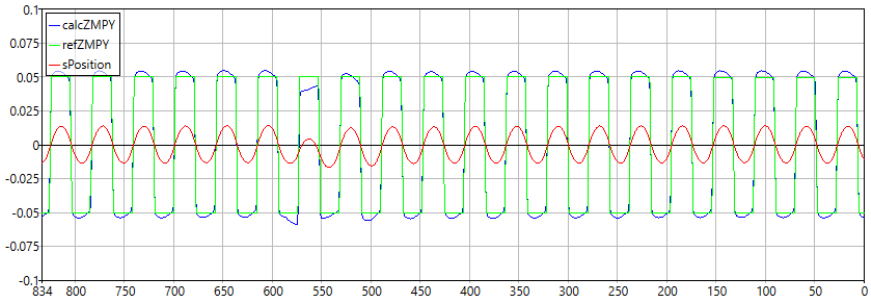


Figure 3.20: Calculated ZMP position, reference ZMP position and actual CoM position for Case 4 of error dynamics 3.2.3 when a force is applied.

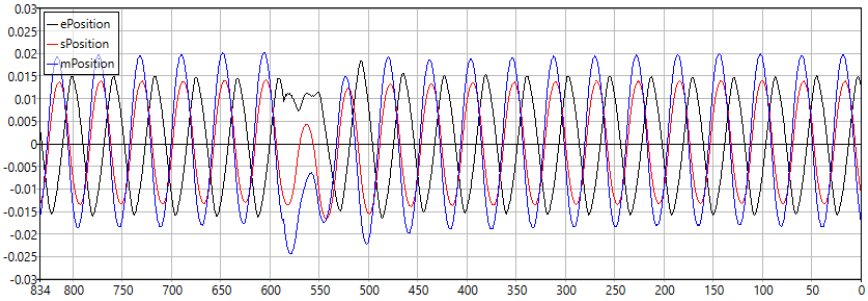


Figure 3.21: Error in estimated and measured positions of CoM when a force is applied.

Case 5. Considering x_{mk} instead of $\hat{x}_k$ in jerk calculation

Here, the measured state x_{mk} is fed directly into the input jerk calculation, which gives an analytically computed input, similar to Case 3, and the equations for control input and the state-space change as follows:

$$\dddot{x}_{mk} = -e^T(P_u^T P_u + \frac{R}{Q} I_{N\times N})^{-1} P_u^T (P_x x_{mk} - Z_k^{ref}) \tag{3.15}$$

$$x_{k+1} = Ax_{mk} + B\,\dddot{x}_{mk} \tag{3.16}$$

The plots below are obtained when error dynamics are implemented in SimRobot.

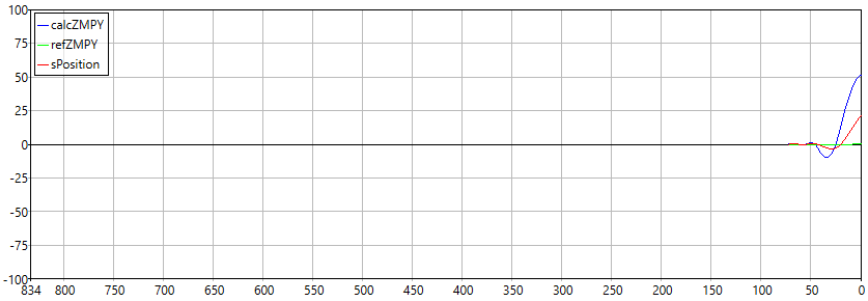


Figure 3.22: Calculated ZMP position, reference ZMP position and actual CoM position for Case 5 of error dynamics 3.2.3.

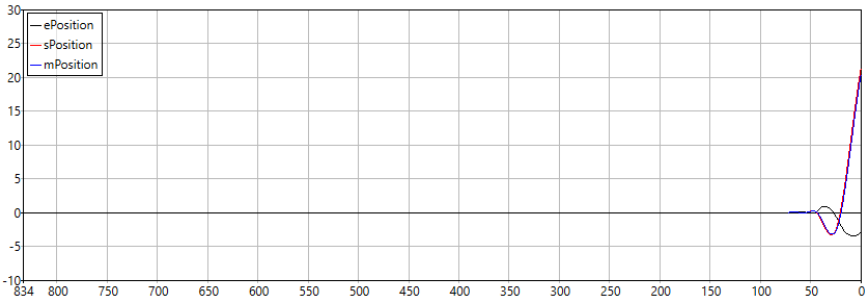


Figure 3.23: Error in estimated and measured positions of CoM.

In this case, the robot exhibited aggressive movements before falling down, hence, this input case was deemed unsuitable. Measured states are inappropriate for MPC since they indicate the system's previous state, however MPC is intended to optimize the system's future behavior. The MPC generates control actions that minimize a performance criterion using a model of the system to forecast future states. If measured states were utilized, the MPC's model of the system and its projections of future states would both be flawed. As a result, the MPC predicts the system's future behavior using a model of the system rather than measured states.

3.3 Testing on Actual Robot

This section demonstrates the results obtained when the parameters were tested on the actual robot. The following observations were made when the above-mentioned scenarios were tested. Each scenario was tested for a time period of 3 minutes. Then, the total distance travelled by the robot was measured.

Oscillation Range (deg/s)	**Notation**	**Inference**
$O < 11.5$		Negligible
$11.5 < O < 23$	*	Low
$23 < O < 34.5$	**	Medium
$34.5 < O < 57$	***	High
$O > 57$	****	Very High

Table 3.3: Oscillation Range (deg/s) and its notation

For the input cases where the robot cannot walk in a stable fashion, instability is induced in its walk in the form of oscillations. For poor values, the robot experiences oscillations about its x-axis that keep on increasing, till it falls down. The level of these oscillations and how they can be inferred have been explained in Table 3.3.

Table 3.4 indicates the values of distance (in mm) the robot walks in 3 minutes, at the mentioned velocities v. The * sign, beside the distance, is an indicator of the intensity of oscillations and the number after that shows the number of times the robot fell down during the testing period. Realistically, during an SPL match, Nao robot encounters external

Input Case	v=**10**	v=**50**	v=**100**	v=**150**
Ideal Case	1.6* | 0	7.9*** | 5	14.4**** | 14	18.2**** | 26
Input Case 1	4.9* | 0	8.1* | 4	16.7** | 9	29.1** | 6
Input Case 2	2.5** | 8	7.6*** | 15	14.9**** | 13	19.5**** | 19
Input Case 3	6.1* | 0	10.1* | 0	19.8** | 1	29.8** | 3
Input Case 4	5.4 | 0	8.9* | 0	21.1* | 0	30.5** | 2

Table 3.4: Distance travelled by the robot at indicated velocities (v) (in mm/s) and the number of times it fell down during the 3-minute testing period.

forces a number of times, which makes the Ideal Case unsuitable for use. Evidently, Input Case 3 and 4 are comparatively better amongst all the cases. However, for lower velocities of $v = 10$ and $v = 50$, the robot walks the maximum distance for Input Case 3. For higher velocities of $v = 100$ and $v = 150$, Input Case 4 proved to be the best in terms of distance covered by the robot in 3 minutes. It can be observed that the oscillation intensity and the number of falls were comparatively lower in this scenario. On the other hand, the worst behaviour was observed in Input Case 2, where the oscillation intensity was very high and the robot fell down the maximum number of times.

3.4 Use of libmpc++ for System Optimization

Libmpc++ [21] is a C++ library that provides an implementation of MPC. It is built on top of the Eigen C++ library, which provides a high-performance linear algebra library. This makes it easy to define and manipulate complex control systems in C++. The library provides a set of classes and functions for defining and solving MPC problems. It supports both linear and nonlinear models, and provides a range of optimization algorithms to find the optimal control inputs. Libmpc++ supports various optimization techniques, including quadratic programming and interior-point methods. It also provides a range of options for defining constraints on the system, including inequality and equality constraints. It has a clear and well-documented API, making it easy for developers to use and extend. The library also provides a range of debugging tools, including detailed error messages and runtime diagnostics. The library is modular and extensible, allowing developers to add their own custom optimization algorithms and constraints. This makes it easy to adapt the library to specific applications and control problems. Libmpc++ provides a range of performance optimizations, including matrix factorization and sparse matrix support. This makes it possible to solve large-scale control problems in real-time. The library includes a range of features for handling uncertainty and disturbances in the control system. These include robust control techniques and stochastic optimization algorithms. Libmpc++ is an open-source project. In summary, libmpc++ is a powerful and flexible library for solving control problems using MPC. Its performance, flexibility, and ease of use make it an suitable solution for real-time control applications.

The system modelled in (3.1) was optimized to generate optimal input sequence by using the in-built cost function of the library. The constraints for the states, input and output were mapped from the ideal case of analytical solution. Further, the error was induced into the system as process noise. The optimal input sequence calculation was based on current system state and previous input applied. This enhances the robustness of the system.

However, the implementation was not fully successful as the computational time for each frame was very high. Moreover, the output function of libmpc++ yields an exorbitantly high value for the calculated position of ZMP (Figure 3.24). Due to this, the robot

could not walk even in SimRobot. Further progress with this library could not be achieved due to time constraints. Libmpc++ is a promising tool and can be optimised further in future.

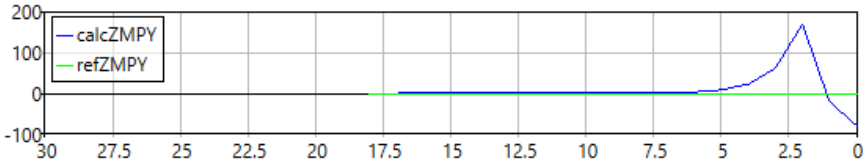


Figure 3.24: Calculated ZMP position, reference ZMP position using libmpc++.

3.5 Conclusion

As a result, Model Predictive Control shows to be a very useful tool for balancing the Nao robot. When using the ZMP Preview Control system, which minimizes the jerk of the robot's trajectory of the CoM over a finite horizon. It is possible to create stable walking movements that can be continually updated online to reflect the robot's present state. Analytical solution of MPC combined with complementary filter yields the most stable walk for the robot, among the tested scenarios. The values for which this occurs on the robot are $N = 84$ and $R/Q = 10^{-8}$, which is a reasonable trade-off between reduced computational time and better system performance. Moreover, libmpc++ shows promise and can be integrated into the framework in future.

Chapter 4

Reinforcement Learning Walking

Reinforcement learning(RL) is a type of machine learning that focuses on decision-making in dynamic environments. It involves an agent that interacts with an environment by taking actions and receiving feedback in the form of rewards or penalty. The goal of this project is to make the agent learn walking motion from scratch using Reinforcement Learning. Furthermore, Inverse Reinforcement Learning was used which utilized the pre-recorded robot walking motion to enhance the reward and improve forward velocity.

This chapter covers the process of selecting the environment and establishing the connection between Matlab(agent) and Visual Studio(Environment). The chapter also explains why we chose Matlab to implement RL and discuss the specific RL algorithm we selected for the walking technique.

4.1 Methodology

This section describes the thought process behind choosing the methodology to carry out the objective of training a NAO robot to walk using RL.

4.1.1 Selection of Environment

When selecting an environment for training a NAO robot to walk in simulation, there are several factors to consider. In this study, Matlab was used as the agent and C++ as the environment, but an alternative approach could have been using libtorch in C++.

On one hand, libtorch in C++ could have provide advantages in terms of speed, control, and flexibility, it may also have disadvantages in terms of the learning curve and hardware requirements. On the other hand, Matlab and C++ may be easier to use and integrate, but may be slower and provide less control over the robot's movements compared to libtorch in C++. Ultimately, the selection of an environment depends on the specific needs of the application and the expertise of the programmer.

Another factor to consider when selecting an environment for training a NAO robot to walk in simulation is the timeframe for the project. In this study, the project had a timeline

of six months, which may not have been sufficient to learn and effectively use libtorch in C++. Learning a new library or programming language can take time, and in this case, using Matlab and C++ was a more feasible option given the available time frame. It is important to consider the trade-offs between the potential advantages and disadvantages of each environment, as well as the available resources and timeframe for the project.

Keeping all of the above in mind we decided to choose the Nao Devil's framework in C++ as the environment and used SimRobot as the simulation environment and chose a simulation scene, where all image processing is disabled and only the robot's physics in simulated which allowed us a faster execution.

TCP/IP Communication

TCP (Transmission Control Protocol) is a reliable protocol that ensures that data is delivered to its intended recipient without errors or loss. In contrast, UDP (User Datagram Protocol) is a faster but less reliable protocol that does not guarantee delivery or error checking. When connecting Matlab to C++, using TCP instead of UDP can provide greater reliability and consistency in data transmission. With TCP, the sender can confirm that each packet of data has been successfully received by the receiver, ensuring that the data is accurately transferred. Our chosen method of communication between the two platforms was "**TCP/IP**" which allowed us to communicate between Matlab and C++ by sending JSON encoded values using an IP and port because of its reliability. This TCP/IP communication was achieved with the help of state machines which will be discussed later.

When UDP and TCP connection was compared between the agent(Matlab) and the environment(C++) it was found that Matlab always took 8ms to send data regardless of the communication protocol(TCP or UDP).

Flow Diagram

To control the robot in the environment(C++), the agent(Matlab) sends six actions namely the left and right of Hip Pitch, Knee Pitch and Ankle Pitch. The environment(C++) then sends observations (Velocity, Measured Angles, Robot Height etc.) back to the agent(Matlab) which are used for determining the best actions and fed to the reward function. We took help of state Machines, to control the flow of the simulation when the agent starts receiving values and environment starts sending values and vice versa. For this reason, as soon as the connection between the Matlab server and C++ client is established we send a set of initial robot observations from the environment for the agent. The state Machine which is a crucial part of this project starts with the Robot state in 'IDLE', which puts the robot in standing position and waits for the agent to send values to the robot. After receiving the actions (joint angles) the state is changed to 'LEARN' in which our system sends and receives the values synchronously between the agent and the

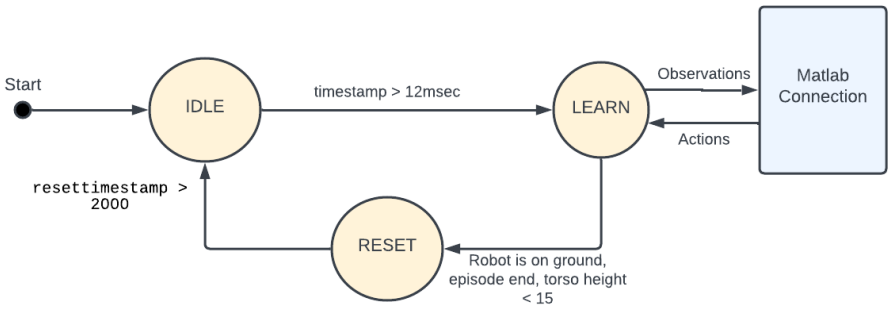


Figure 4.1: Flow Diagram

environment. To change the state to 'RESET' the robot needs to fulfill either one of the 3 situations.

1. If the environment detects that the robot has fallen down completely.
2. If the height of the robot is less than 15 cm which signifies that the robot is falling.
3. If 10 seconds has passed (which signifies that the training episode has ended).

When one of these conditions are fulfilled the state is set to 'RESET' and then the robot is put back to the initial position and the state is set to 'IDLE' again. In this way, the state machines completely determine the robot state through out the training process.

4.1.2 Selection of RL Algorithm

Reinforcement learning is a type of machine learning that involves an agent interacting with an environment to learn a policy that maximizes a reward signal. The goal of RL is to teach an agent to make decisions that result in the most beneficial outcomes in a given environment. Choosing an appropriate RL algorithm is crucial to achieve good performance in a given task. Different RL algorithms are designed to address different types of problems and environments, and they vary in terms of their strengths, weaknesses, and computational requirements.

TD3 Algorithm

TD3 can be a good choice for a wide range of continuous control tasks, where the goal is to learn a policy that can take actions in an environment to maximize a cumulative reward.TD3 is a good choice for a number of reasons:

1. It achieves state-of-the-art performance on continuous control tasks.
2. It addresses some of the weaknesses of the DDPG algorithm.
3. It uses twin Q-value functions to reduce overestimation and improve performance.

4. It employs delayed updates to improve stability and prevent over fitting, and is able to handle high-dimensional state and action spaces, sparse reward signals, and reduce variance in the Q-value estimates.

Overall, TD3 is a powerful and flexible algorithm that can be a good choice for many RL problems, particularly those that involve continuous control tasks, high-dimensional state and action spaces, sparse reward signals, or require long-term planning. The disadvantage of other algorithms and the reason to choose TD3 was discussed in the section 2.2

TD3 trains a deterministic policy in an off-policy way. Because the policy is deterministic, if the agent were to explore on-policy, in the beginning it would probably not try a wide enough variety of actions to find useful learning signals. To make TD3 policies explore better, we add noise to their actions at training time, typically uncorrelated mean-zero Gaussian noise which is explained later in the section 4.2.

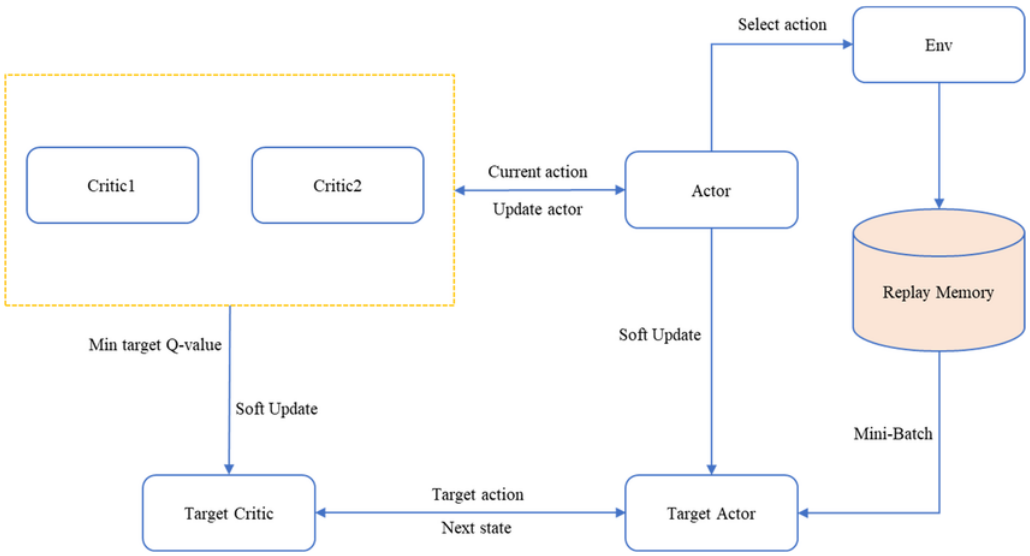


Figure 4.2: TD3 Algorithm Flowchart

4.1.3 Joint Angles extraction using walk request

In order to implement Imitation Learning it was decided to extract walking joint angles using an existing walking request. The VS code in C++ and the Matlab code were modified in accordance with this task: the walking command of 250 mm/s in direction of X axis was provided on the simulation side using the console window for a duration of one hundred seconds, the angles were sent in JSON format, and then they were received on the Matlab side respectively. Additionally, the following parameters were sent as well:

- orientX, orientY;
- Indicators of whether left and right feet touched the ground;

The obtained data was saved in a form of a struct with dimensions 1 by 3491 with 15 fields each. Figure 4.3 displays the obtained 11 joint angles plotted over time. Values starting

from 444 represent the actual walking. These values will play essential role later in the project.

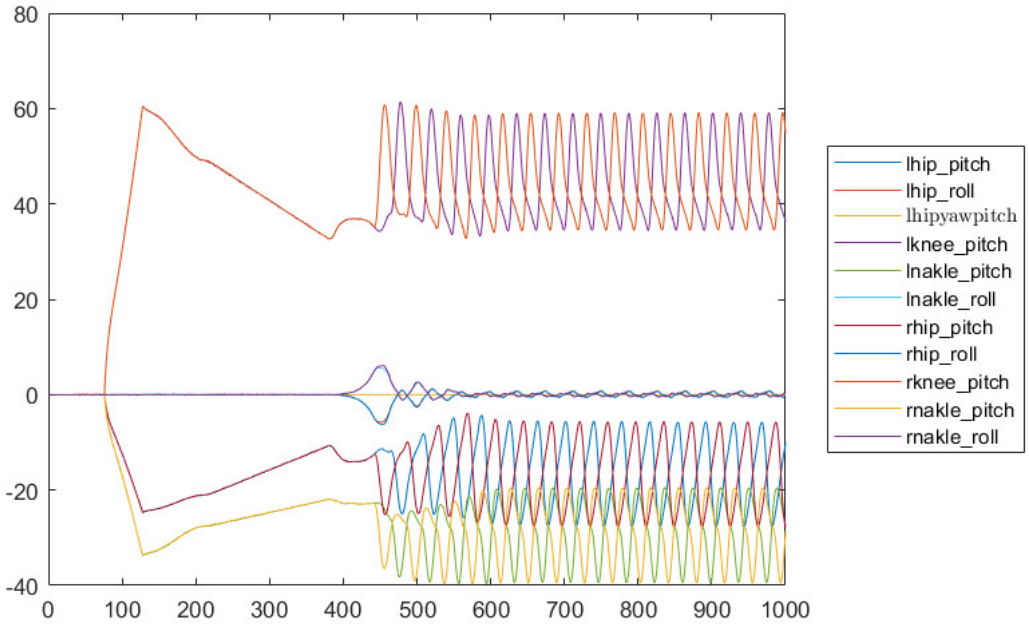


Figure 4.3: Joint angles of walking request

4.1.4 Selection of Joint Angles

To train our robot to walk we needed to accumulate our action space with all relevant joint angles.In our initial approach we decided to train on all leg joint angles, however it was soon apparent that it created a huge data set which was hard to train. In response to this we made a selection of all leg Joint Pitches.

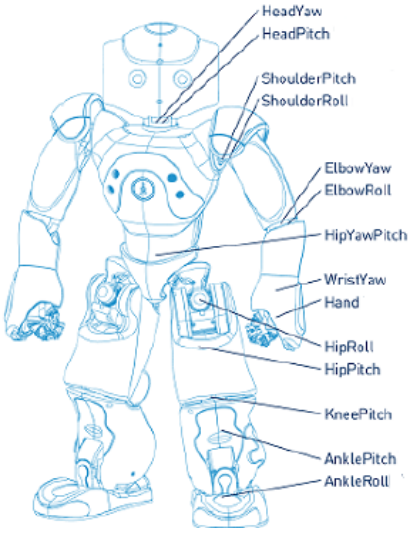


Figure 4.4: Joint Angles Overview

4.1.5 Selection of Observation Parameters

The selection of observation parameters is relevant for Reinforcement Learning (RL), as RL agents use observations to make decisions and take actions. The selection of observation parameters is important because it determines the information that the agent can perceive and use to make decisions.

In RL, the goal is typically to learn an optimal policy that maps observations to actions to maximize a cumulative reward signal. Therefore, selecting relevant observation parameters can significantly impact the performance of an RL agent. By selecting informative observation parameters, an RL agent can reduce the state space and improve its learning efficiency. Moreover, choosing observation parameters that capture important aspects of the environment can help the agent learn a more accurate representation of the state space, leading to better decision-making capabilities.

The list of observations which are used are mentioned as follows:

- Orientation X and Y : The orientation of the robot in the x and y describe the rotation around x and y direction respectively. These parameters are an essential observation parameter for robotic systems, as it helps the agent perceive the robot's position and movement in the x and y direction, leading to more accurate and efficient navigation and control

- Current Joint Angles and Joint Velocities: To help the agent take better actions the observed six joint angles are sent from the framework to the agent. This provides the agent with the observed angles of the robot in simulation to help in learning parameters like Joint velocities. This provides the agent with an idea of how quickly the agent can change its values.

- Velocity in X : Velocity in the X direction is an important input to a reinforcement learning algorithm, as it enables the robot to adjust its movements for appropriate speed and direction. It can also affect the robot's stability and balance, so incorporating velocity in the X direction as an input can help the robot perform more accurately, efficiently, and safely, especially in complex environments.

- Lateral Displacement : The lateral displacement needs to be observed by the agent as the lateral displacement needs to be as close to zero as possible. Even when changing only the pitches, the kinematic structure of the robot enables the robot to move in the y direction which was unexpected. This can be avoided by providing the lateral displacement as an observation which is an undesired value as will be discussed in the reward function.

- Torso Height : In our system we have also included the Torso height of the robot which decreases when the torso / robot falls down. This allows the robot to keep an upright position and penalize the reward as the robot falls.

4.1.6 Selection of Reward function

The selection of a correct reward function is crucial in Reinforcement Learning (RL) because the reward signal guides the agent's learning process. The reward function determines the agent's goal and provides feedback on the quality of its actions. Our initial reward function included the velocity as a positive reward, and the summation of orientation of x and y. The normalized torso height(Initial Height - Actual Height) was provided as a penalty if the torso deviates too much from the Initial Position. Additionally, the joint penalty was included to help reduce actuator effort. This reward function was trained for over 10,000 episodes on different PC's, the agent was unable to learn a policy in the end because of the complexity of the reward function. Hence a simpler approach was discussed which focuses on building the reward function sequentially.

An important part of building reward functions are the respective weights associated with each term, all of the weights were selected using trial and error method which obtained the highest average reward.

Forward velocity and duration

In our next attempt we decided to keep it simple and provide the velocity in x direction as a positive reward to give the robot a push forward. In addition to duration reward the robot was given a time duration reward to help avoid any local minima through out the training process. The reward function can be summarized as follows

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf$$

Forward velocity,duration and torso height

After incorporating the base reward function, the robot began exhibiting erratic behavior, including jumping. To address this, we implemented a penalty for torso height. We gave it an incentive to stabilize around the upright robot height, which was provided as a constant. Again the reward function equation can be written as below :

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z$$

With lateral change penalty

In response to the robot's tendency to walk towards the left or right, we updated the reward function to include a specific range for the robot's movement in the y-direction. This range will provide the robot with a clear boundary for how far it can move left or right, and will incentive it to maintain a more centered trajectory while walking. By implementing this range, we hope to improve the overall stability and accuracy of the robot's movements.The reward function can be stated as below :

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z - w3 \cdot Y$$

Reducing actuator effort

In the end we noticed that the joint actuators were fluctuating towards two extremes of the joint values which is both dangerous for the robot and the energy consumption. To counter this, we provided the joint deviation as a penalty. The reward function can in turn be defined as below:

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z - w3 \cdot Y - w4 \cdot (u[1]^2 + u[2]^2 + ... + u[6]^2)$$

4.1.7 Imitation Learning

Imitation learning is a technique of observing an expert's action in an environment. The agent mimics the action of the expert. Imitation learning works by extracting information about the behaviour of the teacher and the surrounding environment including any manipulated objects, and learning a mapping between the situation and demonstrated behaviour. The purpose of imitation learning is to efficiently learn a desired behaviour by imitating an expert's behaviour. Any system that requires autonomous behaviour similar to human experts can benefit from imitation learning. It learns a policy to execute the given task by imitating the demonstrated motions. In many cases, the experts are human operators and the learners are robotic systems. In this scenario, the robot is fed with predefined angles to walk. The agent learns the neural network of itself and then that policy is used for the walking.

The idea is to use the 6 joint angles of the NAO robot to walk. Pitches of the right knee, left knee, right angle, left ankle, left hip and right hip are the 6 joint angles used

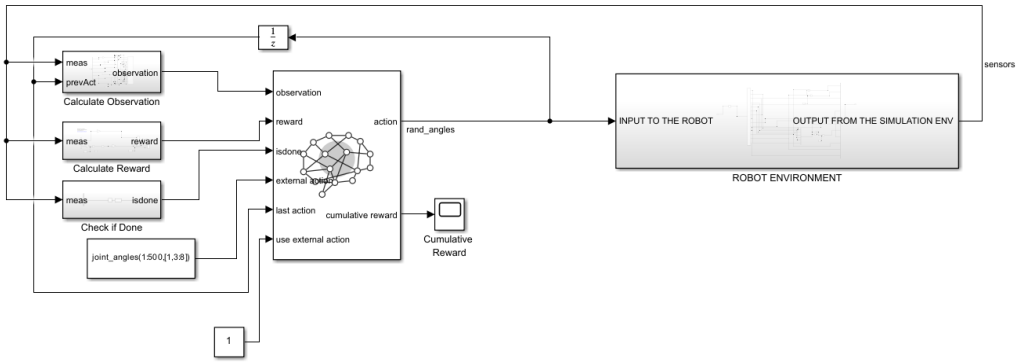


Figure 4.5: External action Block

for walking. The prerecorded values for the above joint angles are used for the imitation learning as explained in the subsection 4.1.3 and the values are called as joint angles.

The joint angles are used in the Simulink as a block and fed to the agent as an external action signal. When the value of the use external action signal is 1, it passes the external action signal to the environment through the action block output as shown in figure 4.5. The agent policy is updated based on the resulting observations and rewards.

Two approaches that are used:

1. Train the robot using Imitation learning for the whole simulation time 10s.

2. Train the robot for 6s using imitation learning and for the rest of the 4s it trains on its own.

The motivation of first approach is to train the actor and critic network with the experts' behaviour for 2000 episodes and use the resulting agent's policy to simulate the robot in the same environment. With this approach, the rewards are consistent and it is abundant to make the robot learn its pattern.

The motivation of the second approach is to train the actor and critic network with the experts' behaviour for 2000 episodes for 6 seconds and let the agent train on its own for 4 seconds. In this approach, the last 4 seconds are used for exploration so that the agent's policy is not limited due to excessive exploitation.

4.1.8 Inverse Reinforcement Learning

Imitation learning involves an algorithm learning to imitate a behavior it has seen in a data set. In contrast, Inverse Reinforcement Learning (IRL) involves an algorithm learning the reward function that caused the observed behavior. By subtracting the reinforcement learning output from the desired joint angles of a robot, the agent can determine the deviation from the expert demonstration in terms of joint angles. This deviation serves as a type of reward signal for the agent to learn from. The aim is to use the external action as a reference point instead of incorporating the exact angles that are required to walk.

In general, Simulink consists of an Environment and an Agent. The actions are given to the environment and the environment transfers the observation to the agent. Due to this approach, the robot takes a longer time to reach the desired walking pattern. Imparting the joint angles as a reference and subtracting the output produces an error term which allows the robot to take better steps as shown in the figure 4.6

Here, the error term is fed to the observation and reward block as shown in the figure 4.7. This is similar to the general feedback control and the reward function is modified to produce more reward for less error. The error term is squared and summed in the reward function. The reward function of this were discussed in the subsection 4.1.6. The experiment results of Inverse Reinforcement Learning are discussed in the subsection 4.3.2.

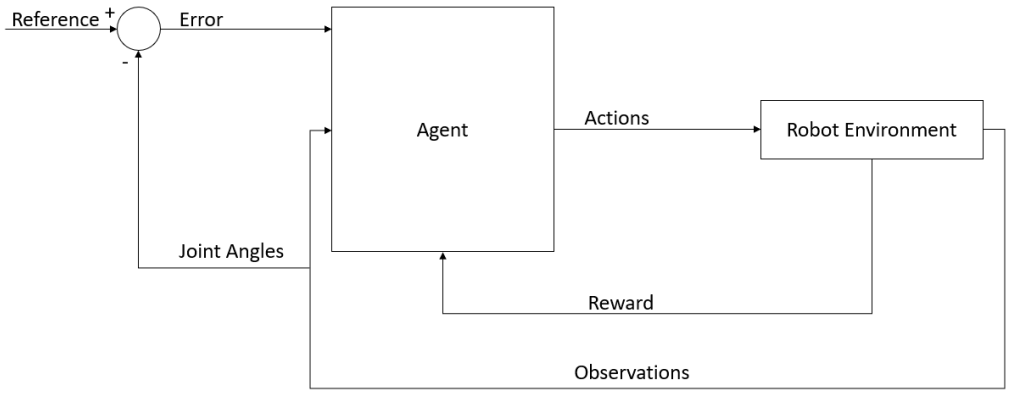


Figure 4.6: Inverse Reinforcement Learning

4.1.9 Layers

Using Matlab it's possible to investigate the layers of the trained agent's actor. The number of layers is 8 and they are as follows:

1. **input**: input Layer, 23 entries.
2. **fc**: In Fully Connected(FC) layer, all the neurons of the input are connected to every neuron of the output layer, 256 fully connected layer.
3. **relu body**: A Rectified Linear Unit(ReLU) layer returns max(x,0) where x is input.

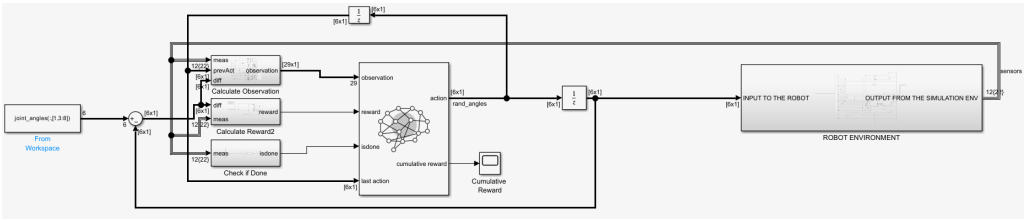


Figure 4.7: Error minimization in Simulink

4. **fc body**: FC layer, 256 fully connected layer.
5. **body output**: ReLU layer.
6. **output**: FC layer, 6 fully connected layer.
7. **tanh**: A hyperbolic tangent (tanh) activation layer,

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

8. **scale**: Scaling layer linearly scales and biases an input array, giving an output.

4.2 Hyper Parameter Tuning

Hyper parameter tuning is an omnipresent problem in machine learning as it is an integral aspect of obtaining the state-of-the-art performance for any model. Most often, hyper parameters are optimized just by training a model on a grid of possible hyper parameter values and taking the one that performs best on a validation sample (grid search). Setting the right hyper parameters could have a huge impact in the performance of the agent[15]. Matlab reinforcement learning designer app provides with an default parameter that presents a decent output. But fine tuning some of the hyper parameters results in a huge impact. The tuned hyper parameters are explained in the below subsection.

4.2.1 Neural Network Parameterizations

First, the agent is initialized with the default neural network structure. The neural network trains to learn the best reward situations for state and action pair mappings. Therefore, not only do we need optimal hyper parameters that can allow the agent to reach optimal reward in fewer episodes. Increasing the layers allows for learning complex actions and patterns such as walking and the optimal number of neurons prevents over fitting. The Recurrent Neural Network (RNN) is primarily used for sequential data and time series data but to decrease the computational effort RNN was ignored.

4.2.2 Agent Options

The agent options provide a wide range of choices to train and test reinforcement learning agents and allow users to customize and build agents for their specific needs as described below.

Sample Time

The sample time of an agent determines how frequently the agent will provide an output action or decision. In our case, the agent sample time is 0.012 which means the agent takes action every 0.012 seconds and takes 83 steps per second that is given by the hardware.

Discount Factor

The discount factor usually ranges between 0 and 1. The discount factor is set to 0.99 which indicates that it gives preference for future rewards more than the immediate reward.

Execution Environment

To increase the speed of the training process, GPU is used as an execution environment.

Batch Size

The default chosen size for all the training is 128. It means that the agent uses 128 transitions in each iteration to update its policy.

Experience Buffer Length

To learn from a larger and more diverse set of transitions and where the state space is large or policy requires a long sequence of actions to achieve a goal. The Experience buffer is set to 100,000 which stores up to specified number of experiences.

Policy and Target Update Frequency and Smooth Factor

The policy and target update frequency is set to 2 so that it updates every 2 steps. Since the robot takes 83 steps per second, both the policy and network are updated approximately 40 times every second. The smooth factor is only supported for TD3 agent.

Additionally, the other two parameters in the RL designer App are reset and save buffer. We have chosen the buffer to be reset so that the behaviour that is experienced during an episode had reset and starts from scratch in the next episode. This allows the robot to explore different walking patterns and not get stuck in a specific behaviour and to get an optimum performance.

4.2.3 Actor and Critic Optimizer Options

Several parameters are used to tune the actor and critic network. But only the parameters which are tuned in the RL designer app are discussed in this section.

Learn Rate

Learn rate is a hyper parameter that determines the step size to update the parameters of the model during training. The learning rate is set to 0.001 for most of the training with and without joint angles so that it learns slowly and does not overshoot.

L2 Regularization

L2 regularization adds a penalty term(lambda) to the loss function which enables learning smaller weights. It prevents over fitting of the data. It is defined as 4.1

$$L2term = \lambda * (w_1^2 + w_2^2 + ...W_n^2) \tag{4.1}$$

4.2.4 Exploration Model

The goal of an exploration model is to find the optimal policy by encouraging the agent to learn from new experiences. Gaussian noise exploration strategy adds random noise to the agent's actions to prevent it from getting stuck into a sub optimal policy and allow it to explore new actions.

Standard Deviation

In general, the variance can be set between 1 and 10% of the action range. It is found [1], the agent takes optimal actions when it is $4.5 * 0.05/\sqrt{(SampleTime)}$ that is 5% of the action range. Based on the calculations performed, the standard deviation is set to 0.046, resulted in optimal rewards and actions for the agent. The rate at which the variance will decay is set to 0.0001 by trail and error.

4.2.5 Target Policy Smoothing Model

The target policy smoothing with Gaussian noise adds noise to the action selected by the policy to improve the stability and robustness of the learned policy.

$$SmoothFactor = clip(policy + N(0, \sigma)) \tag{4.2}$$

Where:

- clip() - it is a function that clips the actions withing the action limits
- $[N(0, \sigma)]$ - Gaussian noise with zero mean and standard deviation $\sigma - 0.1095$

4.3 Experimental Results

This section attempts to show the results achieved in with Reinforcement Learning, Inverse RL and Imitation Learning.

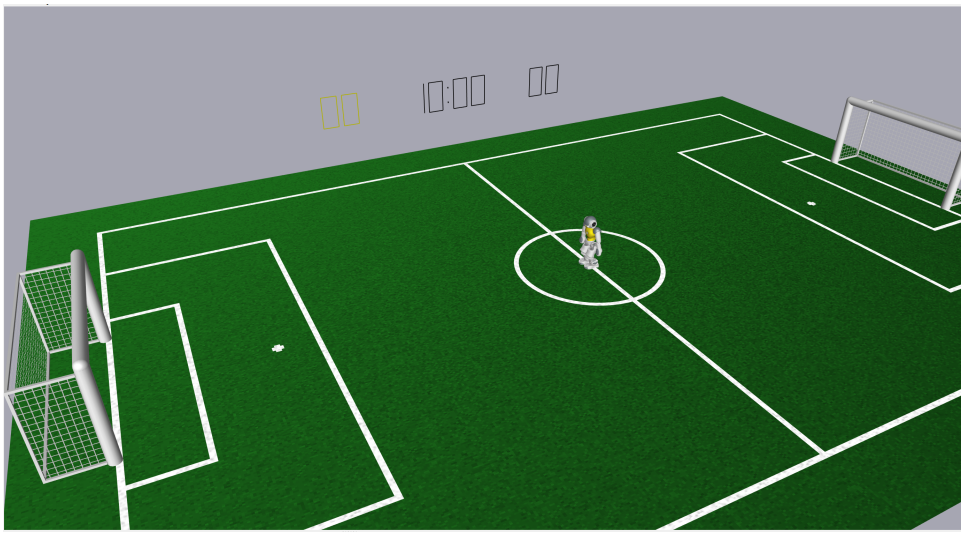


Figure 4.8: Initial Position

4.3.1 Reinforcement Learning

In this section we will discuss the results achieved for the different Reward Functions and effects of different parameters on the training results for the Reinforcement Learning part. The initial position as well as the reset position can be seen in the figure 4.8 which was used for all results for concurrency.

Reward Function with Velocity and Duration

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf$$

Let us discuss the first reward function where Velocity in the X direction(longitudinal direction) and a duration reward at each time step was added to shape the reward function.

This approach served as the base for the training where the robot tried to maximize the Reward function and helped to determine the correct settings for the learn rate, exploration policy, action clipping and other hyper parameters. The idea was that if the robot is able to maximize its reward using the current hyper parameters then those hyper parameters would be continued in the later experiment.

The Experiment with this reward function was a failure since the robot did not have any clue on how to maximize the reward function without falling down. It learned a policy which makes it fall down quickly to achieve the best Vx or simply stay upright to collect the duration reward.

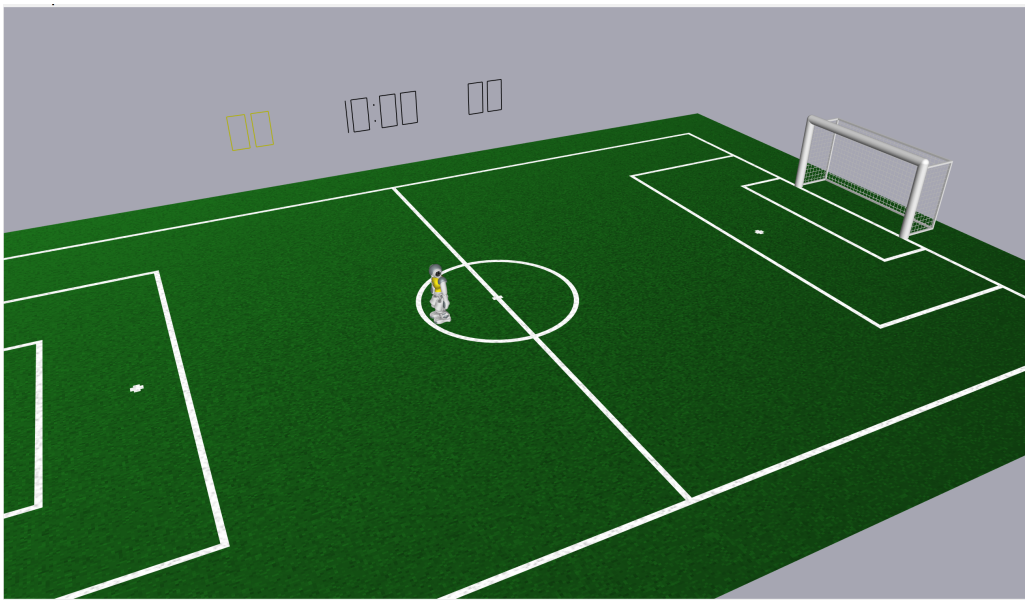


Figure 4.9: Results with Velocity and Duration.

It can also be seen by the graph 4.10 which shows that the maximum reward achievable using this reward function was only 150. The dark blue is the average reward, then light blue is the each episode reward, and the yellow line represents the q_0 value. The reward achieved by using the prerecorded angles was as high as 900 even more than that depending upon the weights we set. The current Reward function does not tell the agent to not fall down since no parameter in the reward function penalizes the falling motion. This served as the motivation for the next reward function.

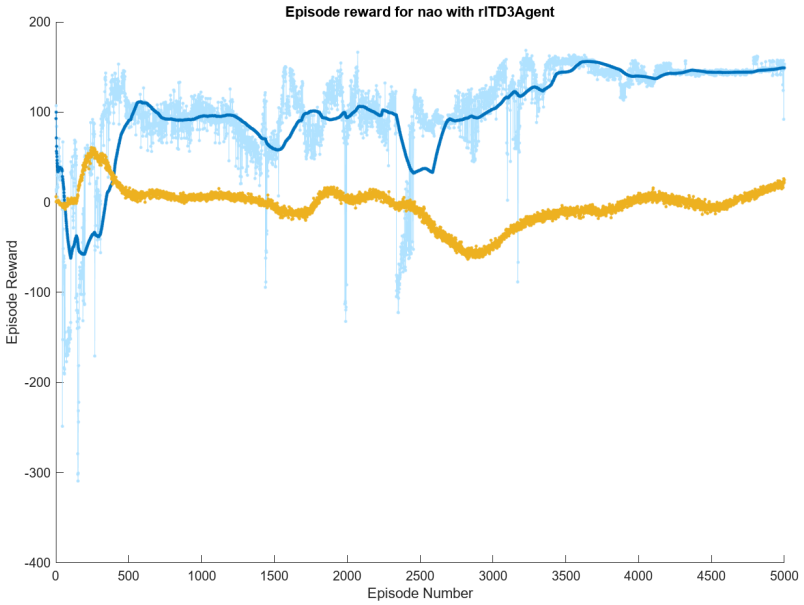


Figure 4.10: Episode Reward with velocity and duration.

With Torso Height Penalty

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z$$

This reward function attempted to build on the work done before, where Vx and duration reward are incorporated with the vertical penalty to form the reward function.

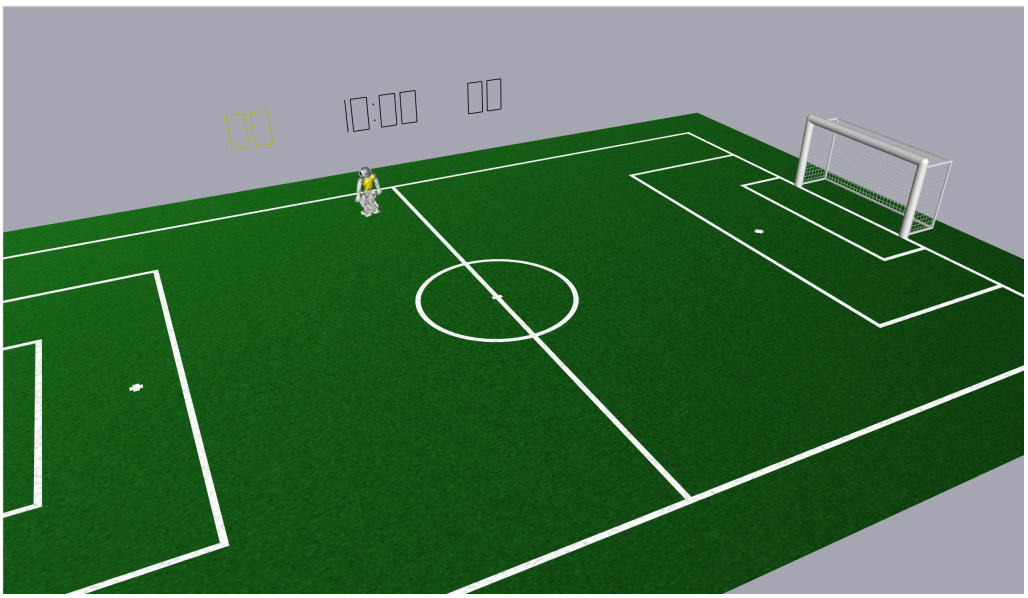


Figure 4.11: Results with Torso Height Penalty Included.

This complicated the task as finding the correct w1 and w2 which form the weights of Vx and z(robot height from ground to torso) and describes which parameter affects the reward function more. As a result w2 is much bigger than w1. With w2 too bigger than w1 the robot learns a policy similar to the previous reward function(falling down as soon as the episode starts to not get penalized for longer periods of time) and with w2$<$ w1 the robot learns a policy to stay up but do nothing. After some trial and error eventually we found the correct settings which made the robot walk but in a rather unconventional way.

The Experiment with this reward function successfully achieved the first correct results where the robot could hop from a point A without falling down through out the training episode. But the hopping was not in a straight line.In addition to the robot walking side ways, the reward can also be seen in the graph 4.12. Even tough it is much higher from the previous function given the fact that the robot has started to walk, the reward is still merely 700. This is again less than the reward of the prerecorded angles. For this reason the next reward function introduces a lateral penalty. It can also be deduced that even tough the robot went to the side the distance covered was a lot more than the previous function.

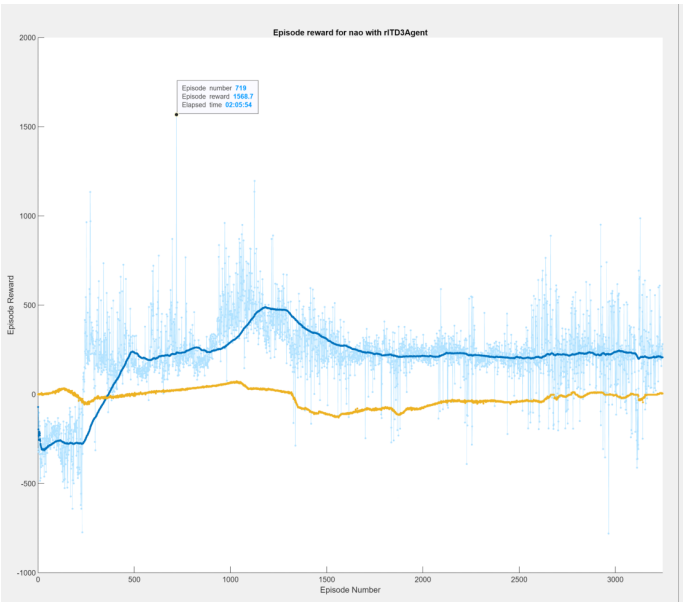


Figure 4.12: Episode reward with Torso Height Penalty Included

Lateral Movement Penalty

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z - w3 \cdot Y$$

This reward function is also an extension of the previous reward function which successfully learn a policy which relates the joint angles to some kind of a robot motion.

However as discussed before, the motion of the robot was not towards the goal post(straight line) and for this reason a lateral penalty was introduced. The first policy learned with this reward function was sub optimal since the robot failed to even replicate the previous achievements. For this reason the reward function was shaped in a way that instead of the Vy the $Max(abs(Y) - 40, 0)$ was used. This expression sets a threshold at 40 cm for the absolute value of Y. If Y is less than 40 cm, then the expression evaluates to zero, meaning that there is no effect. If Y is greater than or equal to 40 cm, then the expression subtracts 40cm from Y, which has the effect of reducing Y if it is above the threshold of 40 cm. This experiment was also a success which induced a kind of a road for the robot which it uses to hop in a straight line without falling down. Since hopping requires a lot of actuator effort and changes in joint angles. The final reward function attempted to reduce the actuator effort.

With Joint Angle Penalty:

$$R = w1 \cdot Vx + 25 \cdot Ts/Tf - w2 \cdot z - w3 \cdot Y - w4 \cdot (u[1]^2 + u[2]^2 + ... + u[6]^2)$$

The above reward function tries to maximize the reward function by moving forward, not falling down, attempting to walk in a straight line and also reduce the actuator effort.

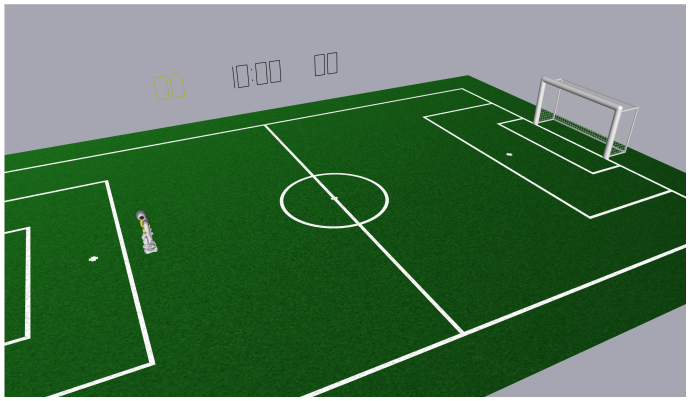


Figure 4.13: Distance Covered by Lateral Penalty

The problem however is that if w4 is set too small the actuator penalized is not realized by the agent and if the actuator penalty is too large the agent does not learn motion with the current exploration settings. Hence the exploration parameters need to be revisited for this particular case. Figure 4.15 show the training results achieved for this reward function.

The best results achieved with the latest reward function show that our agent is able to train the robot to walk much farther than the previous reward function. But the disadvantage is that the robot utilizes too much energy.

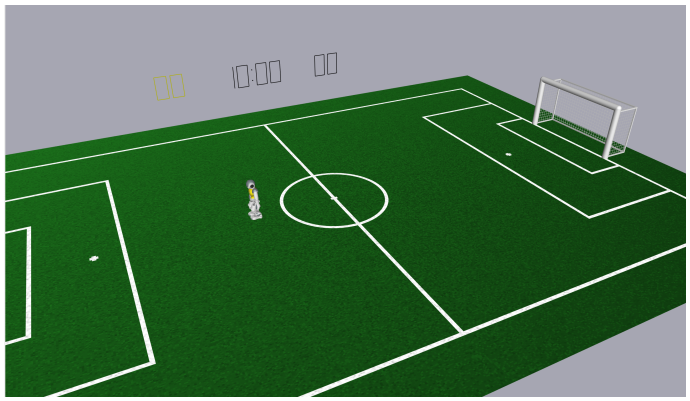


Figure 4.14: Training results with the Reward in X, Penalty in Y, Penalty in Z height, Duration reward and actuator effort penalty

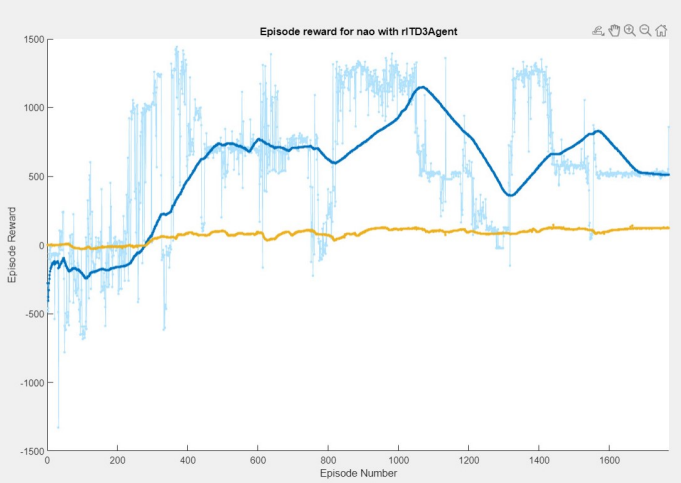


Figure 4.15: Episode Reward with actuator effort penalty

4.3.2 Using External Action

When the "**use external action**" signal is set to 1, the external action is set to true and the joint angles are used as an external action as highlighted in the figure 4.16. After several episodes of training using the joint angles, the robot produced none of the joint angles except the from the first time stamp. Even though the agent had a higher reward during the training, it is observed that the agent did not learn anything but was overridden by the external action block.

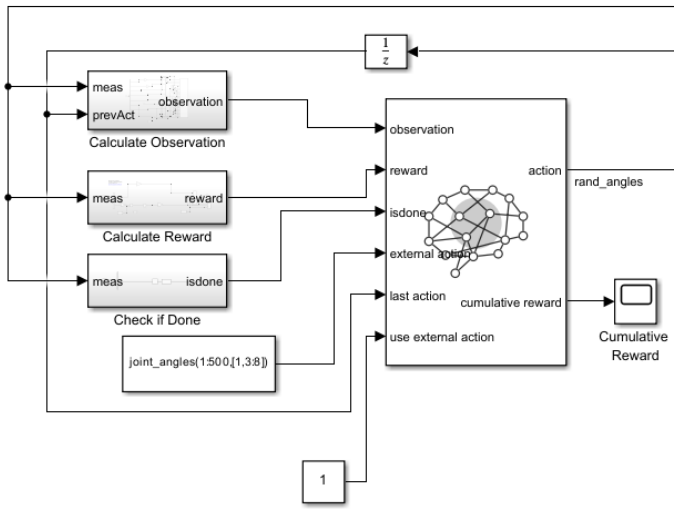
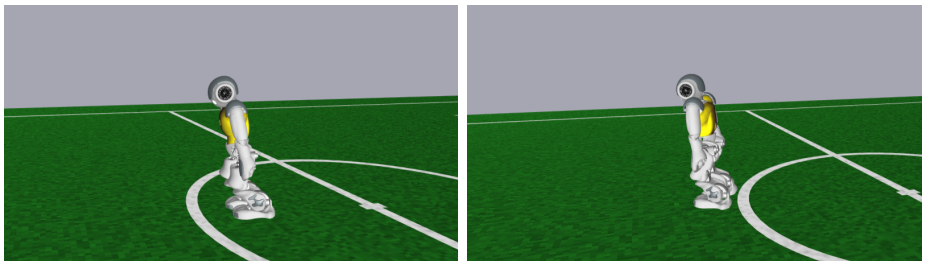


Figure 4.16: External Action Signal

The joint angles are used as a reference as described in 4.1.8, the agent showed an enormous performance peak with 29 observations and 6 actions. The robot walked at its maximum speed until the end of the arena from the centre after just a few episodes of training. It is observed figure4.17 that the robot was able to walk faster when it leaned

forward during the walking motion.The robot adapted its own method to walk faster. The training results of about 1000 episodes are shown in the figure4.18.The average reward stayed around 500 denoted in dark blue and the maximum reward went till approximately 2200 in the episode 720. In the subsection 4.1.3, it was explained how important these



(a) Initially (b) After Few Steps

Figure 4.17: Robot leaning during training

limits are for the proper functioning of the robot. The robot was then simulated for ten episodes using a trained TD3 agent in the Reinforcement Learning Designer toolbox in Matlab, with each episode lasting a maximum of 833 steps. The simulation results are presented in the figure 4.19. The maximum reward of around 4500 was achieved in episodes three and ten. The figure 4.20 also shows that the joint limits were useful in helping the agent maintain the robot's walking pattern. The foot angle was subject to joint limits, and the robot's environment observations were kept within these limits to preserve the walking pattern.

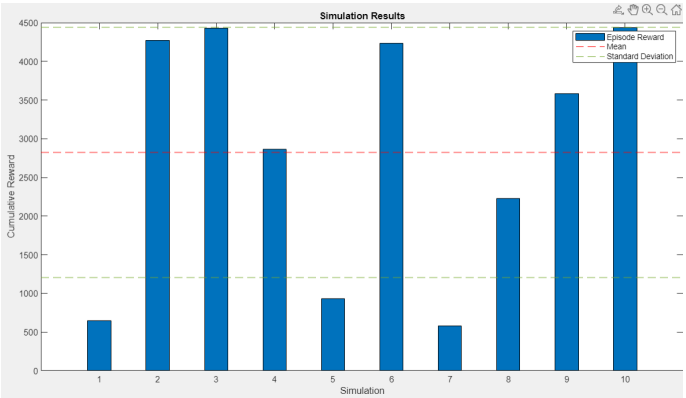


Figure 4.19: Simulation Result of TD3 Agent

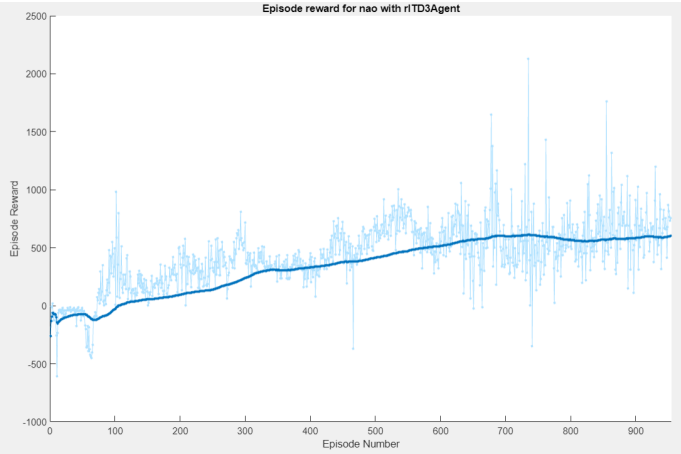
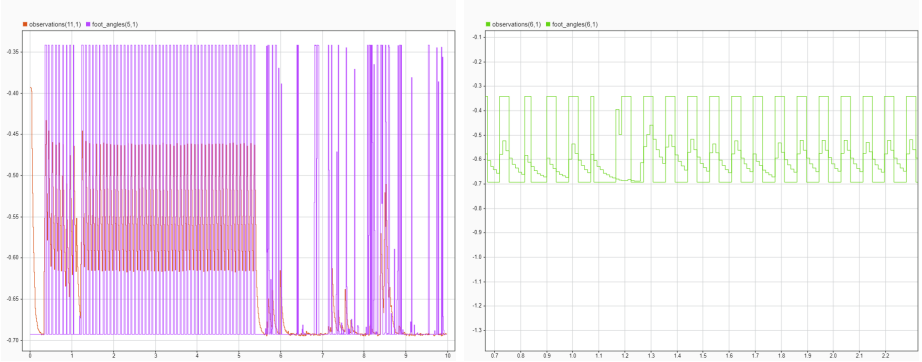


Figure 4.18: Training results with semi supervised learning



(a) Left Ankle Pitch (b) Right Ankle Pitch

Figure 4.20: Joint Limits

4.4 Conclusion

The project aim is to train the robot to walk using RL. The reward signal is designed to optimize a robot's walking technique, such as balancing and walking in a straight line. However, there was an issue during training where the communication of joint angles between simulator and Matlab was not fast enough for the robot to take the required 83 steps per second. As a result, training was slow and took a long time. Nevertheless, the desired output was achieved in few thousand episodes. During the simulation the robot was able to achieve approximately 50 to 60 steps per second. Although the motion was stable and the robot was able to walk for a longer distance, the resulting motion is too aggressive to be implemented on the real robot. This is because the reward function focused more on forward velocity, balancing, duration, and z-axis height, resulted in a motion that was too fast for the real robot to handle. Despite this, the project's objective is accomplished with Reinforcement Learning and Inverse Reinforcement Learning techniques, and further tuning of the robot's joint motion is needed to make it work on the real robot.

Chapter 5

Discussion and Outlook

The objective of the project group was achieved by employing two major approaches: Model Predictive Control and Reinforcement Learning.

MPC has been used to calculate the optimal input and trajectory for the upper-body control of Nao robot through an analytical solution. A complementary filter was used to fuse estimated and measured data to estimate the robot's position, velocity and acceleration. During the testing, the use of this complementary filter has shown to produce comparatively better results in terms of reducing tracking errors and improving the stability of the robot. The implementation of MPC can be made easier by the integration of LibMPC++, an open-source C++ library for MPC-based control strategies.

Future research in MPC-based upper body control of Nao robot could focus on improving the optimization of the MPC controller, enhancing the constraints, and regulating the error in fewer frames. Moreover, state estimators like Extended Kalman Filter [17], Unscented Kalman Filter [24], Moving Horizon Estimator [18] can be used in future instead of complementary filter for more accurate state estimation. Additionally, the development of learning-based MPC algorithms and the integration of machine learning techniques could lead to the development of more advanced and versatile control strategies for the Nao robot. Overall, the use of MPC with complementary filter show promise in achieving smooth and accurate upper body motion control of the Nao robot.

Reinforcement learning involves optimizing a policy through trial and error to maximize a cumulative reward signal. However, faster communication between Matlab and the simulator was a challenge, as the robot only trained at approximately 20 steps per second in SimRobot and 60 steps per second during simulation. The communication protocol could be improved in the future to achieve the desired speed of 83 steps per second. The project's main goal is to train the robot to walk as far as possible at an appropriate velocity, and the observation parameters and final reward function were constructed to meet this objective. However, the resulting motion is currently too aggressive for deployment on a real robot, and the weights of the forward velocity need to be adjusted accordingly in the future.

The Twin Delayed Deep Deterministic Policy Gradient (TD3) algorithm was chosen because it handles high dimension of action and state spaces in continuous control tasks and reduces high variance compared to other algorithms. Another reason for choosing TD3 is that it allows the agent to have a wide exploration range to prevent overestimation. Imitation learning was not effective in this project since the agent was simply overridden by prerecorded joint angles without being trained. Instead, the Inverse Reinforcement Learning (IRL) method was used by subtracting the actual joint angles from the desired joint angles to allow the agent to determine the deviation and correct its walking technique. The same hyper parameters were used for both RL and IRL techniques, and the results from IRL were better than RL. Further tuning of hyper parameters and adjusting the weights of the reward function will be necessary for implementing this on a real robot.

List of Figures

3.1 The MPC scheme diagram 8
3.2 Calculated ZMP position, reference ZMP position and actual CoM position. 12
3.3 Applied control input. 12
3.4 Error in estimated and measured positions of CoM. 13
3.5 Calculated ZMP position, reference ZMP position and actual CoM position for Case 1 of error dynamics 3.2.3. 14
3.6 Error in estimated and measured positions of CoM. 14
3.7 Calculated ZMP position, reference ZMP position and actual CoM position for Case 1 of error dynamics 3.2.3 when a force is applied. 14
3.8 Error in estimated and measured positions of CoM when a force is applied. 15
3.9 Calculated ZMP position, reference ZMP position and actual CoM position for Case 2 of error dynamics 3.2.3. 15
3.10 Error in estimated and measured positions of CoM. 16
3.11 Calculated ZMP position, reference ZMP position and actual CoM position for Case 2 of error dynamics 3.2.3 when a force is applied. 16
3.12 Error in estimated and measured positions of CoM when a force is applied. 16
3.13 Calculated ZMP position, reference ZMP position and actual CoM position for Case 3 of error dynamics 3.2.3. 17
3.14 Error in estimated and measured positions of CoM. 17
3.15 Calculated ZMP position, reference ZMP position and actual CoM position for Case 3 of error dynamics 3.2.3 when a force is applied. 17
3.16 Error in estimated and measured positions of CoM when a force is applied. 18
3.17 Complementary filter for fusion of states (3.13) 18
3.18 Calculated ZMP position, reference ZMP position and actual CoM position for Case 4 of error dynamics 3.2.3. 19
3.19 Error in estimated and measured positions of CoM. 19
3.20 Calculated ZMP position, reference ZMP position and actual CoM position for Case 4 of error dynamics 3.2.3 when a force is applied. 19
3.21 Error in estimated and measured positions of CoM when a force is applied. 20
3.22 Calculated ZMP position, reference ZMP position and actual CoM position for Case 5 of error dynamics 3.2.3. 20

3.23 Error in estimated and measured positions of CoM. 20
3.24 Calculated ZMP position, reference ZMP position using libmpc++. 23

4.1 Flow Diagram 26
4.2 TD3 Algorithm Flowchart 27
4.3 Joint angles of walking request 28
4.4 Joint Angles Overview 29
4.5 External action Block 32
4.6 Inverse Reinforcement Learning 33
4.7 Error minimization in Simulink 33
4.8 Initial Position 37
4.9 Results with Velocity and Duration. 38
4.10 Episode Reward with velocity and duration. 38
4.11 Results with Torso Height Penalty Included. 39
4.12 Episode reward with Torso Height Penalty Included 40
4.13 Distance Covered by Lateral Penalty 41
4.14 Training results with the Reward in X, Penalty in Y, Penalty in Z height, Duration reward and actuator effort penalty 41
4.15 Episode Reward with actuator effort penalty 42
4.16 External Action Signal 42
4.17 Robot leaning during training 43
4.19 Simulation Result of TD3 Agent 43
4.18 Training results with semi supervised learning 44
4.20 Joint Limits 44

Bibliography

[1] In: (). URL: https://de.mathworks.com/matlabcentral/answers/467153-noise-parameters-in-reinforcement-learning-ddpg.

[2] D. Adams. *The Hitchhiker's Guide to the Galaxy*. San Val, 1995. ISBN: 9781417642595. URL: http://books.google.com/books?id=W-xMPgAACAAJ.

[3] Schulman J. et al. "Proximal Policy Optimization Algorithms". In: 2017. DOI: 10.48550/arXiv.1707.06347.

[4] E. F. Camacho and C. Bordons. "Introduction to Model Predictive Control". In: *Model Predictive control*. Springer London, 2007. ISBN: 978-0-85729-398-5. DOI: 10.1007/978-0-85729-398-5_1. URL: https://doi.org/10.1007/978-0-85729-398-5_1.

[5] "Constrained model predictive control: Stability and optimality". English (US). In: *Automatica* 36.6 (June 2000), pp. 789–814. ISSN: 0005-1098. DOI: 10.1016/S0005-1098(99)00214-9.

[6] Zheng W. Dankwa S. "Twin-Delayed DDPG: A deep reinforcement learning technique to model a continuous movement of an intelligent robot agent". In: 2019, pp. 1–5. DOI: 10.1145/3387168.3387199.

[7] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. "A survey of actor-critic reinforcement learning: Standard and natural policy gradients". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42.6 (2012), pp. 1291–1307.

[8] K. Harada, S. Kajita, K. Kaneko, and H. Hirukawa. "An analytical method on real-time gait planning for a humanoid robot". In: *4th IEEE/RAS International Conference on Humanoid Robots, 2004.* Vol. 2. 2004, 640–655 Vol. 2. DOI: 10.1109/ICHR.2004.1442676.

[9] M. A. S. L. Cuadros J. C. Jesus J. A. Bottega and D. F. T.Gamarra. "Deep Deterministic Policy Gradient for Navigation of Mobile Robots". In: 2021, pp. 349–361. DOI: 10.3233/JIFS-191711.

[10] S. Kajita, O. Matsumoto, and M. Saigo. "Real-time 3D walking pattern generation for a biped robot with telescopic legs". In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*. Vol. 3. 2001, 2299–2306 vol.3. DOI: 10.1109/ROBOT.2001.932965.

[11] S. Kajita, O. Matsumoto, and M. Saigo. "Real-time 3D walking pattern generation for a biped robot with telescopic legs". In: *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*. Vol. 3. 2001, 2299–2306 vol.3. DOI: 10.1109/ROBOT.2001.932965.

[12] Shuuji Kajita. "Linear Inverted Pendulum-Based Gait". In: *Humanoid Robotics: A Reference*. Ed. by Ambarish Goswami and Prahlad Vadakkepat. Dordrecht: Springer Netherlands, 2018, pp. 1–18. ISBN: 978-94-007-7194-9. DOI: 10.1007/978-94-007-7194-9_42-1. URL: https://doi.org/10.1007/978-94-007-7194-9_42-1.

[13] Shuuji Kajita. "Linear Inverted Pendulum-Based Gait". In: *Humanoid Robotics: A Reference*. Ed. by Ambarish Goswami and Prahlad Vadakkepat. Dordrecht: Springer Netherlands, 2018, pp. 1–18. ISBN: 978-94-007-7194-9. DOI: 10.1007/978-94-007-7194-9_42-1. URL: https://doi.org/10.1007/978-94-007-7194-9_42-1.

[14] Shuuji Kajita, Fumio Kanehiro, K. Kaneko, Kiyoshi Fujiwara, Kensuke Harada, Kazuhito Yokoi, and H. Hirukawa. "Biped walking pattern generation by using preview control of zero-moment point". In: vol. 2. Oct. 2003, 1620–1626 vol.2. ISBN: 0-7803-7736-2. DOI: 10.1109/ROBOT.2003.1241826.

[15] Mariam Kiran and Buse Melis Özyildirim. "Hyperparameter Tuning for Deep Reinforcement Learning Applications". In: *CoRR* abs/2201.11182 (2022). arXiv: 2201.11182. URL: https://arxiv.org/abs/2201.11182.

[16] LJ. Lin. "Self-improving reactive agents based on reinforcement learning, planning and teaching. Machine Learning". In: 1992, pp. 293–321. DOI: 10.1007/BF00992699.

[17] Priya Shree Madhukar and L.B. Prasad. "State Estimation using Extended Kalman Filter and Unscented Kalman Filter". In: *2020 International Conference on Emerging Trends in Communication, Control and Computing (ICONC3)*. 2020, pp. 1–4. DOI: 10.1109/ICONC345789.2020.9117536.

[18] Mohamed W. Mehrez, George K. I. Mann, and Raymond G. Gosine. "Nonlinear moving horizon state estimation for multi-robot relative localization". In: *2014 IEEE 27th Canadian Conference on Electrical and Computer Engineering (CCECE)*. 2014, pp. 1–5. DOI: 10.1109/CCECE.2014.6901134.

[19] Esmaeil Najafi, Gabriel Lopes, and Robert Babuska. "Balancing a Legged Robot Using State-Dependent Riccati Equation Control". In: vol. 19. Aug. 2014. ISBN: 9783902823625. DOI: 10.3182/20140824-6-ZA-1003.01724.

[20] J. Peters and S. Schaal. "Policy Gradient Methods for Robotics". In: 2006, pp. 2219–2225. DOI: 10.1109/IROS.2006.282564.

[21] N. Piccinelli. "libmpc++: a library to solve linear and non-linear MPC". In: (). URL: https://github.com/nicolapiccinelli/libmpc.

[22] Athanasios S Polydoros and Lazaros Nalpantidis. "Survey of model-based reinforcement learning: Applications on robotics". In: *Journal of Intelligent & Robotic Systems* 86.2 (2017), pp. 153–173.

[23] Oliver Urbann, Ingmar Schwarz, and Matthias Hofmann. "Flexible Linear Inverted Pendulum Model for cost-effective biped robots". In: *2015 IEEE-RAS 15th International Conference on Humanoid Robots (Humanoids)*. 2015, pp. 128–131. DOI: 10.1109/HUMANOIDS.2015.7363525.

[24] E.A. Wan and R. Van Der Merwe. "The unscented Kalman filter for nonlinear estimation". In: *Proceedings of the IEEE 2000 Adaptive Systems for Signal Processing, Communications, and Control Symposium (Cat. No.00EX373)*. 2000, pp. 153–158. DOI: 10.1109/ASSPCC.2000.882463.

[25] Christopher JCH Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[26] Pierre-brice Wieber. "Trajectory Free Linear Model Predictive Control for Stable Walking in the Presence of Strong Perturbations". In: Jan. 2007, pp. 137–142. DOI: 10.1109/ICHR.2006.321375.