People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University of M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronics Engineering

Department of Power and Control

Final Year Project Report Presented in Partial Fulfillment of the Requirements of the Degree of

MASTER

In Electrical and Electronic Engineering

Option: Control

Title:

Adaptive Impedance Control for Unknown Environment

Presented By:

- **BENBOUALI Mohamed Amine**

Supervisor:

Mrs. N. Derragui

Registration Number: 2021/2022

Abstract

This report presents a simulation, design, build and evaluation of a 2 degree of freedom robotic arm guided by an adaptive impedance control algorithm governed by a neural network. This neural network is trained and evaluated by an adaptive genetic algorithm, the implementation of the robot arm consists of a 3D printed interconnected body controlled by Python and Arduino equipped with high torque servo motors. Interaction force sensors are installed around the end effector of the robotic arm as feedback to the neural network to exert manipulative decisions.

Acknowledgements

This report is entirely proposed by the supervisor Mrs. N. Derragui, all thanks and appreciation for suggesting the and supervising the progress of this project and report, pushing me to maximum ability and hard work.

Content List

Nomenclature

Ai: Artificial Intelligences

DOF: Degrees of freedom

ML: Machine Learning

NEAT: Neuro-Evolution of Augmented Topologies

Nozzle: The metal mechanism within which the plastic is melted and extruded through the nozzle's bottom opening

VIC: Variable Impedance Control

VIL: Variable Impedance Learning

VILC: Variable Impedance Learning Control

General Introduction:

In the foreseeable future, robots will be increasingly integrated into human society, serving human beings in various fields such as old-age care, education and entertainment, forming a new harmonious social environment with us [1], [3]. In recent years, more and more robots have appeared in the human environment of nursing homes, hospitals, shopping malls and even families [4].

Applying robot manipulators to a wider class of tasks, that will be necessary to control not only the position of a manipulator but also the force exerted by the end-effector to the environment. While many of the tasks performed by the robot manipulator are that the robot interacts with its environment such as surgeries, pushing, touching, carrying sensitive objects and cutting, etc. The implementation of all these tasks require that the robot, besides realizing the expected position, provide the necessary force either to overcome resistance from the environment, or comply with the environment. [2]

Constant impedance parameters cannot fulfill requirements in a task where the robot's environment dynamically changes. A robot manipulator increases its impedance when there is an external disturbance and decreases it when the disturbance vanishes to save its control effort [1]. A robot gripper that is catching a flying object needs to be compliant in the direction in which the object is moving but stiff in another direction to hold its position [2]. Therefore, predefining constant impedance parameters is impractical, if not impossible, to achieve the task objectives in impedance control. Researchers have attempted impedance adaptation and impedance learning using various techniques.

There are two main categories of force control schemes: hybrid position-force control and impedance control. However, the former does not take into account the dynamic interaction between the robot's end effector and the environment. In contrast, **impedance control includes regulation and stabilization of robot motion by creating a mathematical relationship between the interaction forces and the reference trajectories.** It involves an energetic pair of a flow and an effort, instead of controlling a single position or a force**. A mass-spring-damper impedance** set is generally used **for safe interaction purposes**. [12].

However, the state of the art of adaptive impedance control relies either on passive algorithms for maintaining stability, reinforcement learning for bias elimination in datasets which suffers from lack of flexibility across large domains of interaction force and intelligence due to its basic Markov Decision Process techniques, or supervised learning for the involvement of a neural network which suffers from high degrees of bias during dataset selection in addition to the obligation of obtaining large datasets for training the neural network.

In this report, we study a new adaptive impedance control approach, called Genetic Algorithms, it combines the intelligence of supervised learning and neural networks without the bias problem and the training ability of reinforcement learning without the local maxima limitations of its intelligence.

In the first chapter, we will discuss traditional impedance control with constant gain controllers and adaptive impedance control. A brief state of the art of adaptive impedance control will be presented as the variety of use cases in the field and around the world.

In the second chapter, we modelize the genetic algorithm controller, we define the relationship between the neural network and the impedance controller's gains, the genetic algorithm is in charge of choosing these gains based on interaction force which is fed to the input neuron.

Finally, a two degree of freedom robotic arm is simulated along with defining its kinematic model, dynamic model and process flowchart, as well as building a real experimental robotic arm with the help of 3D modeling, 3D printing, Arduino and Python.

The ability of the genetic algorithm and the process of its training of the neural network across many generations are both tested, compared and evaluated between the simulation and the real arm experimentation across many different orders of magnitude of external interaction force from an unknown environment to the dynamic model.

1. Chapter 1: Overview

1.1 Introduction:

In this chapter we will be going over what impedance control is and the problems which come with predefined gain impedance control, how adaptive impedance control improves on it and what it entails, the various applications of it in the industry, what adaptive impedance control is, we will cover the state of the art and the latest research titles.

1.2 Overview of Impedance Control:

Impedance control is one of the approaches to dynamic control by taking into account the dynamic interaction between the robot's end effector and environment. It represents this interaction by a mass-spring-damper system as in **Figure 1.1**. Impedance control outperforms its preceding position and torque controllers and includes regulation and stabilization of robot motion with predefined and constant gains tuned specifically for the system in question.

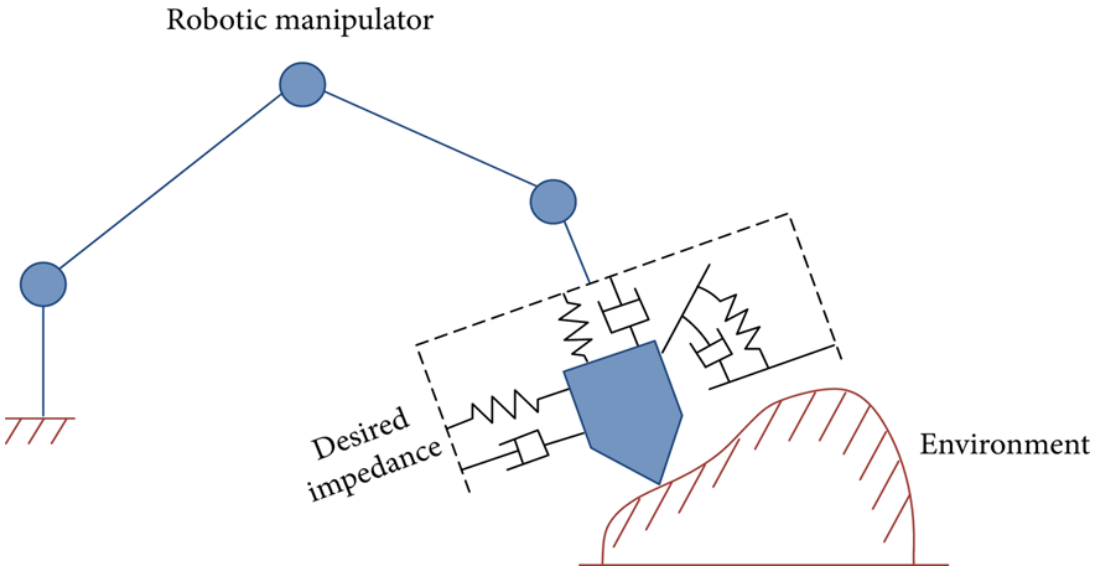

Figure 1.1: Description of impedance control for a robot in contact w/ external environment [12]

As stated, impedance control attempts to make a dynamic relationship between the interaction force and position error by assuming a virtual mass-spring-damper model with the desired trajectory; accordingly, the target impedance function can be expressed as **Figure 1.2**

$$m_d(\ddot{x} - \ddot{x_d}) + b_d(\dot{x} - \dot{x_d}) + k_d(x - x_d) = f_c \quad (1.1)$$

Where f_c is the adaptive impedance controller's output force, x is the current trajectory, x_r is the desired trajectory.

m_d, b_d, and k_d are the desired target impedance coefficients that govern the performance of the controller.

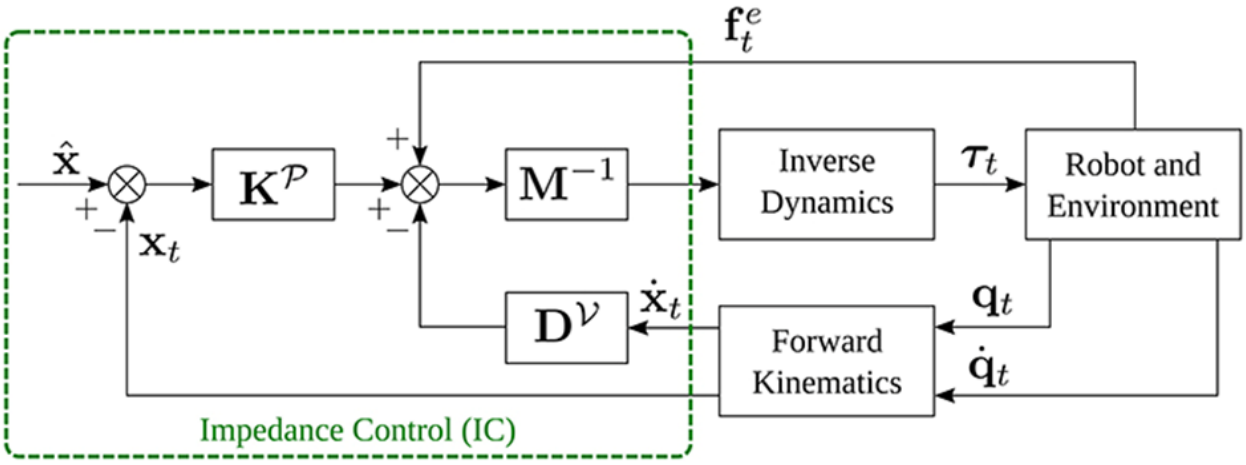

Figure 1.2: Impedance Control System

1.3 Disadvantages of Impedance Control System:

As observed in the aforementioned impedance control equation, the gains of the controller consist of constant and predefined parameters chosen during the design process, this is necessary under consistent circumstances and known/expectable environment conditions. However, it fails drastically when presented with unknown environments with objects which vary massively in stiffness, these unknown environments present the robot with obstacles of all aspects from weak interaction forces to tough ones, this requires that the impedance controller gains to be adaptive based on interaction force.

1.4 Adaptive Impedance Control System:

Adaptive impedance control is the technique of adapting the gains of the impedance control system's gains to the environment, the relationship between the gains and interaction force is largely determined by the engineer or designer of the dynamic system. Adaptive impedance control reduces interaction force massively compared to its predecessor as it directly relates to environment conditions with the impedance controller. Over the decades, researchers have invented several approaches such as Variable Impedance Control (VIC), Variable Impedance Learning (VIL) and Variable Impedance Control (VILC).

1.4.a Variable Impedance Control VIC:

The Variable Impedance Control technique is an efficient, flexible and safe technique in adaptive impedance control, initially proposed by Ikeura and Inooka (1995) [35], it either focuses on stability/passivity or a human in the loop of the system. (**Figure 1.3**)

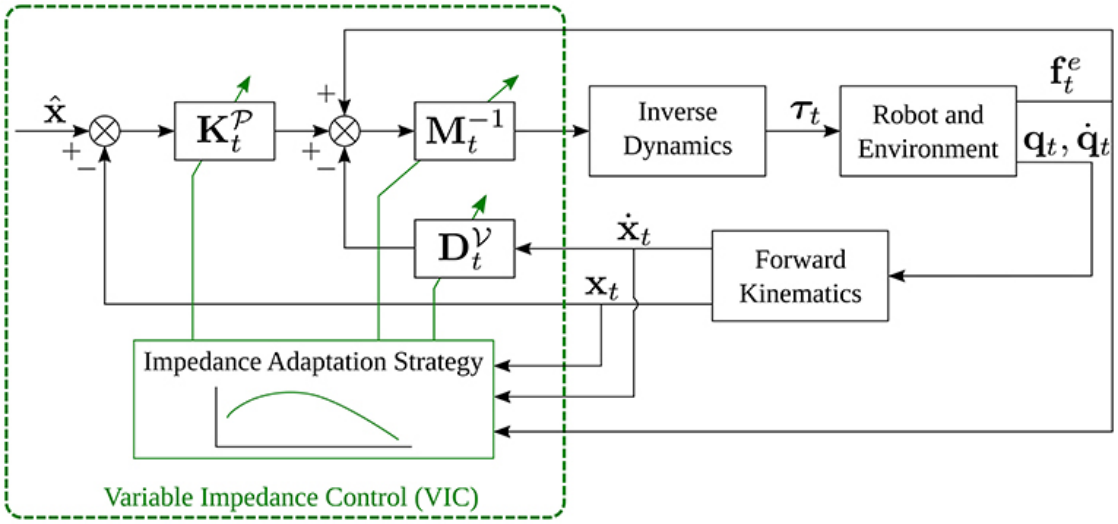

Figure 1.3: VIC Passivity-Based System [11]

1.4.b Variable Impedance Learning VIL: Variable Impedance Learning is based on a cost function which allows the system to learn over time from previously provided data points from desired environments, this allows the system to behave appropriately even with never-seen-before situations. (**Figure 1.4**)

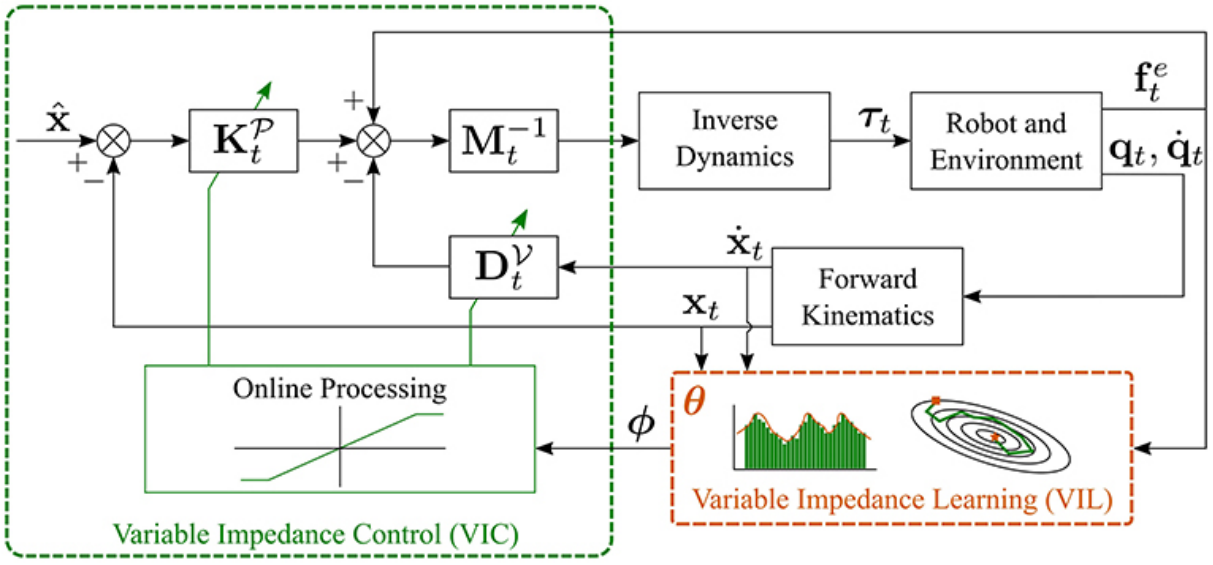

Figure 1.4: VIL System [11]

1.4.c Variable Impedance Learning Control VILC: The key difference between VIL and VILC is that in VILC the data collection process itself depends on the underlying control structure. Therefore, the learning and control block are merged in **Figure 1.5**, while they are separated in **Figure 1.4**. Compared to standard VIC, VILC approaches adopt more complex impedance learning strategies requiring iterative updates and/or robot self-exploration.

Moreover, VILC also updates the target pose (or reference trajectory) while typically relying on constant inertia matrices. (FrontiersIn 2020 report)

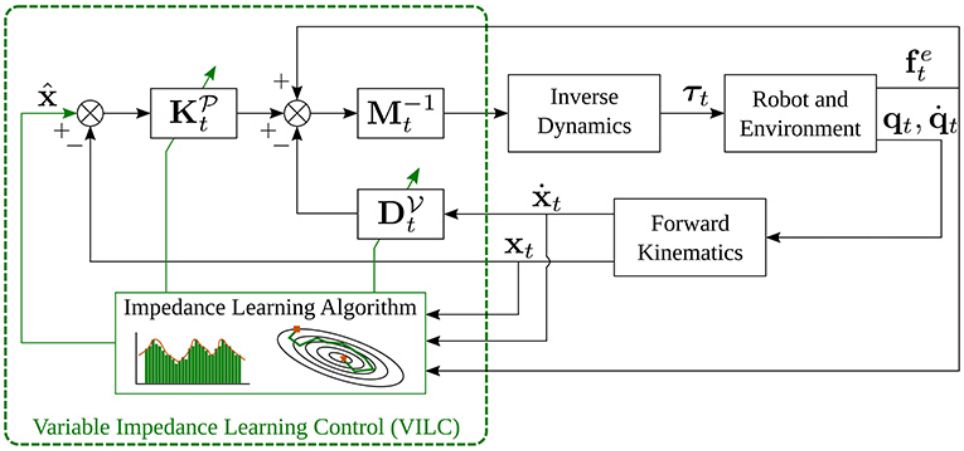

Figure 1.5: VILC System [11]

Each of these approaches being part of the variable impedance control category as seen in **Figure 1.6**.

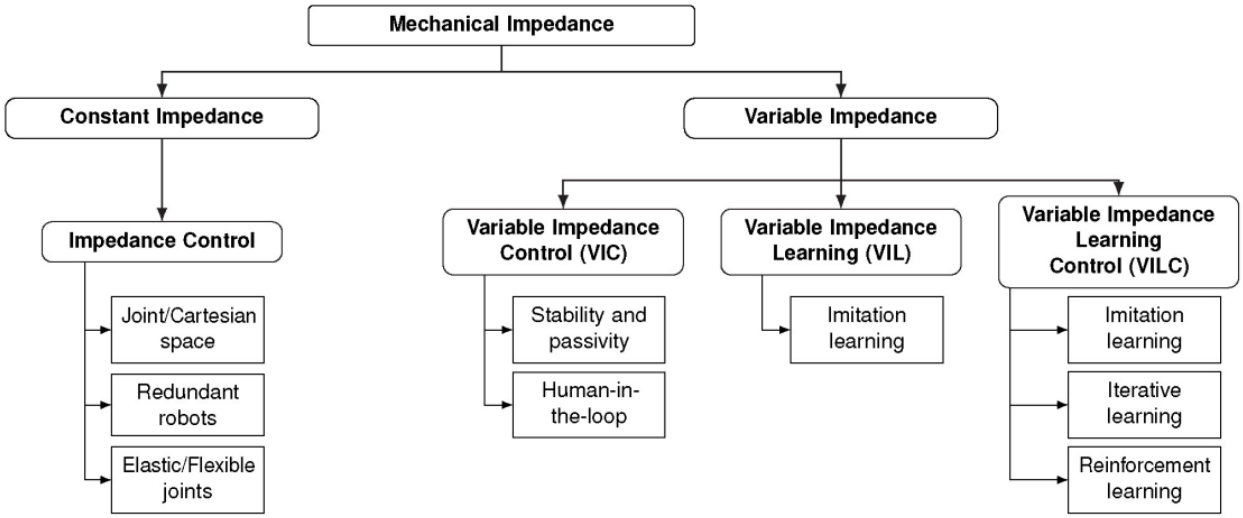

Figure 1.6: Existing approaches for variable impedance control and learning [7]

1.5 Applications of Impedance Control:

Impedance control is essential in a vast range of domains such as:

1.5.a Medical Care: Da Vinci Surgical System

When it comes to robotic surgery, the Da Vinci Surgical System (**Figure 1.7**) is the first which comes to mind. Listed at a price tag of $2,500,000, the Da Vinci robot is an advanced

damped system, it relies heavily on impedance control to interact with the human body. Being actuatable only by a surgeon, this robot takes general orders on where to move and how much pressure to apply, meaning it must adapt to interaction forces automatically. Impedance control enables this surgical system to travel with very little stiffness but very high accuracy as to avoid dangerous interactions which stiff robots suffer from, the Da Vinci system cuts through the skin with very high delicacy and corrects for little consequential mistakes the surgeon commits.

In **Figure 1.8**, an example surgery was performed by the Da Vinci Surgical System on a grapefruit as part of its campaign.

Figure 1.7: Da Vinci Surgical System [13]

Figure 1.8: Da Vinci Surgical System Surgery Test [14]

1.5.b Unknown Environment Inspection: Boston Dynamics Spot, Stretch & Atlas Robots

Boston Dynamics is a 30 year old robotics company, they manufacture smart industrial robots such as Spot (**Figure 1.9**), Stretch (**Figure 1.10**) and Atlas (**Figure 1.11**), these robots must interact with unpredictable environments at all times where low stiffness and delicacy is a must.

Figure 1.9: Spot - Boston Dynamics [15]

Figure 1.10: Stretch - Boston Dynamics [16]

Figure 1.11: Atlas - Boston Dynamics [17]

1.5.c Industrial Settings & Factories: Tesla Factory

The Tesla factories (**Figure 1.12**) are highly automated car manufacturing facilities. Equipped with hundreds of industrial grade robotic arms, impedance control is a must. These robotic arms come in contact with human workers often, they work collaboratively with humans to achieve required tasks. These robot arms are equipped with non stiff and damped systems, otherwise known as impedance control systems.

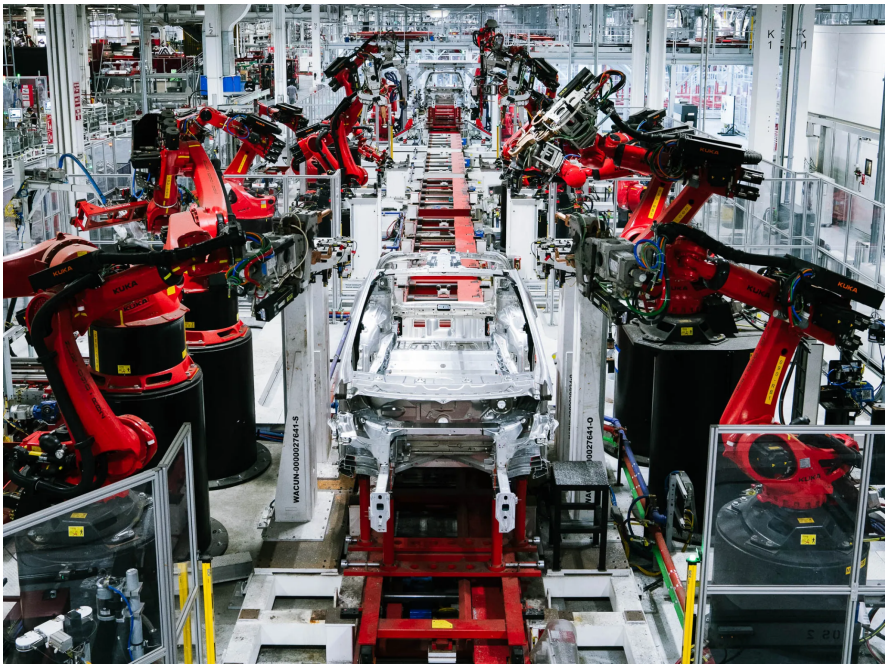

Figure 1.12: Tesla Factory [18]

1.6 Conclusion:

As observed, impedance Control turns any robotic system into a damped non stiff system, hence reducing interaction force with humans and the environment in medical care facilities, industrial settings, and inspection. However, constant parameter impedance does not perform outside of predefined and tested environments, Adaptive Impedance Control is a variable parameter strategy, it related the interaction force with the gains of the impedance control system, this relation is chosen by the developer of the system, throughout the decades this relationship has developed from directly proportional relationships, with the goal of improving stability and passivity **VIC**. Later on, **VIL** for learning strategies were introduced by researchers in order to teach the model impedance control. Finally, Adaptive Impedance Control has moved onto relying on neural networks through **VILC**.

2. Chapter 2: Modeling

2.1 Introduction:

In this chapter, an improved solution to VILC is introduced, the supervised model-based training of variable impedance learning control algorithms strictly rely either on dataset based training or reinforcement learning based training. A merge of the two techniques would result in an intelligent and flexible system. In this chapter, we discuss Genetic Algorithms and neural networks and their implementation and role in adaptive impedance control (VILC).

2.2 Problem Description:

As has been observed, constant parameter impedance control was improved upon with the introduction of variable impedance control, generally implemented in one of the following strategies:

1. VIC (Stability & Passivity, Human-In-The-Loop, ect..)
2. VIL or VILC (Imitation Learning (Supervised Learning), Iterative Learning (Reinforcement Learning)).

The VIC strategy is viable when the environment is encapsulated and fully known with no chance of never-seen-before situations, the learning strategies consist of two methods: **Supervised Learning** which require lots of training data which must be collected and provided to the system's learning algorithm by the designer of the system, meaning the training data is selected by a human who's prone to biases, unbalancing the data and missing many real cases. **Reinforcement Learning** which does not require hand picked training data, it allows the control system to iteratively learn on its own by exploring the unknown environment and learn from previous mistakes based on the policy function. However, Iterative Learning with Reinforcement Learning algorithms such as the Markov Decision Process is difficult to tune in the right direction (requires a human in the training decision process). It does not rely on a neural network which actively learns autonomously, it is very impractical in non-encapsulated environments and it does not generalize based on specific test cases, but instead keeps in mind the proper decisions to take under specific situations. [34]

2.3 Objective:

In this project, the aforementioned Reinforcement learning and supervised learning techniques of training an intelligent computer are combined resulting in a genetic algorithm which consists of reinforcement learning training techniques and supervised learning learning techniques involving a set of neural networks.

The objective of this merge is essentially providing a brain to a reinforcement learning system instead of a generic Markov Decision Process and eliminating the human from the training operation.

2.4 Dynamic Model:

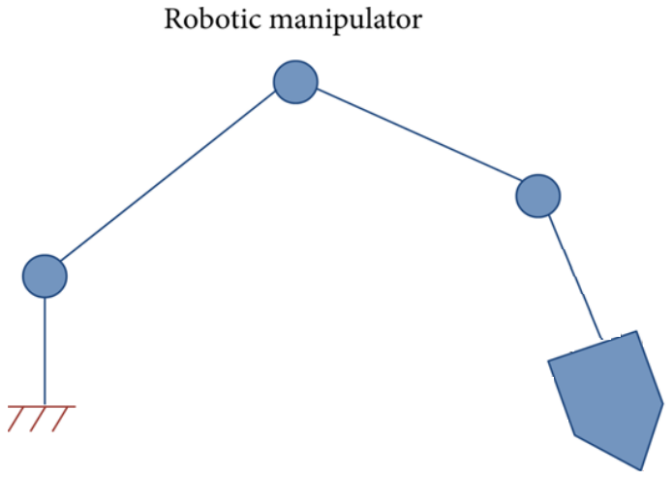

Figure 2.1: Robotic Manipulator [12]

The dynamic model of the system shown in **Figure (2.1)** consists of f_e which is the torques applied by the external environment, f_c being the impedance controller's torque, J^T is the jacobian of the system in transpose state, $G(\theta)$ represents the gravity forces affecting the dynamic system, $B(\dot{\theta},\theta)$ is the centrifugal and coriolis forces experienced, and $M(\theta)$ is the inertia matrix.

$$M(\theta)\ddot{\theta} + B(\dot{\theta},\theta) + G(\theta) = J^T f_c + f_e \quad (2.1)$$

This dynamic model is often obtained using the Lagrangian method where the potential and kinetic energies of the system are subtracted and differentiated based on the Euler method as observed in equations (2.2), (2.3) and (2.4).

$$L = K - U \quad (2.2)$$

$$E_1 = \frac{dL}{\theta_1} - \frac{dL}{d\dot{\theta_1}} \quad (2.3)$$

$$E_2 = \frac{dL}{\theta_2} - \frac{dL}{d\dot{\theta_2}} \quad (2.4)$$

The dynamic model equation (2.1) is a refactorization of equations (2.3) and (2.4). The final step to obtaining the differential equations of this model, the angular acceleration is factored out from the equation (2.1) as seen in equation (2.5)

$$\ddot{\theta} = M^{-1}(\theta)(-B(\dot{\theta},\theta) - G(\theta) + J^T f_c + f_e) \quad (2.5)$$

Due to the dynamic system potentially experiencing gravity forces, a gravity compensation must be added with the impedance controller's output decision value, the gravity compensation T is simply obtained via the equation (2. 6)

$$T = M(\theta)\ddot{\theta} + B(\dot{\theta}, \theta) + G(\theta) - f_e \quad (2.6)$$

Turning the factorized dynamic model (2. 5) into (2. 7)

$$\ddot{\theta} = M^{-1}(\theta)(-B(\dot{\theta}, \theta) - G(\theta) + T + J^T f_c + f_e) \quad (2.7)$$

In order to introduce this impedance controller, it is first converted into a torque by multiplication with J^T then introduced to the dynamic model of the robot as an internal torque as observed in equation (2. 1)

2.5 Proposed Controller:

The interfacing between the genetic algorithm and variable impedance controller is achieved through tuning the gains of the impedance controller in real time depending on current interaction force and previous experience with the environment, the output layer of the neural network consists of two neurons λ_1 and λ_2, each neuron depicts the delta applied to its respective gain in the impedance controller.

Where f_c is the adaptive impedance controller's output force, x is the current trajectory, x_r is the desired trajectory.

m_d, b_d, and k_d are the desired target impedance coefficients that govern the performance of the controller.

$$m_d(\ddot{x} - \ddot{x_d}) + b_d(\dot{x} - \dot{x_d}) + k_d(x - x_d) = f_c \quad (2.8)$$

Where $$k_d = k_{cd} + \lambda_1 \quad (2.9)$$

and $$b_d = b_{cd} + \lambda_2 \quad (2.10)$$

with k_{cd} and b_{cd} and predefined constants.

The acceleration factor of the impedance controller is negligible in our case as speeds are depicted constant at all times resulting in the following equation

$$b_d(\dot{x} - \dot{x_d}) + k_d(x - x_d) = f_c \quad (2.11)$$

Where $$\dot{x} = J\dot{\theta} \quad (2.12)$$

Insinuating that the impedance controller used is based on cartesian data instead of torques.

2.6 Genetic Algorithm:

Genetic algorithm training is the process of generating a batch of several neural networks and actively training them in real time on a real/realistic environment, the training process is done through generations and the best performing neural networks are mated and combined at the end of each generation, the succeeding version of the neural networks is then duplicated and exposed in a new generation to the unknown environment where the same process is repeated for several generations, essentially improving the neural network repeatedly. Genetic algorithms employ many different strategies and methods but the one studied in this report is N.E.A.T (Neuro-Evolution of Augmented Topologies) as per **Figure 2.2.**

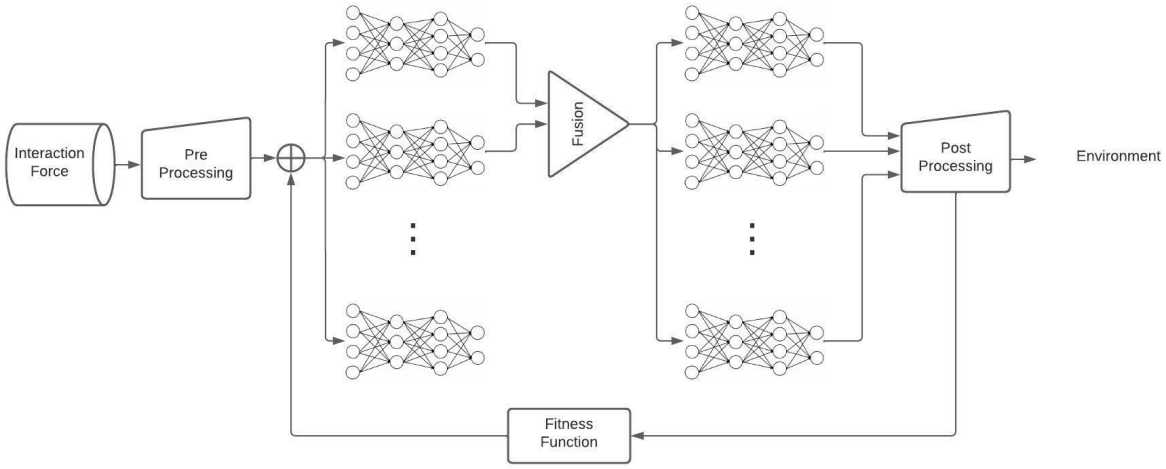

Figure 2.2: Genetic Algorithm Scheme

2.4.a Neuro-Evolution of Augmented Topologies:

N.E.A.T is a genetic algorithmic training method developed by Kenneth O. Stanley [33] which duplicates a neural network into many genomes and essentially allows them to compete against each other in the environment in a time frame separated into generations.

N.E.A.T combines the flexibility and intelligence of supervised learning (neural network flexibility) and the real time training aspect of reinforcement learning resulting in a more intelligent and more efficient adaptive impedance control system solution. In order to train a neural network to predict two gains of the impedance controller, a multi class output layer is required. Furthermore, a Sigmoid output activation function is obsolete as it is designed for binary output.

TanH, reLU and eLU are multi class activation functions seen in **Figure 2.3** decide the behavior of the output neuron of the neural network

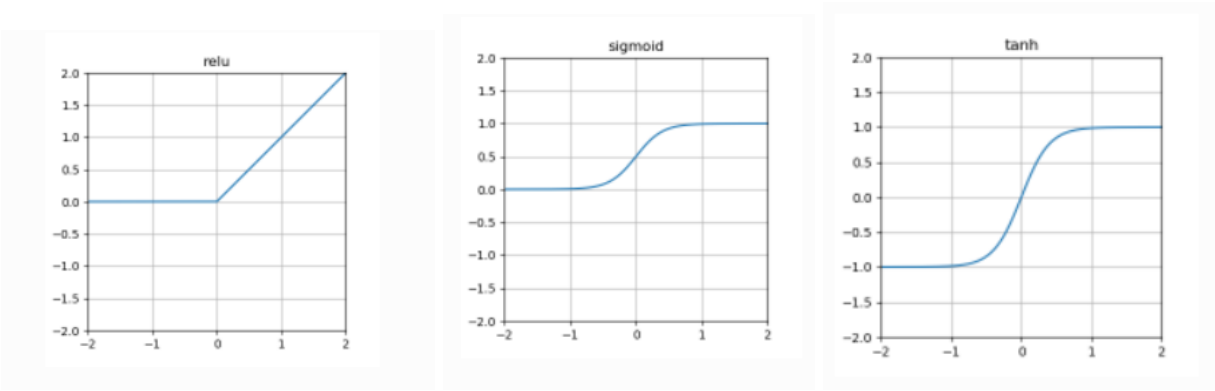

Figure 2.3: Common Machine Learning Cost Functions [23]

reLU is a linear purely positive activation function, this activation function suits impedance control gain prediction as it is positive ended and does not offer negative predictions as represented in equation (2.13) where x is a real number supplied by a neuron in an internal layer of the neural network.

$$f(x) = max(0, x) \quad where \quad x \in \Re \qquad (2.13)$$

The supplied input to the neural network is the external environment interaction force f_e, it is then multiplied through a dot product with the weights of each neuron throughout the layers of the deep convolutional neural network until the output layer is reached, summing the result of each layer as seen in **Figure 2.4** where $a^{(n)}$ is the output result of dot product between the previous layer's input $a^{(n-1)}$ and the layer's weight values $w^{(n-1)}$ and h is the applied activation function to a before supplying it to the subsequent layer which is called forward propagation for an output.

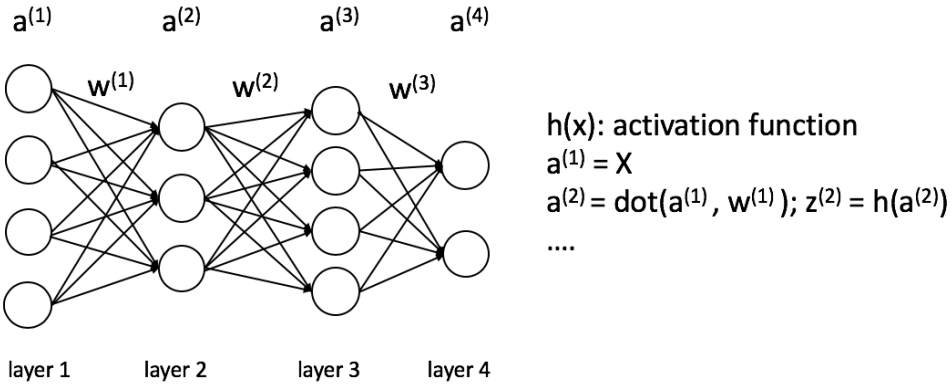

Figure 2.4: Deep Convolutional Neural Network [32]

This summation of dot products is called convolution which is why it's called a convolutional neural network, this turns the relations from **Figure 2.4** into **Figure 2.5**.

$$a^{(1)} = X$$
$$a^{(2)} = h(z^{(2)}), z^{(2)} = a^{(1)} * w^{(1)}$$
$$a^{(3)} = h(z^{(3)}), z^{(3)} = a^{(2)} * w^{(2)}$$
$$a^{(4)} = h(z^{(4)}), z^{(4)} = a^{(3)} * w^{(3)}$$

Figure 2.5: Layer outputs [32]

The genetic algorithm is inserted into the dynamic system with the impedance controller in the manner shown in **Figure 2.6**.

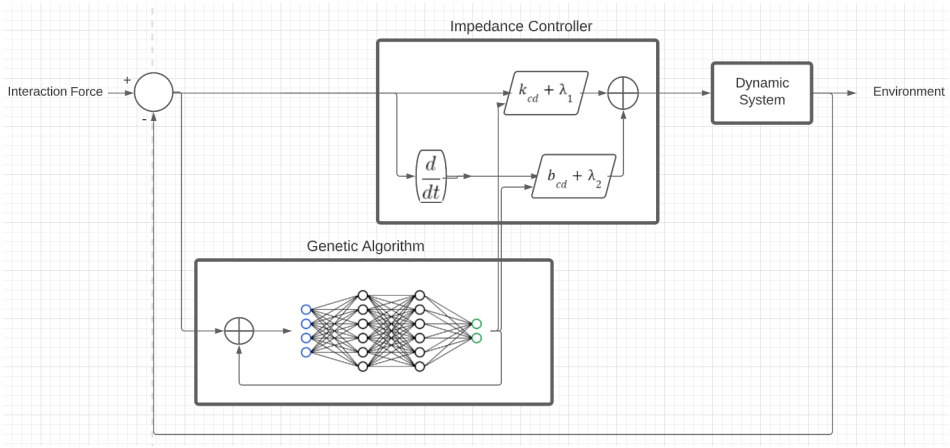

Figure 2.6: Training System Loop Scheme

2.7 Conclusion:

Impedance control is an approach to safety by reducing stiffness on dynamic systems, traditional impedance control relies on predefined gains to achieve its decisions, it is only in use in highly controlled environments while variable impedance control relates the gains of the control system with interaction force further reducing interaction force by introducing flexibility.

Genetic algorithm (N.E.A.T) is an intelligent neural network training algorithm which further reduces interaction force in a more efficient way compared to VIC, VIL and VILC techniques available presently.

3. Chapter 3: Simulation & Results

3.1 Introduction:

In this chapter we simulate an impedance controlled 2 Degree Of Freedom robotic arm in Python and Matlab, the simulation abides by realistic restraints by defining the dynamic and kinematic system of the robotic arm. The differential equations are derived and the neural network's output neurons are interfaced with the impedance controller's gains included in the dynamic system.

3.2 Kinematic Model:

The kinematics of the robotic arm are the same to that by Tuna Orhanli where the first link tilts by θ_1 from the horizontal axis with length $L_1 = 0.3m$ and mass $M_1 = 0.5kg$, and the second link tilts by θ_2 beginning from θ_1 of the first link with length $L_2 = 0.3m$ and mass $M_2 = 0.5kg$, the gravity is chosen to be $g = 9.8m/s^2$ as observed in **Figure 3.1**.

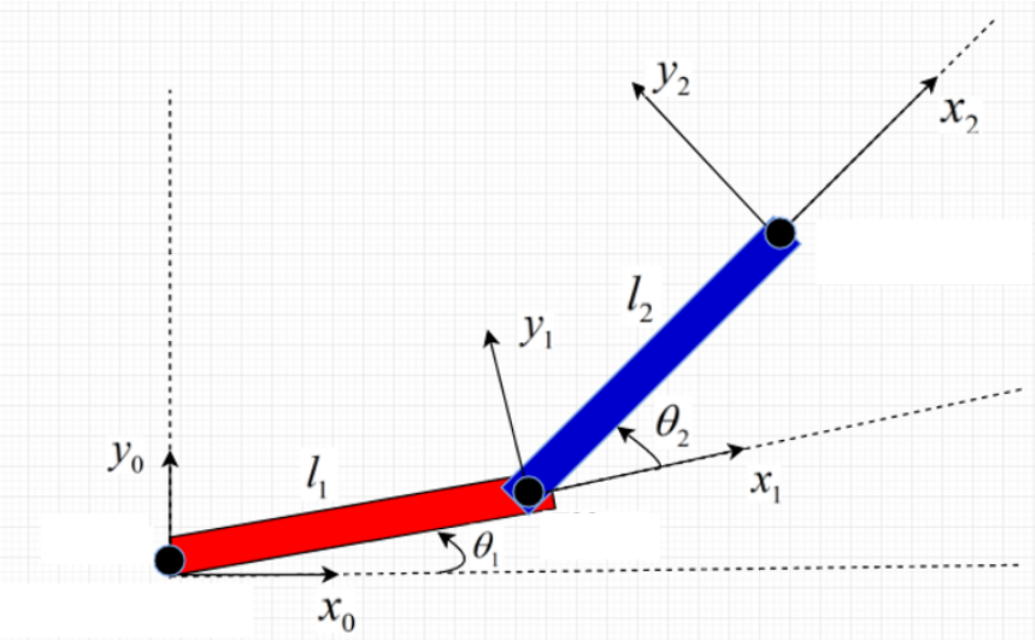

Figure 3.1: Kinematic System of 2 DOF Robotic Arm [22]

The forward kinematic of point $P_1\ (x_1, y_1)$ is a Pythagorean relation between L_1 and θ_1 resulting as in equations (3.1) and (3.2):

$$x_1 = L_1 cos(\theta_1) \qquad (3.1)$$

$$y_1 = L_1 sin(\theta_1) \quad (3.2)$$

Furthermore, the kinematics of point P (x_2, y_2) being the end effector follows the same method in equations (3.3) and (3.4)

$$x_2 = L_1 cos(\theta_1) + L_2 cos(\theta_1 + \theta_2) \quad (3.3)$$

$$y_2 = L_1 sin(\theta_1) + L_2 sin(\theta_1 + \theta_2) \quad (3.4)$$

The forward kinematics allow for converting angular displacement into cartesian displacement, the jacobian of the velocity is also required in simulation to convert cartesian forces into torques, the cartesian velocities V_{x_1}, V_{y_1}, V_{x_2} and V_{y_2} are obtained by differentiating the relationships in equations (3.1), (3.2), (3.3) and (3.4) as follows:

$$V_{x1} = -L_1\dot{\theta}_1 sin(\theta_1) \quad (3.5)$$

$$V_{y1} = L_1\dot{\theta}_1 cos(\theta_1) \quad (3.6)$$

$$V_{x2} = -L_1\dot{\theta}_1 sin(\theta_1) - L_2 sin(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2}) \quad (3.7)$$

$$V_{y2} = L_1\dot{\theta}_1 cos(\theta_1) + L_2 cos(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2}) \quad (3.8)$$

In order to obtain the 2x2 jacobian matrix we represent the velocity as a 3x1 matrix with the third row below a zero vector as follows:

$$V = [-L_1\dot{\theta}_1 sin(\theta_1) - L_2 sin(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2}),$$

$$L_1\dot{\theta}_1 cos(\theta_1) + L_2 cos(\theta_1 + \theta_2)(\dot{\theta_1} + \dot{\theta_2}),$$

$$0]; \quad (3.9)$$

The relationship between the angular velocities and V is derived by factoring the angular velocities $\dot{\theta}_1$ and $\dot{\theta}_2$ in equation (3.9) into a 2x1 vector as follows in equation (3.10):

$$V = [-L_1 sin(\theta_1) - L_2 sin(\theta_1 + \theta_2), -L_2 sin(\theta_1 + \theta_2),$$

$$L_1 cos(\theta_1) + L_2 cos(\theta_1 + \theta_2), \quad L_2 cos(\theta_1 + \theta_2), \quad [\dot{\theta}_1, \dot{\theta}_2]$$

$$0, \quad 0 \quad] \quad (3.10)$$

Where the jacobian matrix is as follows in equation (3.11) (the third row is added for calculation simplification):

$$J = [-L_1 sin(\theta_1) - L_2 sin(\theta_1 + \theta_2), \quad -L_2 sin(\theta_1 + \theta_2),$$
$$L_1 cos(\theta_1) + L_2 cos(\theta_1 + \theta_2), \quad L_2 cos(\theta_1 + \theta_2),$$
$$0, \quad 0 \quad] \quad (3.11)$$

In order to convert cartesian forces back to torques, all that is required is to multiply the transpose of the jacobian matrix by the cartesian forces.

3.3 Dynamic Model:

The dynamic system of a physical object is its mathematical representation within a manageable equation, this equation is the basis of the simulation of this robotic arm. The impedance control system interferes with the dynamic system as an internal torque originally in cartesian and later converted into torques before being supplied to the differential equation of the system.

Where M is the 2x2 inertia matrix in equation (3.16):

$$M = [(M_1 + M_2)L_1^2 + M_2 L_2^2 + 2M_2 L_1 L_2 cos(\theta_2), \quad M_2 L_2^2 + M_2 L_1 L_2 cos(\theta_2)$$
$$M_2 L_2^2 + M_2 L_1 L_2 cos(\theta_2), \quad M_2 L_2^2 \quad]; \quad (3.16)$$

And $B(\theta, \dot{\theta})$ is the coriolis and centrifugal forces 1x2 matrix :

$$B(\theta, \dot{\theta}) = [-M_2 L_1 L_2 (2\dot{\theta_1}\dot{\theta_2} + \dot{\theta_2}^2) sin(\theta_2);$$
$$-M_2 L_1 L_2 \dot{\theta_1}\dot{\theta_2} sin(\theta_2)] \quad (3.17)$$

And $G(\theta)$ is the gravitational forces 1x2 matrix:

$$G(\theta) = [- (M_1 + M_2) g L_1 sin(\theta_1) - M_2 g L_2 sin(\theta_1 + \theta_2) ;$$
$$- M_2 g L_2 sin(\theta_1 + \theta_2) \quad] \quad (3.18)$$

The interaction force matrix f_e of shape 2x1 and the converted impedance control equation from cartesian into torques are introduced into the dynamic system's equation on the right hand side (the impedance control system is in cartesian for this simulation).

$$f_e = [f_x; f_y] \quad (3.19)$$

Turning the dynamic system's equation into the following equation (3.20):

The impedance control's torque is considered an internal torque while the interaction force f_e is an external torque. In order to obtain the differential equations of the dynamic system, equation (3.21) is solved by multiplying both sides by the inverse of the inertia matrix M:

$$[\ddot{\theta}_1, \ddot{\theta}_2] = M^{-1}(\theta)(-B(\dot{\theta}, \theta) - G(\theta) + J^T f_c + f_e) \quad (3.21)$$

Since the robotic arm experiences gravity, a compensation for this gravity effect must be factored into the internal torques within the dynamic system's equation. We first must derive the equations for the internal torques to determine the gravity compensation.

This gravity compensation's torque (2.13) must be combined with the impedance control's torques seen in equation (2.1) before supplying the differential equations for an output.

3.4 Genetic Algorithm:

Genetic Algorithms are the state of the art in terms of machine learning model training, it combines the real time aspect of reinforcement learning with automatically designed and tuned neural networks. The neural networks are trained over what is known as generations.

At the start of a generation, a batch of neural networks is generated by the genetic algorithm, these neural networks are different in architecture, weights and behavior. The neural networks are then exposed to the desired environment and are given access to inputs from the environment as well as access for decision making. At this stage of training, the designer must write a policy function, a policy function is a set of rules which dictate what is essentially desirable and undesirable, the neural networks are then graded with a score called fitness based on its performance, the non-compliant neural networks which have low fitness are then eliminated during the generation.

At the end of a generation, the surviving neural networks are then mutated and combined in order to generate a new batch of neural networks usable in the next generation.

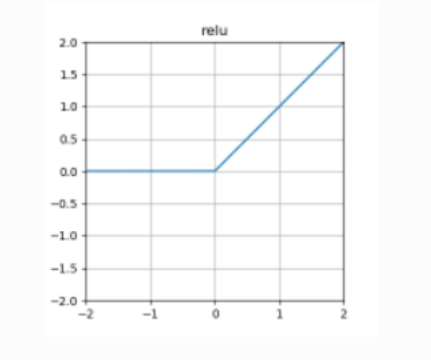

Figure 3.2: reLU Activation Function [23]

3.5 Interfacing:

The simulation is a closed loop system with a feedback of interaction force fed constantly back into the genetic algorithm and impedance controller, essentially training the neural network in real time with real data from its interaction with the unknown environment as observed in **Figure 3.3.**

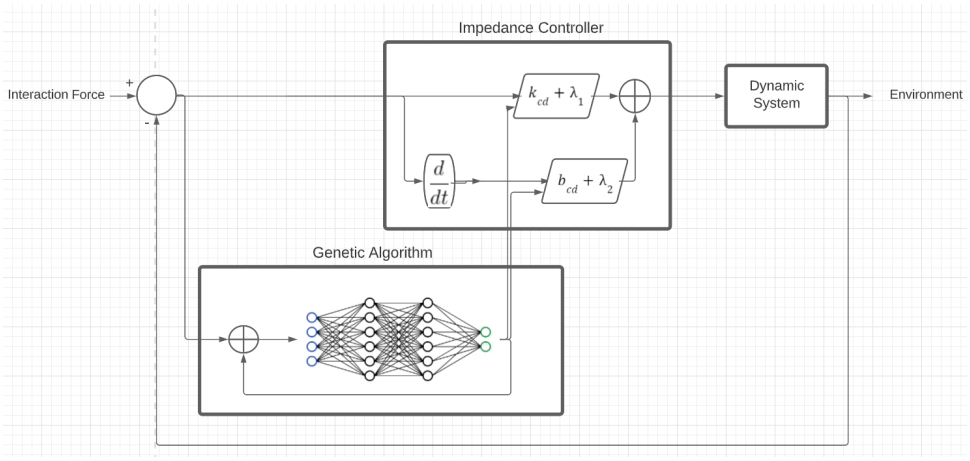

Figure 3.3: Training System Loop Scheme

The gains are updated based on decisions from the neural networks as they are actively tuned by the genetic algorithm, the output of the neural network is the offset applied to the gain.

3.6 Simulation Results:

The purpose is to test the response of the arm to unexpected objects within an unknown environment, the arm must react with the least interaction force conceivable in intelligent and dynamic manners.

First, the training portion of the jupyter notebook is executed where the neural networks are run through multiple generations of experimental interaction force f_e from the environment. These experiments consist of the following three test cases: $f_e = 50N$, $f_e = 5N$ and $f_e = 0.5N$

3.6.a First case: ($f_e = 50N$) The first test case is applied to a batch of 50 genomes over 100 generations. As the neural networks are exposed to an initial interaction force from the three experimental values mentioned, the neural networks decide the values of the gains of the impedance controller according to the underlying interaction force.

As observed in **Figure 3.4**, in the middle of the first window are 50 robotic arms training simultaneously, each one of these robot arms is controlled by a neural network which in turn has a fitness reward/punishment value, this fitness value is governed by the fitness function monitoring the neural network's behavior.

In the left hand side of the screen appear the current generation, the constantly varying gains of the impedance control, the peak velocity of the best performing genome and finally the current time out of the maximum allowed time for convergence (5 seconds this test).

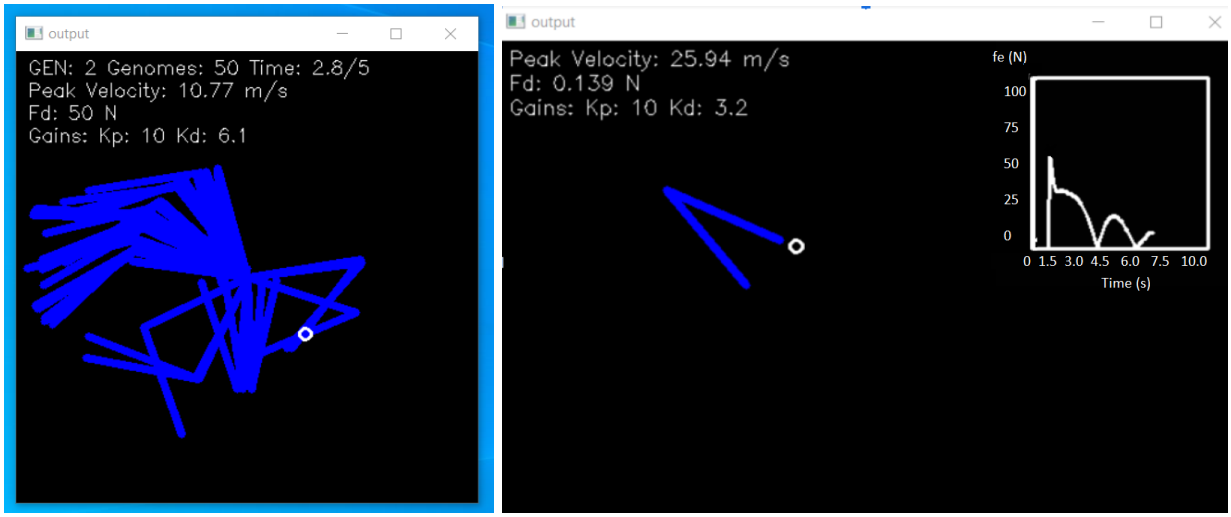

Figure 3.4: Test case 1 2nd generation training & transient response

In the right hand side is the last living genome of the second generation, and next to it is the interaction force graph over time, at time t0, the initial interaction force of 50 N is introduced. As seen in the graph, the system is highly oscillatory with a transient phase of 3.9s and settling time of 6.2s (including transient phase) due to an aggressive neural network, fast forwarding to the 50th generation where the neural network has learned several orders of magnitude more information about the neural network we notice the oscillations have been completely eliminated as seen **Figure 3.5.**

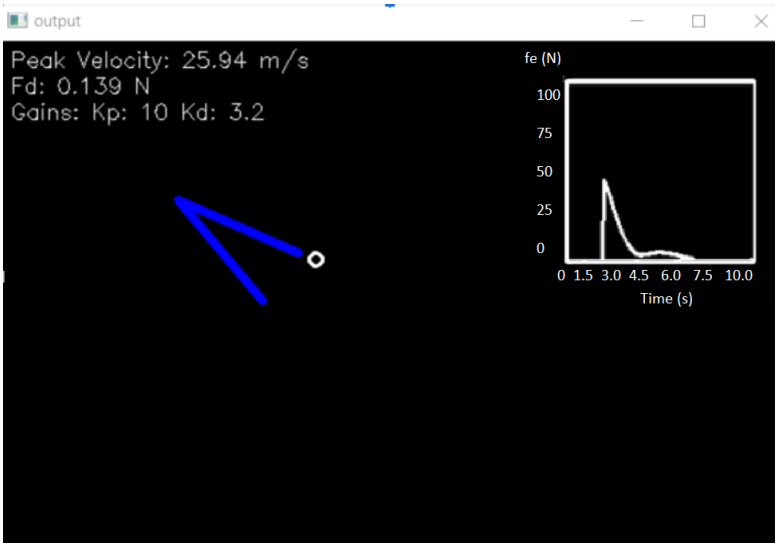

Figure 3.5: Test case 1's last surviving genome 50 generations of training

The interaction force decays rapidly within 2.5s with a total settling time of 3.5s,the neural network is constantly varying the impedance controller's gains to maintain the lowest interaction force possible at any given time based on the previously sampled values from the simulated sensors.

Convergence is the result of the neural network having learned what is necessary to perform interaction force reduction to a null steady state as seen in **Figure 3.6**.

It is also noticed that the proportional gain is set by the neural network way lower than typical only when there is no interaction force in sight (empty space), this strategy was developed by the neural network which consists of keeping stiffness very low in anticipation for sudden external interaction forces.

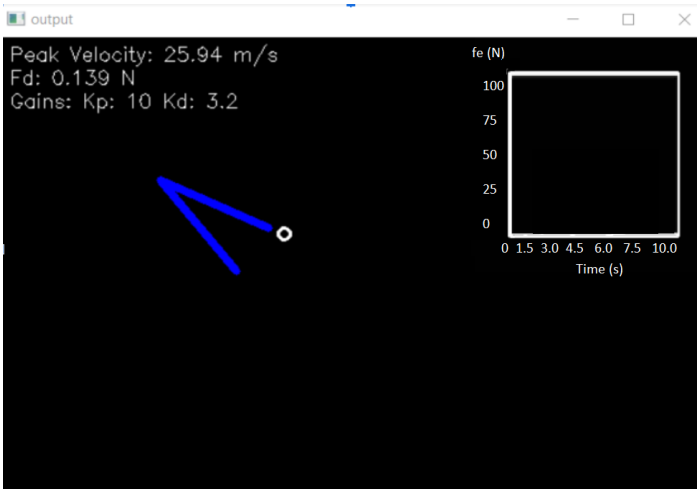

Figure 3.6: Test case 1 steady state

3.6.b Second case: ($f_e = 5N$) The second test case is applied in a similar manner as observed in **Figure 3.7**, the transient phase is very similar to that of the first case concluding that the neural network is able to handle interaction forces of different levels and is able to converge to a steady state with a transient phase of only 2.3s and a total settling time of 3.1s which is to be expected as the second case's initial interaction force is an order of magnitude smaller than the first case and so the settling time is considerably lower.

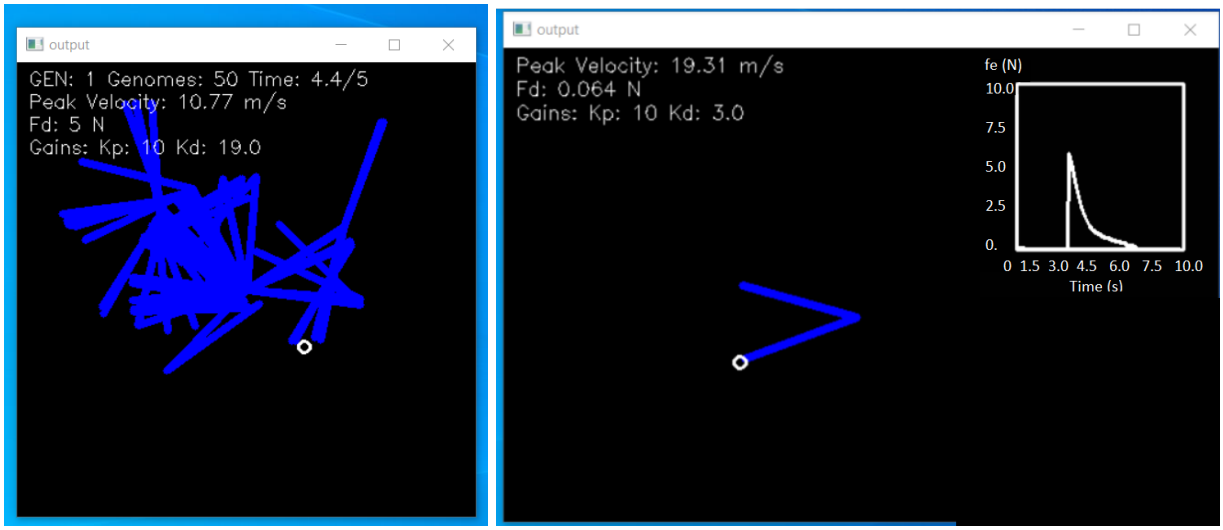

Figure 3.7: Test case 2 training & transient response

3.6.c Third case: ($f_e = 0.5N$) The third test case is also applied in a similar manner as observed in **Figure 3.8**, the transient phase is also very similar to that of the first case also concluding that the neural network is able to handle interaction forces of different levels and is able to converge to a steady state with a transient phase time of 2.1s and settling time of 2.9s.

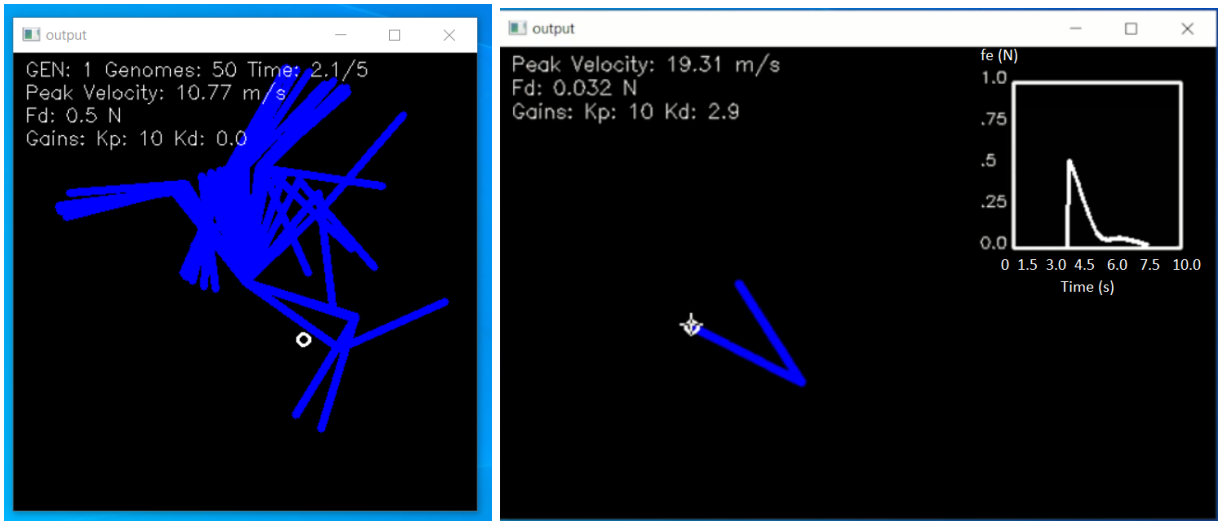

Figure 3.8: Test case 3 training & transient response

The three performed simulated values show the following results:

Force Introduced f_e	Transient Phase (s)	Settling Time (s)
0.5 N (0.05kg)	2.1	2.9
5 N (0.5kg)	2.3	3.1
50 N (5.1kg)	2.5	3.5

3.7 Conclusion:

- The transient phase of the response is considerably smaller at rest.
- Dynamic system simulation is a lengthy process involving realistic period selection and differential equation execution.
- Using Python and Jupyter Notebook, the simulation and the graphical interface were developed which aided in analyzing the interaction force based on time change which is essential for an evaluation.
- A genetically trained neural network through policy functions can adapt to multiple different orders of magnitude of interaction forces. The arm was exposed to 0.5N , 5N and 50N of applied force and it performed adequately.
- Simulating kinematic robots with complex controllers helps immensely in the development process of applied robotic algorithms and training neural networks in a short period of time and low cost of R&D (research & development).
- Simulating real objects using dynamic systems allows for grasping the realistic results. Once the simulation is proven to perform, a real implementation can be confidently implemented.

4. Chapter 4: Experimentation & Results

4.1 Introduction:

In chapter four, we will build a physical 2 DOF robotic arm which matches the simulated geometrics. However, a simulated robotic arm can have infinite force/pressure points which sense interaction force without extra monetary cost, real force/pressure sensors have hefty price tags and due to that we will be installing at most 4 sensors positioned around the end effector. Ideally, the entire body of the robotic arm must be able to sense applied force/pressure.

4.2 Materials & Parts:

For an experimental & educational arm, a great size is desired for robustness, usability and clearity, an approximate desired size of 40x40cm is ideal and maintainable with relatively affordable structures and actuators.

4.2.a Servo Motors:

For a robust 2 DOF robotic arm, the smaller SG90 and MG90s servos seen in **Figures 4.1 and 4.2** only offer a maximum of 2.5kg/cm torque which snaps with little applied force in any direction especially for the SG90 variant with plastic internals.

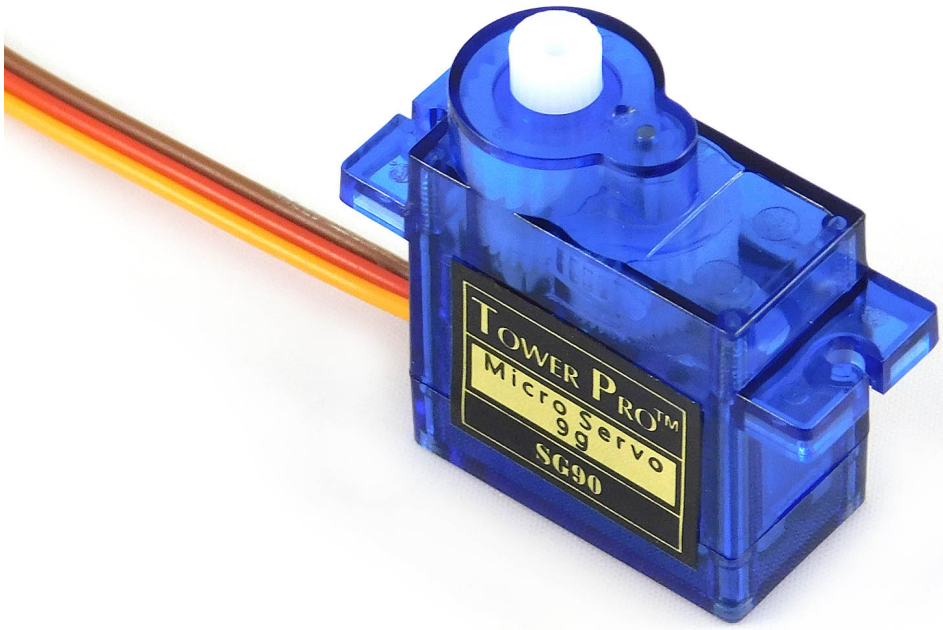

Figure 4.1: SG90 Servo Motor [24]

Figure 4.2: MG90S Servo Motor [25]

The MG996R robust servo motors observed in **Figure 4.3** offer 11kg/cm in a manageable form factor and volume, its strong and metal internals can withstand human handling and sudden external forces/pressures.

Figure 4.3: MG996R Servo Motor [26]

4.2.b Force Sensors:

There is a variety of methods of sensing applied force, ranging from physical touch resistive sensors to motor shaft torque sensors, torque sensors come with hefty price tags for the cheapest ones, sensor observed in **Figure 4.4** senses up to only 10NM with a price tag of 825$ for one single torque sensor, the cheapest variant seen in **Figure 4.5** is still at a minimum of 200$ for a single unit.

Figure 4.4: Torque Sensor [27]

Figure 4.5: Cheaper Torque Sensor [28]

With the high pricing of torque sensors, we resorted to resistive physical pressure sensors which sense applied surface force. However, resorting to these sensors requires that the full body of the robot is wrapped with these sensors and due to their unforgivable prices we are resorting to installing four sensors around the end effector for demonstration and educational purposes.

The sensors are positioned around the end effector as it's the easiest to demonstrate behaviors in experiments.

For this test case, we are using the FSR402 flex pressure sensor observed in **Figure 4.6**, this sensor offers a range of measurable forces from a minimum of 1N up to a maximum of 100N. Its minimum required physical strain of 1N will interfere with the responsivity and sensitivity of the impedance controller but being the only affordable sensor available we settled with it.

This pressure sensor is of the resistive type, its resistance value varies based on the applied force from very high resistance to very low resistance. A voltage divider between it and a regular 5k resistor is enough for an arduino's 10 bit ADC input to read its voltage value according to a 10 bit range between 0 and 1023.

Figure 4.6: FSR402 Flex Pressure Sensor [29]

4.2.c Microcontroller:

An arduino is a practical microcontroller for demonstrative and educative operations, it wraps the atmega328p chip with 5v system voltage level, the arduino nano variant seen in **Figure 4.7** is used in this experiment.

Figure 4.7: Arduino Nano Microcontroller [30]

4.2.d 3D Printer:

In order to build a complex body for the robotic arm within limited time, a 3D printer is required, in our case we are using multiple units of the Ender 3 Pro V2 commercial 3D printer manufactured by Creality seen in **Figure 4.8**.

The print parameters for a robust semi-heavy duty body are as follow:

- **Layer Height:** 0.2mm ; 0.24mm initial layer
- **Nozzle Width:** 0.4mm
- **Infill Percentage of Plastic:** 50% (essential for robustness)
- **Infill Pattern of Plastic:** Cubic Pattern (essential for robustness)
- **Print speed:** 90mm/s ; 15mm/s initial layer
- **Nozzle Temperature:** 215°C (robust parts require higher temperature)
- **Bed Temperature:** 60°C
- **Support Structure:** Yes
- **Rafts:** No
- **Total Print Time:** 84 hours

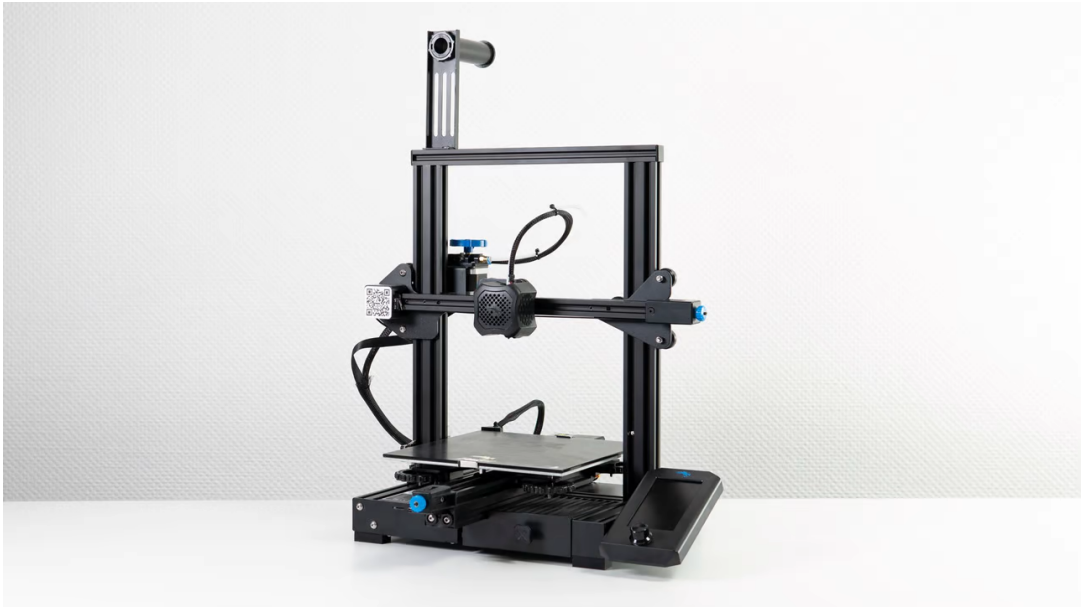

Figure 4.8: Creality Ender 3 Pro V2 [31]

4.3 Circuit Diagram:

The mentioned materials are mounted in the following configuration seen in **Figure 4.9**, the MG996R servo motors are both powered through external power sources with the required 2A of minimum current supply available at 4.8v.

The circuit was mounted on an experimental breadboard to encourage modification and improvement of the system, the resistors were picked to be $5k\Omega$ in order to reduce the current draw by the sensors as they do not require high current.

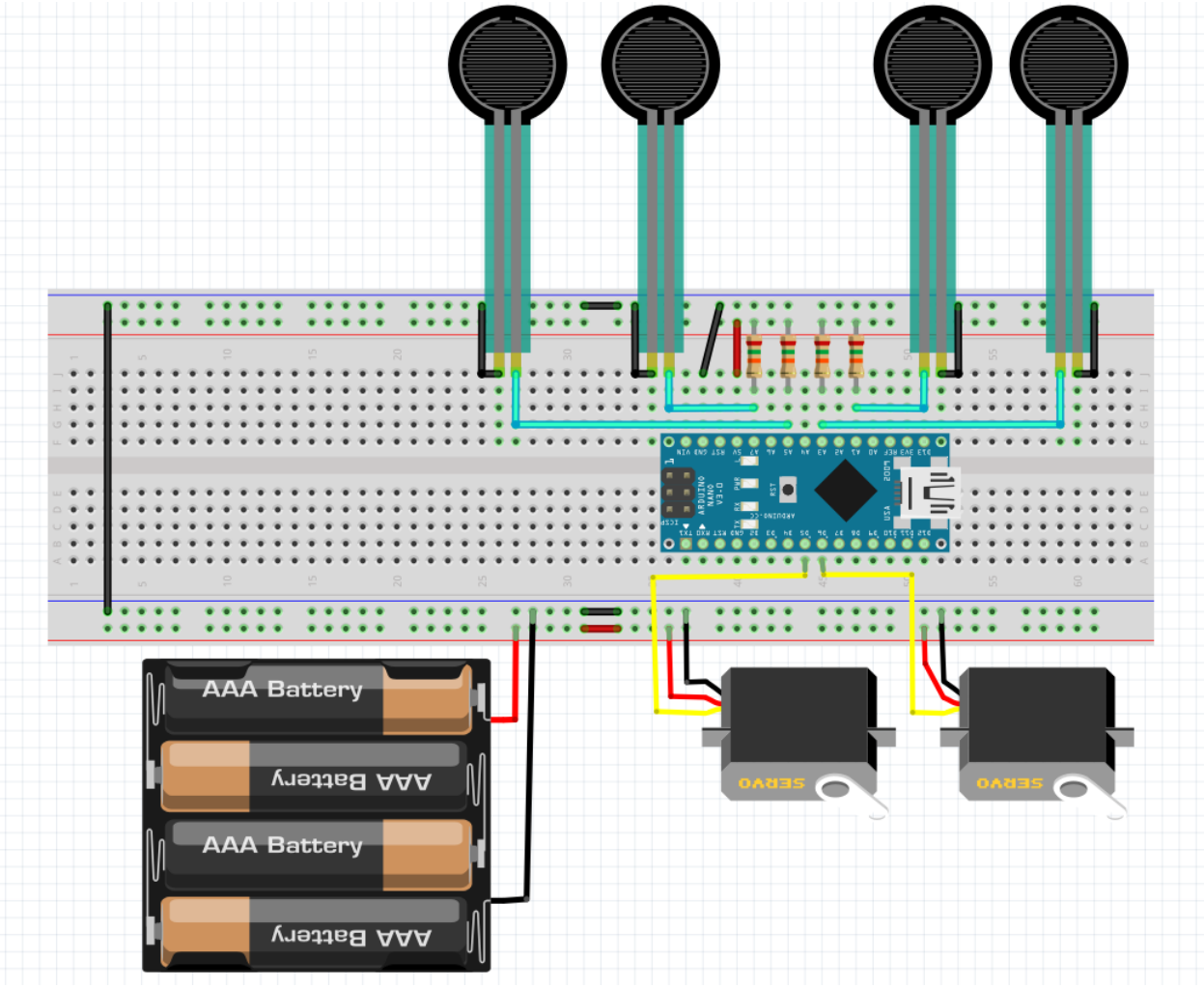

Figure 4.9: 2 DOF Robotic Arm Circuit Diagram

4.4 Hardware:

4.4.a 3D Modeling:

We have designed all of the parts of this project within solidworks, it is a well known professional 3D mechanical part modeling software compatible with 3D printing and realistic sizes (in mm), the robotic arm for this project is a 270° free horizontal mechanism which allows for just as much freedom as the simulated robotic arm.

4.4.b Main Base:

As observed in **Figure 4.10**, the main base is a circular inboxing which houses the first MG996R servo motor, it is responsible for varying θ_1 from our dynamic system.

The four long extending feet in **Figure 4.11** serve for stability and to keep the robotic arm upright when fully extended horizontally. The four L shape pinners also seen in **Figure 4.11**

serve for keeping the plate which connects the base to the first forearm upright and prevent it from tilting off.

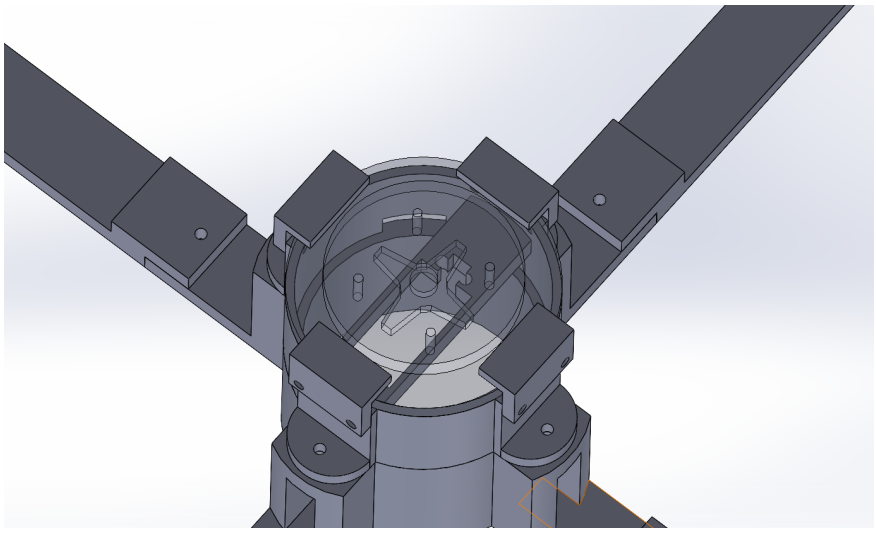

Figure 4.10: Main Base Housing

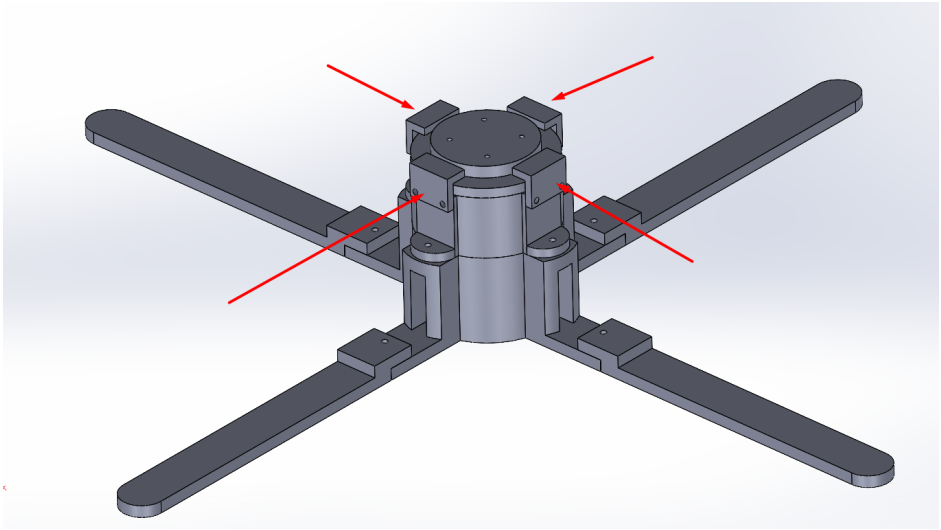

Figure 4.11: Main Base Stability Structure

4.4.c Forearms:

In **Figure 4.12**, the two links of the robotic arm are almost identical parts serving as the two forearms of the robotic arm, the horizontally cut strips serve for reducing the total weight of the arm for ease of travel.

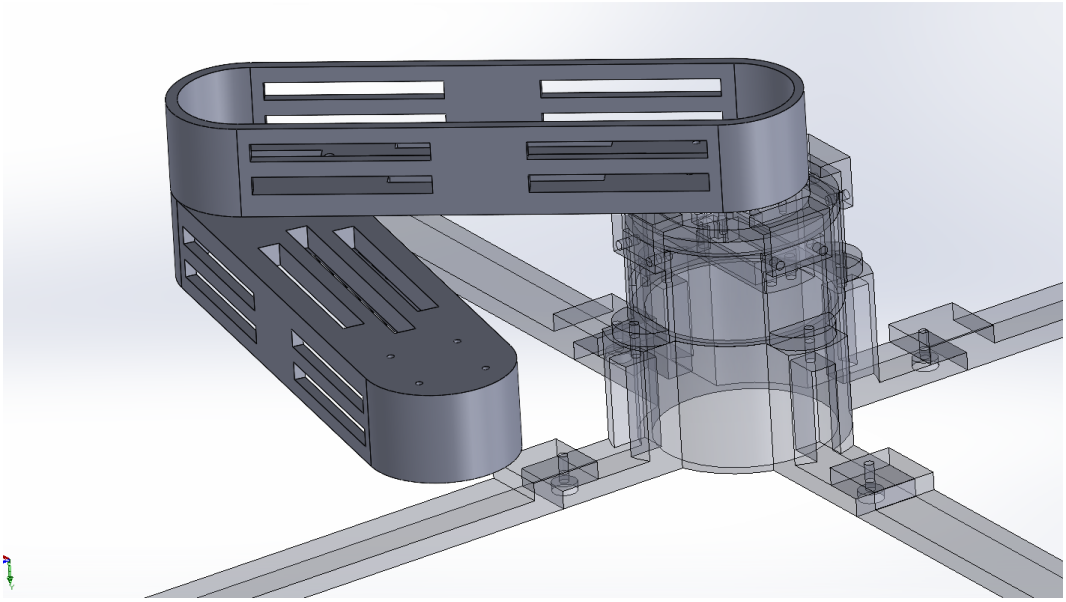

Figure 4.12: Two links of robotic arm

4.4.d End Effector:

As observed in **Figure 4.13**, the end effector of the robotic arm houses the four pressure sensors with a sliced ball body, the external slice body converts the pressure applied on the entire ball into directional pressure against the sensor as also seen in **Figure 4.14**.

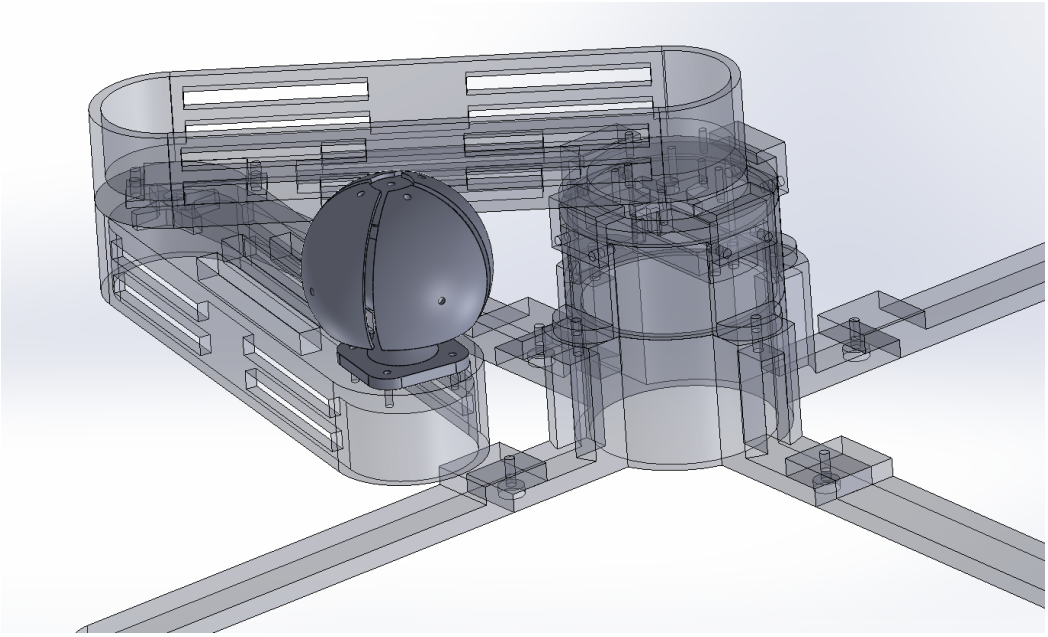

Figure 4.13: End Effector Positioning

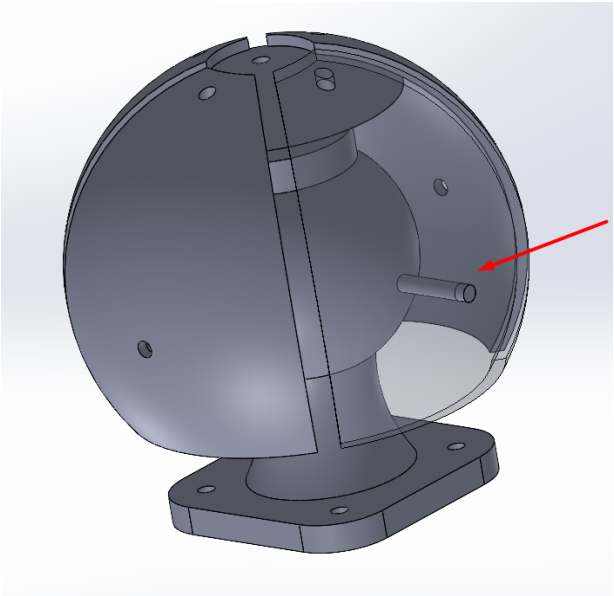

Figure 4.14: End Effector Mechanism

4.4.e 3D Printing:

The 3D printing of the parts required little support as it was designed for ease of 3D printing as observed in **Figures 4.15, 4.16 and 4.17**. The mechanism is smooth and quiet and friction is negligible. The end effector mechanism lines up with the four pressure sensors housed inside behind the sliced wings of the external touch ball which aids in simulating interaction force.

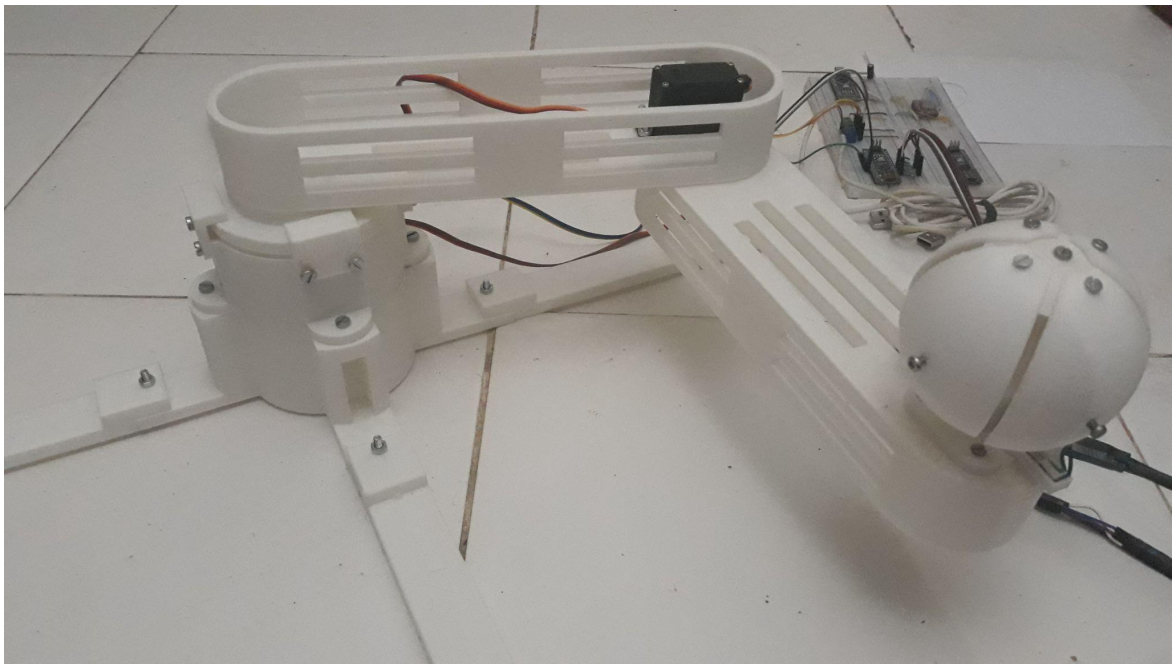

Figure 4.15: 2 DOF Robotic Arm

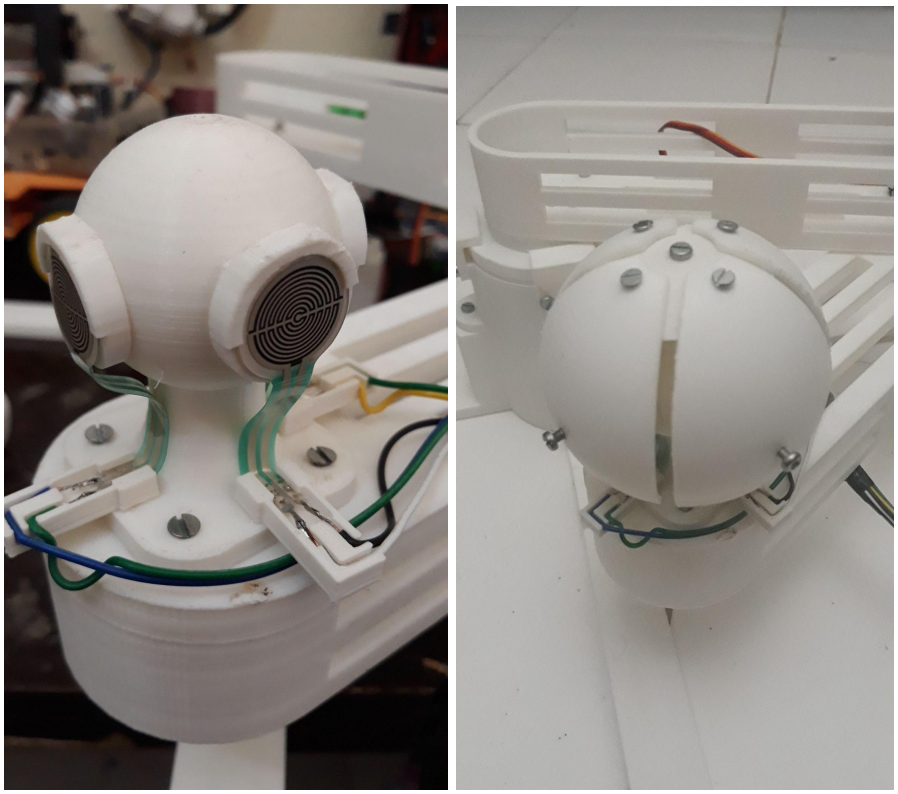

Figure 4.16: End Effector of 2 DOF Robotic Arm

Figure 4.17: MG996R Servo Motors Installation

4.5 Software:

The neural network which is trained on the simulation in chapter 2 can be directly ported into the physical robotic arm as the simulation directly represents the real representation, the benefit is that neural networks can be trained within accelerated and fast environments within the simulation. When fully trained and tested, the best neural network is directly applied onto the real robotic arm saving time and physical strain as training on a real robotic arm is unpredictable and may destroy the robotic arm during failures.

As per the robotic arm's diagram seen in **Figure 4.18**, the Python code is essentially identical to the simulation with a communication part introduced in order to communicate with the arduino.

The arduino communicates with the robotic arm by applying the motor angles received from Python and reporting the interaction forces measured from the sensors back to Python for activating the pre-trained neural network.

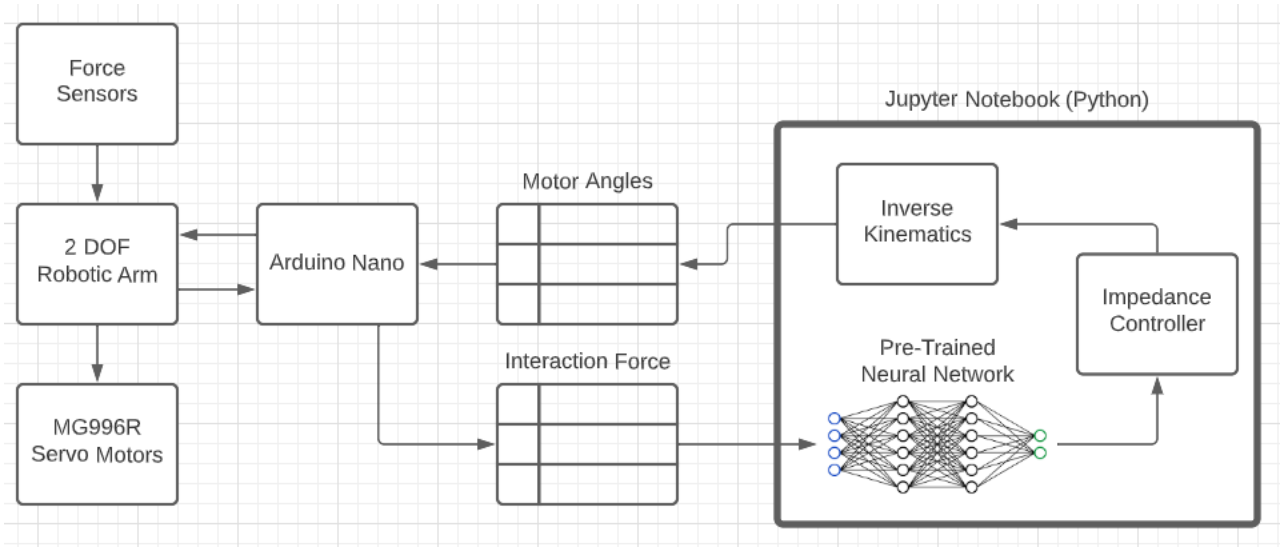

Figure 4.18: 2 DOF Robotic Arm Diagram

In this implementation, libraries such as N.E.A.T, graphical interfaces and reporting (libraries such as OpenCV) and identical matrix implementation to that of Matlab with access to the aforementioned libraries within the same runtime. Sympy is a Python library which offers all of the necessary Matrix operations required for developing a dynamic system.

Implementing the experimentation in Python results in a compact and reliable runtime. Jupyter Notebook is a Python library which allows for partial execution of code which is necessary in this project.

4.5.a Library Imports:

As noticed in **Figure 4.19**, the required libraries for the experiment are first imported:

- **cv2**: OpenCV library which allows for serving matrices of pixels of colors to be displayed and interacted with by the user.
- **numpy**: The essential numerical operations Python library which organizes matrices and performs matrix multiplication
- **sympy**: The essential symbolic numerical operations Python library which is necessary for symbolic equations and developing our dynamic system. ODE integration is not required as integration will be performed in real time with dt
- **neat**: Standing for Neuro-Evolution of Augmented Topologies, this library envelops all of the necessary neural network and genetic algorithm functions.

Imports

```
•[1]: # Graphical Interfacing
      import cv2
      # Mathematical Operations
      from math import sqrt
      import numpy as np
      import sympy as smp
      # Genetic Algorithms
      import neat
```

Figure 4.19: Imported libraries Code

4.5.b Symbol Declaration:

As noticed in **Figure 4.20**, the necessary symbols used in the dynamic system's differential equations and the impedance controller's symbols are declared using Sympy.

Symbols Init

```
•[2]: g = smp.symbols('g')
      M1, M2 = smp.symbols('M1, M2')
      L1, L2 = smp.symbols('L1, L2')
      T1, T2 = smp.symbols('T1, T2')
      Fx, Fy = smp.symbols('Fx, Fy')


      KP, KD = smp.symbols('KP, KD')
      desiredX, desiredY = smp.symbols('desiredX, desiredY')
      desiredXdott, desiredYdott = smp.symbols('desiredXdott, desiredYdott')
      currentX, currentY = smp.symbols('currentX, currentY')

      the1, the2 = smp.symbols('the1, the2')
      the1d, the2d = smp.symbols('the1d, the2d')
      th1dd, th2dd = smp.symbols('the1dd, the2dd')
```

Figure 4.20: Declared Equation Symbols Code

4.5.c Dynamic System:

As noticed in **Figure 4.21**, the equations of the forward kinematics, jacobian and dynamic model determination are declared accordingly.

Dynamic Model

```
•[3]: # Equation (12) in Report
J = smp.Matrix([[- L2*smp.cos(th1 + th2) - L1*smp.cos(th1), -L2*smp.cos(th1 + th2)],
                [- L2*smp.sin(th1 + th2) - L1*smp.sin(th1), -L2*smp.sin(th1 + th2)],
                [                                        0,                      0]])
# Equation (26) in Report
Fd = smp.Matrix([Fx, Fy]).T;
# Equations (2) and (3) in Report
P1 = smp.Matrix([  L1*smp.cos(the1) , L1*smp.sin(the1) ])
# Equations (4) and (5) in Report
P2 = smp.Matrix([ L1*smp.cos(the1) + L2*smp.cos(the1 + the2) ,
                  L1*smp.sin(the1) + L2*smp.sin(the1 + the2) ])
# Equation (23) in Report
M = smp.Matrix([ [(M1+M2)L12 + m2*L22 + 2*M2*L1*L2*smp.cos(the2), M2*L2**2 + M2*L1*L2*smp.cos(the2)] ,
                 [M2*L2**2 + M2*L1*L2*smp.cos(the2)             , M2*L2**2                         ] ])
# Equation (24) in Report
B = smp.Matrix([ -M2*L1*L2*(2*the1d*the2d+the2d**2)*smp.sin(the2) ,
                 -M2*L1*L2*the1d*the2d*smp.sin(the2)              ])
# Equation (25) in Report
G = smp.Matrix([ - (M1+M2)*g*L1*smp.sin(the1) - M2*g*L2*smp.sin(the1+the2) ,
                 - M2*g*L2*smp.sin(the1+the2)                              ])

F = smp.Matrix([ Fd*P1 , Fd*P2 ]).T

Fimp = smp.Matrix([  T1 , T2 , 0 ]).T
# Equation (28) in Report
thedd = M.inv() * ( - B - G + J.T*Fimp + F)
```

Figure 4.21: Dynamic System, Jacobian & Forward Kinematics Code

4.5.d Gravity Compensation:

As noticed in **Figure 4.22**, the gravity compensation equation is solved using Sympy in order to determine the gravity compensation.

Gravity Compensation

```
Tformulas = smp.solve([thedd[0], thedd[1]], (T1, T2), simplify=True)
Tformulas = [Tformulas[T1], Tformulas[T2]]
```

Figure 4.22: Gravity Compensation determination Code

4.5.e Impedance Controller:

As noticed in **Figure 4.23**, the equation of the impedance controller is initialized in order to determine the symbolic formula for Fimp

Impedance Control

```
desiredPosition = smp.Matrix([desiredX, desiredY, 0 ])
desiredVelocity = smp.Matrix([desiredXdott, desiredYdott, 0])

thed = smp.Matrix([th1d, th2d])

currentPosition = smp.Matrix([currentX,  currentY, 0])

Fimp = KP*(desiredPosition - currentPosition) + KD*(desiredVelocity - J*thed)
```

Figure 4.23: Cartesian Impedance Controller Code

4.5.f Symbolic Expression to Function Conversion:

As noticed in **Figure 4.24**, the aforementioned equations are converted from symbol expressions into callable Python functions for our implementation.

Turn formulas into functions

```
[5]: thdotdot1_f = smp.lambdify( (Fx, Fy, T1, T2, g, M1, M2, L1, L2, th1, th2, th1d, th2d) , thedd[0])
     thdotdot2_f = smp.lambdify( (Fx, Fy, T1, T2, g, M1, M2, L1, L2, th1, th2, th1d, th2d) , thedd[1])

     gravityCompensation1_f = smp.lambdify((Fx, Fy, g, M1, M2, L1, L2, th1, th2, th1d, th2d), Tformulas[0])
     gravityCompensation2_f = smp.lambdify((Fx, Fy, g, M1, M2, L1, L2, th1, th2, th1d, th2d), Tformulas[1])

     impedanceControl1_f = smp.lambdify((KP, KD, desiredX, desiredY, desiredXdott, desiredYdott, currentX, currentY, L1, L2, th1, t
     impedanceControl2_f = smp.lambdify((KP, KD, desiredX, desiredY, desiredXdott, desiredYdott, currentX, currentY, L1, L2, th1, t
```

Figure 4.24: Symbolic to Function Conversion Code

4.5.g Graphical Interface:

As noticed in **Figure 4.25**, the graphical interface is developed using OpenCV and the frames are declared using Numpy as matrices of pixels where the θ angles determined from the dynamic system after integration are used to display the robotic arm accurately.

The interaction force and total energy consumed are both graphed next to the robotic arm and the graphical pixel representations are combined together using the Numpy Vstack and Hstack functions, the graphical matrix is finally displayed.

The current impedance controller gains and interaction force f_e are displayed in the top left of the window.

```python
def animation_thread():
    global window, stopped

    cv2.namedWindow('output', cv2.WINDOW_AUTOSIZE)
    # allow the user to edit the desired point real time
    cv2.setMouseCallback('output', forceIntroduced)

    # graph init:
    graph1Values = []
    graph2Values = []
    graph1 = np.ones((graph_height, graph_width, 3), dtype=np.uint8) #*255
    graph2 = np.ones((graph_height, graph_width, 3), dtype=np.uint8) #*255

    # graphing forces needed
    simulationStarted = False

    while True:

        if cv2.waitKey(1) & 0xFF == ord('q'):
            stopped = True
            break

        if stopped:
            break

        if lowest_max_speed!=None:
            cv2.putText(window,"Peak Velocity: " + str(lowest_max_speed) + " m/s",
                (10,20), font, fontScale, fontColor, thickness, lineType)
            cv2.putText(window,"Fd: " + str(lowest_interaction_force) + " N",
                (10,40), font, fontScale, fontColor, thickness, lineType)
            cv2.putText(window,"Gains: " + "Kp: " + str(round(gains_of_genom_with_lowe
                (10,60), font, fontScale, fontColor, thickness, lineType)


        # debugging: display all genoms to the screen

        if not simulationStarted:
            simulationStarted = len(errorsWithForce) > 0
        else:
            if forceBeingIntroduced:
                graph1, graph1Values = graph(None, errorsWithForce[0], graph1Values)
            else:
                graph1, graph1Values = graph(None, errorsWithDesired[0], graph1Values)

        graph2, graph2Values = graph(None, 1*20, graph2Values)
        graphs = np.vstack((graph1, graph2))
        todDisplay = np.hstack((window, graphs))
        cv2.imshow('output', todDisplay)

    cv2.destroyAllWindows()
```

Figure 4.25: Graphical Interface Function Code

4.5.h Impedance Controller Execution:

As noticed in **Figure 4.26**, the impedance controller symbolized functions are executed and gravity compensated before returning the cartesian values

```python
def impedenceControl(self, currentX, currentY):

    tempDesiredX, tempDesiredY = center_me(self.desiredX, self.desiredY)

    impedanceControl1 = impedanceControl1_f(self.Kp, self.Kd, tempDesiredX, tempDesiredY, self.desiredXdott, self.desiredY
    impedanceControl2 = impedanceControl2_f(self.Kp, self.Kd, tempDesiredX, tempDesiredY, self.desiredXdott, self.desiredY

    # gravity compensation
    gravityCompensation1 = gravityCompensation1_f(0, 0, self.g, self.m1, self.m2, self.l1, self.l2, self.theta1, self.thet
    gravityCompensation2 = gravityCompensation2_f(0, 0, self.g, self.m1, self.m2, self.l1, self.l2, self.theta1, self.thet
    T1_ = impedanceControl1 + gravityCompensation1
    T2_ = impedanceControl2 + gravityCompensation2

    return T1_, T2_
```

Figure 4.26: Impedance Controller Function Code

4.5.i Dynamic System Execution:

As noticed in **Figure 4.27**, the calculated impedance controller's forces and experienced interaction force are used in the execution of the dynamic system differential equations, the resulted angular accelerations are then integrated into velocity and further into the two θ angles, the period dt is predefined which allows for real time integration alongside the window.

```
    theta1dotdot = thdotdot1_f(Fx_, Fy_, T1_, T2_, self.g, self.m1, self.m2,
    theta2dotdot = thdotdot2_f(Fx_, Fy_, T1_, T2_, self.g, self.m1, self.m2,

    self.theta1dot = theta1dotdot*dt + self.theta1dot
    self.theta2dot = theta2dotdot*dt + self.theta2dot

    self.theta1 = self.theta1dot*dt + self.theta1
    self.theta2 = self.theta2dot*dt + self.theta2
```

Figure 4.27: Dynamic System Execution Code

4.5.j Genetic Algorithm:

As noticed in **Figure 4.28**, the function nominated as run() initializes the genetic algorithm with our specifications such as the activation function mentioned in this chapter, the amount of generations which the neural networks are run through, the suggested depth of the neural network as well as minimum accepted fitness values which the neural networks must surpass to be deemed successful.

Furthermore, the population is also created in this function via the neat.Population() internal function. Finally, the fitness function is provided to the genetic algorithm, a fitness function governs the operation of the genomes and contains the policy and rules that the neural networks must abide by to get rewarded.

```
def run(config_path):
    config = neat.config.Config(neat.DefaultGenome, neat.DefaultReproduction, neat.DefaultSpeciesSet, neat.DefaultStagnation, config_path)

    p = neat.Population(config)

    winner = p.run(fitness_function, 100)

config_path = os.path.join("impedenceConfig.txt")
run(config_path)
```

Figure 4.28: Genetic Algorithm Initialization Code

As observed in **Figure 4.29**, the fitness function first initializes the list of neural networks and their corresponding genomes and robotic arms which will undergo the current generation's test, all of the neural networks used in our operation are feed forward deep convolutional neural networks with initial fitness of 0.

```
def fitness_function(genomes_, config):
    GEN += 1

    neural_networks = []
    genomes = []
    arms = []

    for _, ge in genomes_:
        net = neat.nn.FeedForwardNetwork.create(ge, config)
        neural_networks.append(net)

        genom = Arm(m1, m2, l1, l2, g_, initial_theta1, initial_theta2, initial_theta1dot, initial
        arms.append(genom)
        ge.fitness = 0
        genomes.append(ge)

    while True:
        for i, arm in enumerate(arms):


            try:
                arm.updateDesired(desiredX, desiredY, initial_desiredXdott, initial_desiredYdott)

                # obtain the interaction force
                interaction_force = arm.getInteractionForce()

                # the Ai will decide values for Kp and Kd
                outputs = neural_networks[i].activate((interaction_force,))

                new_Kp = outputs[0]
                new_Kd = outputs[1]

                arm.updateGains(new_Kp, new_Kd)

                arm.updateTheta()

                window = arm.draw(window)


                timed = millis() - last_desiredPoint_update
                if timed > convergencePeriod:
                    if interaction_force >= 0.03:
                        genomes[i].fitness -= 10
                        arms.pop(i)
                        neural_networks.pop(i)
                        genomes.pop(i)
                    else:
                        genomes[i].fitness += 10
                else:
                    genomes[i].fitness += 10

            except ValueError as e:
                genomes[i].fitness -= 10
                arms.pop(i)
                neural_networks.pop(i)
                genomes.pop(i)

    if timed > convergencePeriod:
        for arm in arms:
            print("Winner", arm.getGains())
```

Figure 4.29: Fitness Function Code

A continuous while loop operates the fitness function until the generation is deemed over (total extinction of the genomes or deemed successful genomes still alive), in each iteration the list of robotic arms is looped through and processed strictly to the following steps:

- **Normal Operation:** Decide the current desired position of the end effector (normal uninterrupted operation)
- **Normal Operation:** Sense any external interaction forces/pressures against the robotic arms
- **Genetic Algorithm Operation:** The neural network of the robotic arm is then activated with the measured interaction force and is able to return two outputs due to the two output neurons we predefined in the configuration, these two values being the stiffness and damping gains of the impedance controller.
- **Normal Operation:** The gains are then updated into the arm's impedance controller, and the dynamic system's differential equation is then executed for the

current sample of time with the decided impedance controller's gains from the neural network.

- **Reward/Punishment System:** The reward and punishment, also known as the policy function is what allows the engineer to influence the general direction of learning for the neural networks, the fitness value is rewarded for executing desired actions (maintaining low interaction force and converging to the desired point with no interaction force), and is punished for undesired actions (diverging with unstable gains, high stiffness and high interaction force).

As observed in **Figure 4.30**, a time limit is enforced on the genomes to encourage the fastest convergence with the lowest interaction force conceivable. Otherwise, the genomes would settle with zero movement and in some cases oscillate infinitely around the desired point after external interaction force has reached zero (empty space)

```
if timed > convergencePeriod:
    for arm in arms:
        print("Winner", arm.getGains())
```

Figure 4.30: Time limit Code

4.5.k Arduino:

The arduino program however must apply commands from Python to the servos and report measured interaction forces back to Python.

.As observed in **Figure 4.31**, initially the count of sensors attached to the robotic arm is initialized, in our case it is capped at 4. The sensors are read via analog pins and the readings are reported periodically to Python within a combined string message via Serial Communication Protocol.

```
..
const int sensor_count = 4;
int sensor_analog_pins[sensor_count] = {A3, A5, A1, A7};

 // read sensor values, report if appropriate time
 if(millis() - previous_report_time > max_sensor_vals_report_period){
   previous_report_time = millis();
   report = "Ok, sensors:";
   for(int i=0; i<sensor_count; i++)
     report +=  " " + String(1023 - analogRead(sensor_analog_pins[i]));
   Serial.println(report);
 }
```

Figure 4.31: Reporting Sensor Values

Arduino nano also receives the servo motor angles decided by Python by inverse kinematics from the output of the impedance controller as observed in **Figure 4.32**.

```
if(parts_of_msg[0] == "ANGLES"){ // ANGLES <angle1> <angle2>
  long temp_angle1 = parts_of_msg[1].toInt();
  long temp_angle2 = parts_of_msg[2].toInt();
  go_to_coordinates(temp_angle1, temp_angle2, Speed);

  Serial.println("Ok, updating angles into " + String(angle1) + " " + String(angle2));
  Serial.flush();
  return;
```

Figure 4.32: Receiving Motor Values

The function go_to_coordinates() responsible for directing the motors towards the received angles synchronizes the steps of the two motors allowing for simultaneous travel from current location to the decided location by the impedance controller as observed in **Figure 4.33**.

```
void go_to_coordinates(int m1, int m2, int steps){

  long angle1_step = (m1 - angle1)/steps;
  long angle2_step = (m2 - angle2)/steps;

  for(int i= 0; i < another_speed; i++){
    angle1 += angle1_step;
    motor1.write(angle1);
    angle2 += angle2_step;
    motor2.write(angle2);
  }
}
```

Figure 4.33: Go to Coordinates Function

4.5.l Python Serial Communication:

Python uses the pyserial library which allows it to send the decided motor angles to Arduino via Serial Communication Protocol, and receive force sensor reports as in **Figure 4.34**.

```
# run communication
while running:
    if motor1_angle != current_motor1_angle or motor2_angle != current_motor2_angle:

        if time() - previous_command_time > period:
            previous_command_time = time()

            # save the data
            current_motor1_angle = motor1_angle
            current_motor2_angle = motor2_angle

            #print("Applying angles:", current_motor1_angle, "°", current_motor2_angle, "°")
            send_command(serial, "ANGLES " + str(current_motor1_angle) + " " + str(current_motor2_angle))

    # FUTURE: check for force reports in order for us to decide where to take the robot arm

    # check serial for msgs
    bundle = Read(serial)
    message = Clean(bundle)
    if message: # skip what's below this line
```

Figure 4.34: Serial Communication Function

4.5.m Impedance Controller:

Since the 2 DOF robotic arm travels on a two dimensional plane (x and y), the applied force naturally consists of a two dimensional vector across the x and y axis. Hence, the impedance controller's input formula is tuned to calculate the length of the vector of applied force and velocity before execution as observed in **Figure 4.35**.

```
# impedance contrller's output
impedanceOutput = Kp * sqrt(px_difference**2+py_difference**2) \
        + Kd * sqrt((px_difference-previous_px_difference)**2+(py_difference-previous_py_difference)**2)
```

Figure 4.35: Impedance Controller Formula

4.6 Experimentation Results:

In this section, we present and discuss the results of the genetic algorithm's performance on a real 2 DOF robotic arm, the robotic arm will be subjected to three manual interventions by hand in three different strengths, it is expected that the arm reacts similarly to that in the simulation but it is essential to note that the real arm only consists of four force sensors positioned at the end effector as seen in **Figure 4.36**.

4.6.a First case: ($f_e = 50N$) As observed in **Figure 4.36**'s force graph in the top right hand side, the sensor is subjected to 51% of its maximum datasheet physical strain, the neural network is able to influence the robotic arm according to the direction of applied force by determining the gains of the impedance controller. As observed in **Figure 4.36**, the arm's transient phase time is 2.9s with a 13.79% error from the simulated value, a total settling time of 3.2s with a 8.57% error from the simulated value.

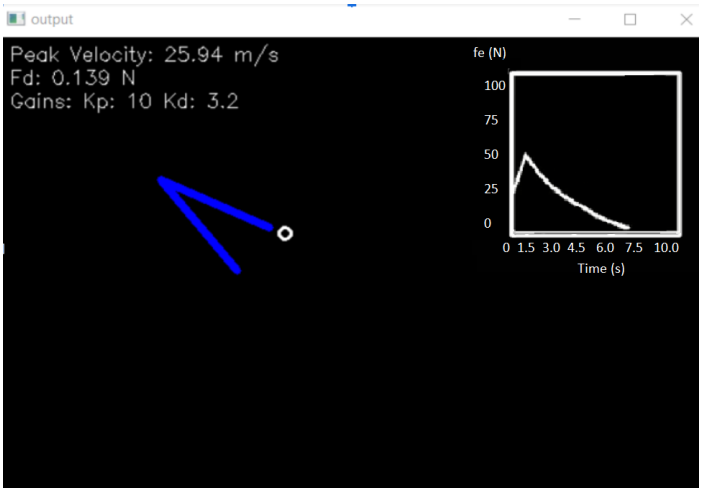

Figure 4.36: Test case 1 transient response

The result obtained aligns with the simulated results with little inaccuracies due to the limited amount of sensors.

4.6.b Second case: ($f_e = 5N$) In a similar manner, the robotic arm is influenced by approximately 5N of applied force, the robotic arm displays comparable transient response with smooth and non-stiff interaction as observed in **Figure 4.37** resulting in a transient phase time of 2.5s (8% error) and a total settling time of 2.7s (12.9% error).

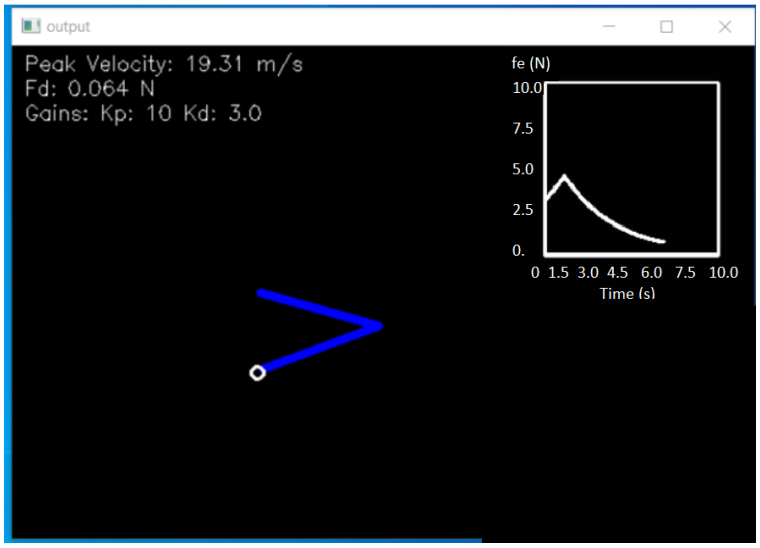

Figure 4.37: Test case 2 transient response

The neural network is able to perform adequately the same at different applied interaction forces.

4.6.c Third case: ($f_e = 0.5N$) In a similar manner , the robotic arm is influenced by approximately 0.5N of applied force, the robotic arm displays no reaction as observed in **Figure**

4.38, this is due to the minimum of 1N required interaction force on the FSR402 sensors used in this operation.

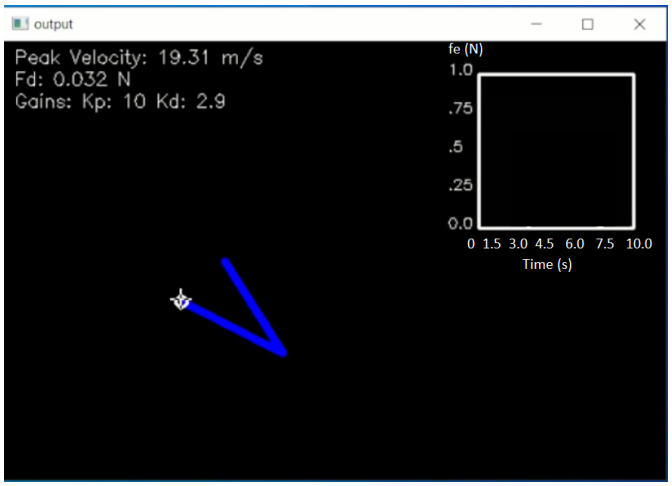

Figure 4.38: Test case 3 transient response

The results of this experiment and settling time errors compared to simulation are resumed in the following table

Force Introduced f_e	Transient Phase (s)	Settling Time (s)	Error Settling Phase (%)
0.5 N (0.05kg)	X	X	X
5 N (0.5kg)	2.5	2.7	12.9%
50 N (5.1kg)	2.9	3.2	8.57%

4.7 Conclusions:

- In this chapter, we applied the trained neural network to the real robotic arm and compared performances accordingly.
- The values obtained highly compared to the simulated values, the experienced errors are due to the limited count of force sensors.
- As potentially noticed, simulating robots before instantiating the physical implementation allows for maximum development at low cost (software simulation vs hardware implementation), it is also beneficial in training neural networks at astronomical speeds with realistic neural networks.
- The arm was exposed to 0.5N , 5N and 50N of applied force and it performed adequately. A genetically trained neural network can adapt to multiple different orders of magnitude of interaction forces.
- Due to the minimum of 1N of applied force implied by the affordable FSR402, the third test case is unable to be tested on the real robotic arm.

General Conclusion

Traditional impedance control with predefined gains is an unadaptable system which cannot be ported into environments in which it is not specifically tuned for, adaptive impedance controllers introduce variable and adaptable gain factors taking into account the interaction force with the environment.

Genetic algorithm training presents a vast improvement in evaluation and training of adaptive impedance control, it allows the dynamic controller to adapt its actions and learn from past experiences in a limitless manner.

In this report we sought for an upgraded adaptive impedance control model based on genetic algorithms, it combined reinforcement learning and supervised learning for a more flexible, unlimitedly trainable without encountering limits in local maximas and with adequate performance across a vast range of interaction force cases from an unknown environment.

Varying the gains based on interaction force allows for grasping the decision process adopted by the neural network and as such; learning from it. During the training of the neural network, intelligent habitats were noticed such as the neural network reducing stiffness considerably during rest in anticipation of sudden interaction forces which was conceived by the neural network from the genetic algorithmic training supplied.

The simulation and experimental results showed that the genetic algorithm performed adequately and in a maintained manner against unknown interaction forces from the three test cases introduced to it at random instants of time. The proposed framework can ensure a small interaction force and achieve a smooth interaction. This desired goal was achieved in a simulated dynamic model with realistic interaction forces and a real experimental two degrees of freedom robotic arm.

Implementations of impedance controlled robotics is a difficult and costly feat, the FSR402 physical pressure sensors have hard limits to force detection which highly affect the performance of the robotic arm in a real environment. To improve on this experiment, more costly and more accurate sensors are to be used in higher quantities (only 4 were used in this experiment placed around the end effector).

Arduino and Python are handy experimental tools used in developing systems and prototypes in a short period of time but are inaccurate and slow. More persistve and compute efficient systems can and will improve operation and simulation results such as faster programming languages and more capable microcontrollers which are built with stability and performance in mind.

References:

[1] W. He and Y. Dong, “Adaptive fuzzy neural network control for a constrained robot using impedance learning,” IEEE Transactions on Neural Networks and Learning Systems, vol. 29, no. 4, pp. 1174–1186, 2018.

[2] Journal of Dynamic Systems, Measurement and Control - Hogan MARCH 1985

[3] Z. Li, B. Huang, A. Ajoudani, C. Yang, C.-Y. Su, and A. Bicchi, “Asymmetric bimanual control of dual-arm exoskeletons for human-cooperative manipulations,” IEEE Transactions on Robotics, vol. 34, no. 1, pp. 264– 271, 2018.

[4] Q. Xu and S. S. Ge, “Adaptive control of redundant robot manipulators with null-space compliance,” Assembly Automation, vol. 38, no. 5, pp. 615–624, 2018.

[5] W. He, S. S. Ge, Y. Li, E. Chew, and Y. S. Ng, “Neural network control of a rehabilitation robot by state and output feedback,” Journal of Intelligent & Robotic Systems, vol. 80, no. 1, pp. 15–31, 2015.

[6] Y. Li, K. P. Tee, W. L. Chan, R. Yan, Y. Chua, and D. K. Limbu, “Continuous role adaptation for human–robot shared control,” IEEE Transactions on Robotics, vol. 31, no. 3, pp. 672–681, 2015.

[7] Z. Li, B. Huang, Z. Ye, M. Deng, and C. Yang, “Physical human– robot interaction of a robotic exoskeleton by admittance control,” IEEE Transactions on Industrial Electronics, vol. 65, no. 12, pp. 9614–9624, 2018.

[8] Z. Li, J. Liu, Z. Huang, Y. Peng, H. Pu, and L. Ding, “Adaptive impedance control of human–robot cooperation using reinforcement learning,” IEEE Transactions on Industrial Electronics, vol. 64, no. 10, pp. 8013–8022, 2017.

[9] S. S. Ge, Y. Li, and C. Wang, “Impedance adaptation for optimal robotenvironment interaction,” International Journal of Control, vol. 87, no. 2, pp. 249–263, 2014.

[10] H. Navvabi and A. H. Markazi, “Hybrid position/force control of stewart manipulator using extended adaptive fuzzy sliding mode controller (eafsmc),” ISA transactions, vol. 88, pp. 280–295, 2019.

[11] Variable Impedance Control and Learning—A Review - Frontiers 2020

[12] Active Impedance Control of Bioinspired Motion Robotic Manipulators: An Overview Volume 2018, Article ID 8203054

[13] Joanna Szyman for PMR: Robotics - the future of surgery 2019

[14]Surgery on a grape Ashitha Nagesh

[15] Boston Dynamics Spot - Jon Porter - 2021

[16]https://www.therobotreport.com/boston-dynamics-stretch-now-commercially-available/

[17]https://www.youtube.com/watch?v=lULK_e0LK70

[18]https://www.wired.com/story/elon-musk-defies-lockdown-orders-reopens-tesla-factory/

[19]https://www.frontiersin.org/journals/robotics-and-ai

[20]Machine Learning Techniques for Personalized Medicine Approaches in Immune-Mediated Chronic Inflammatory Diseases: Applications and Challenges 2021

[21]An introduction to neural networks - Alex K

[22] Tuna Orhanli (2022). Forward Dynamics of Planar 2-DOF Robot Manipulator, MATLAB Central File Exchange. Retrieved May 21, 2022.

[23] Activation Functions: https://neat-python.readthedocs.io/en/latest/activation.html

[24] https://nettigo.eu/products/sg90-small-hobby-servo

[25] https://www.jsumo.com/mg90s-micro-servo-motor

[26] https://www.az-delivery.de/en/products/az-delivery-servo-mg996r

[27] https://ar.aliexpress.com/item/32894180679.html?gatewayAdapt=glo2ara

[28] https://ar.aliexpress.com/item/32952522356.html?gatewayAdapt=glo2ara

[29] https://ar.aliexpress.com/item/32883191019.html?gatewayAdapt=glo2ara

[30] https://opencircuit.shop/product/arduino-nano-r3-clone

[31] https://all3dp.com/1/creality-ender-3-v2-review-3d-printer-specs/

[32] Back propagation & Its implementation details - Lu Zheng 2020

[33] NEAT documentation https://neat-python.readthedocs.io/en/latest/

[34] Reinforcement Learning (DQN) - Adam Paszke

[35] Variable impedance control of a robot for cooperation with a human - Ikeura and Inooka (1995)
https://www.semanticscholar.org/paper/Variable-impedance-control-of-a-robot-for-with-a-Ikeura-Inooka/14ed240d18b8798731caa7179cee13476262e9ce