People's Democratic Republic of Algeria

Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdès

Institute of Electrical and Electronic Engineering Department of Electronics

Adaptive Impedance Control for Unknown Environment

Masters Degree Project

Author: Benbouali Mohamed Amine

Supervisor: Mrs. Derragui Nabila

Abstract

Impedance control is one of the most well known approaches to dynamic control, it has enabled human-robot and robot-robot interaction for decades as many advanced robots in factories and out in unpredictable environments rely heavily on it to reduce interaction force with the environment, these robots have enabled robotic surgery which is allowing for minimally invasive surgeries to take place around the world today, they enable the surgeon to move the robotic arms in a smooth and non-stiff manner, impedance control is also used in humanoid robots to be able to travel through the environment with very little stiffness which would otherwise be a danger to humans and the environment, impedance controlled robots have the ability to reduce their stiffness while navigating the environment without losing power and falling apart, ever since the creation of impedance control it has improved little by little reaching the ability to tune the control gains based on interaction force, which is known as adaptive impedance control, deemed better and more diverse, today machine learning is entering the world of impedance control introducing smart predictive control.

Acknowledgements

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

Content List

Abbreviations & Symbols Table

Ai: Artificial Intelligences

ML: Machine Learning

VIC: Variable Impedance Control

VIL: Variable Impedance Learning

VILC: Variable Impedance Learning Control

NEAT: Neuro-Evolution of Augmented Topologies

DOF: Degrees of freedom

Activation Function: Mathematical functions mapping neural network outputs into a range between -1 and 1, rendering it usable for decision making.

Jupyter Notebook: A Python library allowing partial execution of programming code, it aids in organization and development of simulations and machine learning algorithms

Infill Percentage: The percentage of plastic housed within the 3D printed part

Infill Pattern: The pattern in which the infill plastic is draw in in order to maintain the structure

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

Supports: The detachable plastic parts printed for the sole purpose of supporting air hanging portions of a 3D printed part during the printing process (plastic can not be printed in the middle of the air)

Neural Network: A network of float numbers assigned to nodes which are interconnected representing neurons similar to those in the human brain, a neural network is a computer representation of the human brain allowing computers to learn by tuning the float values of each neuron and gaining human cognition.

Pre-trained Neural Network: A neural network with highly tuned weights of the nodes resulting from a heavy training process, pre-trained neural networks have experience and are able to make educated decisions based on what it has learned.

Genetic Algorithm: A repetitive training process which mates and duplicates neural networks over many generations.

Genome: A neural network is governed by a repetitive training process known as Genetic Algorithms, a genome learns during a certain generation. If succeeded, it is then mated with other succeeding genomes.

General Introduction:

1. Chapter 1: Overview

1.1 Introduction:

In this chapter we will be going over what impedance control is, 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 and define the problem with them.

1.2 Overview of Impedance Control:

Robots nowadays are coming into contact with humans more than ever before. In health care facilities, factories and industrial settings, humans either work collaboratively with robots or are physically served by robots. However, robots are stiff and strong systems that cannot consider the amount of damage they apply to the environment while achieving a certain task or activity, an external control system that senses interaction forces and controls the robot in a manner which reduces interaction forces is required.

It is widely acknowledged that impedance control is one of the most effective and robust approaches of human-robot interaction (pHRI) (Haddadin and Croft 2016), and for that reason it has been the focus of research for decades, being improved upon year by year. Impedance control consists of treating the environment as an admittance and the robot manipulator as an impedance, this enables the robot to behave like a mass-damper-spring system (Albu-Schaffer el al., 2007), which allows the robot's dynamics and kinematics to be affected by the mass and inertia. Impedance control can be both implemented mechanically and in software and in this report we will be focusing on software implementation of the system controlling physical and simulated robot arms.

1.3 Applications of Impedance Control:

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

Medical Care: Da Vinci Surgical System

When it comes to robotic surgery, the Da Vinci Surgical System (Figure 1.1) 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 1.2, an example surgery was performed by the Da Vinci Surgical System on a grapefruit as part of its campaign.

Figure 1: Da Vinci Surgical System [1]

Figure 2: Da Vinci Surgical System Surgery Test [2]

<u>Unknown Environment Inspection</u>: Boston Dynamics Spot, Stretch & Atlas Robots

Boston Dynamics is a 30 year old robotics company, they manufacture smart industrial robots such as Spot (Figure 1.3), Stretch (Figure 1.4) and Atlas (Figure 1.5), these robots must interact with unpredictable environments at all times where low stiffness and delicacy is a must.

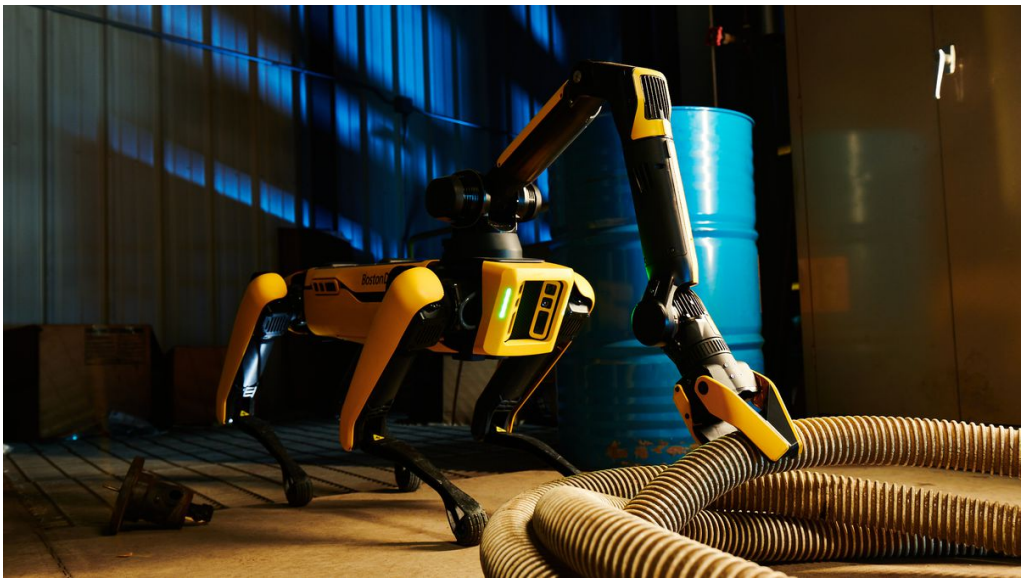

Figure 3: Spot - Boston Dynamics [3]

Figure 4: Stretch - Boston Dynamics [4]

Figure 5: Atlas - Boston Dynamics [5]

<u>Industrial Settings & Factories</u>: Tesla Factory

The Tesla factories (Figure 1.6) 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.

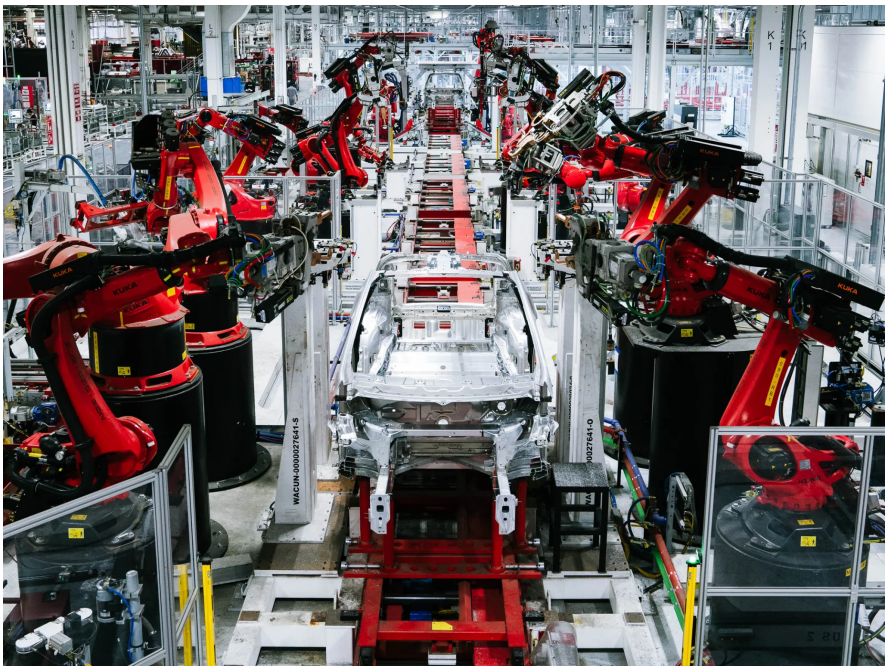

Figure 6: Tesla Factory [6]

1.4 Current State of The Art:

Impedance control with predefined parameters is not viable for an unpredictable environment. Therefore, a more practical method which allows this system to adapt the robot's impedance depending on the interaction force, called Adaptive Impedance Control. Researchers have attempted impedance adaptation and impedance learning using various techniques, some of which we will cover.

In figure 1.7 made by Frontiers in Robotics and Ai [7], the various developed techniques for adaptive and predictive impedance control are categorized.

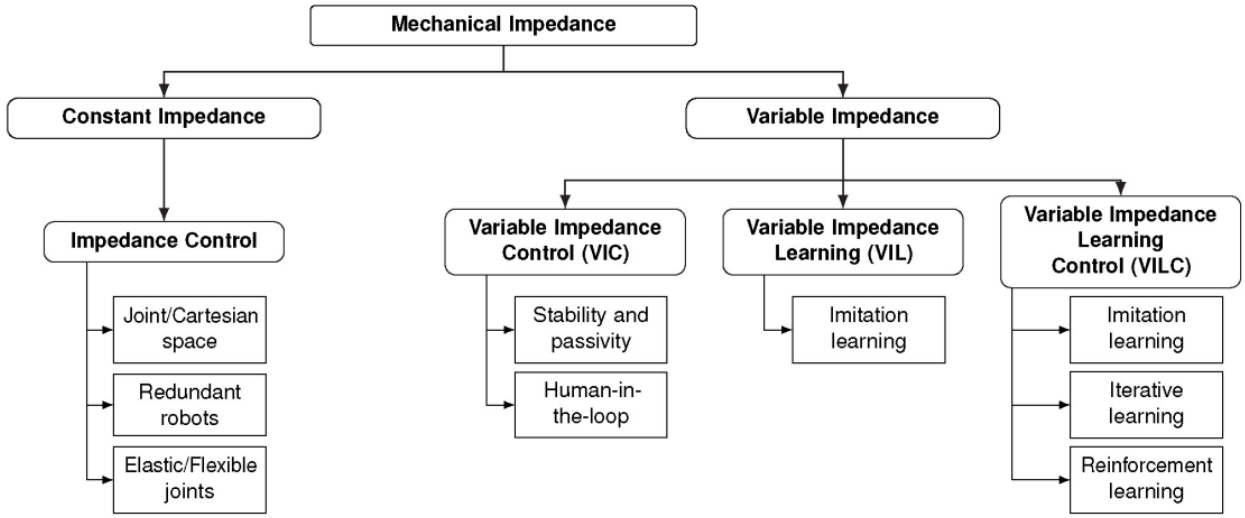

Figure 7: Existing approaches for variable impedance control and learning [7]

Under **Mechanical Impedance** we have two types, **Constant Impedance** also recognized as traditional impedance control with constant parameters, and **Variable Impedance** which includes **Variable Impedance Control** which consists of a system developed by humans that focuses on desired stability and passivity, **Variable Impedance Learning** which consists of showing a neural network various examples of what is deemed acceptable behavior, the neural network will learning to imitate those behaviors on previously seen examples from the dataset and never seen before cases, and finally **Variable Impedance Learning Control** which merges the two previous methods, excluding the human from the tuning process and teaches a machine learning model through imitation learning, **VILC** also includes **Reinforcement Learning** which does not involve a neural network and is based on the Markov Decision Process invented in 1960.

Many researchers have worked massively on **Variable Impedance**, all which we picked from the recent review by Frontiers in Robotics and Ai [1] are:

Variable Impedance Control (VIC):

VIC Stability & Passivity:

Stability and passivity does not include a human in the loop and focuses on the stability of the system; it has been studied in Hogan (1985) and later in Colgate and Hogan (1988), and later improved into a second-order linear time varying system by Lee and Buss (2008).

Ganesh et al. (2012) implemented a versatile controller able to automatically adjust the stiffness relative to a reference trajectory [1]. A passivity-based approach is presented to ensure stability of time-varying impedance controllers (Ferraguti et al., 2013).

VIC with a Human in the Loop:

Studied by Burdet et al., (2001) and Ajoudani (2016), transferring the knowledge of the human's impedance to robots via several concepts, essentially teaching the robot from specific decisions collected from a human, it is called tele-impedance.

Variable Impedance Learning (VIL):

VIL via Imitation Learning:

Imitation Learning (IL) or Learning from Demonstration (LfD) methods are tools to give machines the ability to mimic human behavior to perform a task (Hussein el al. (2017) and and Ravichandar et al. (2020).

Koromushev et al (2011) also worked on encoding the position and force information into a time-driven Gaussian Mixture Model (GMM).

Variable Impedance Learning Control (VILC):

VILC via Imitation Learning:

Calinon et al (2010) has proposed an active learning control method from training data to approximate variable stiffness from the inverse of the position covariance in a GMM.

Later, Khansari-Zadeh et al. (2014) derived the trajectory and variable impedance gains from a GMM and used them to compute the control input u.

VILC via Iterative Learning:

Based on reinforcement learning and markov decision processes, iterative learning techniques were researched by Cheah and Wang (1998), Gams et al. (2014), Uemura et al. (2014), Abu Dakka et al. (2015), Kramberger et al. (2018), confirming Bristow et al. (2006)'s conclusions that a reinforcement learning agent should be able to learn from past experience.

VILC via Reinforcement Learning:

With the rise of machine learning and all its subdomains, more work was put into developing the algorithms and optimizing the techniques with computers, reinforcement learning was bound to be applied to variable impedance control as it offers realistic learning curves to that

of a human without a predefined dataset curated by the trainer, vastly studied by Sutton and Barton (2018), Kober et al. (2013), Chatzy Geroudis et al. (2020), Arulkumaran et al. (2017) and Rey et al. (2018).

1.5 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. Stability & Passivity
2. Human-In-The-Loop
3. Imitation Learning (Supervised Learning)
4. Iterative Learning (Reinforcement Learning)

The first two strategies are viable when the environment is encapsulated with no chance of never-seen-before situations, the third strategy requires lots of training data which must be collected and provided to the system's learning algorithm by the developer of the system, meaning the training data is selected by a human who's prone to bias, unbalancing the data and missing out on many test cases, the four strategy does not require hand picked training data and allows the control system to iteratively learn on its own by exploring the environment and learning from past mistakes based on the policy function.

However, Iterative Learning with Reinforcement Learning algorithms such as the Markov Decision Process is very difficult to tune in the right direction. It does not rely on a neural network which actively learns, resulting in more human interventions in the learning process, the state and action table included in the Markov Decision Process algorithm is highly inspired by the operator's decisions and not on a smarter/less prone to mistakes party such as a neural network, a Markov agent become very impractical in non-encapsulated environments and it doesn't generalized based on specific test cases, but instead keeps in mind the proper decisions to take under specific situations.

Training a Reinforcement Learning model costs lots of capital and necessits accurate training decisions by the human, Reinforcement Learning can reach its maximum capacity in prediction very quickly if relied on by its own.

1.6 Conclusion: Chapter 1:

Reinforcement Learning and Supervised Learning are two different subdomains of Machine Learning (Figure 1.8).

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, museums, schools and aid facilities. 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. Later on, human-in-the-loop strategies were introduced by researchers where the system mimiques decisions made by a human in order to teach the model impedance control from the human brain. Finally, Adaptive Impedance Control has moved onto relying on neural networks through **Imitation Learning** by providing the artificial intelligence with thousands of training data samples hand-picked and labeled manually, and Reinforcement Learning models through **Iterative Learning** by allowing the model to experience the environment on its own without human intervention in the process. However, reinforcement learning is very difficult to train as it only relies on Markov Decision Processes, these processes include updating a state and action table based on the environment which becomes highly impractical in unencapsulated environments, a better training solution is required.

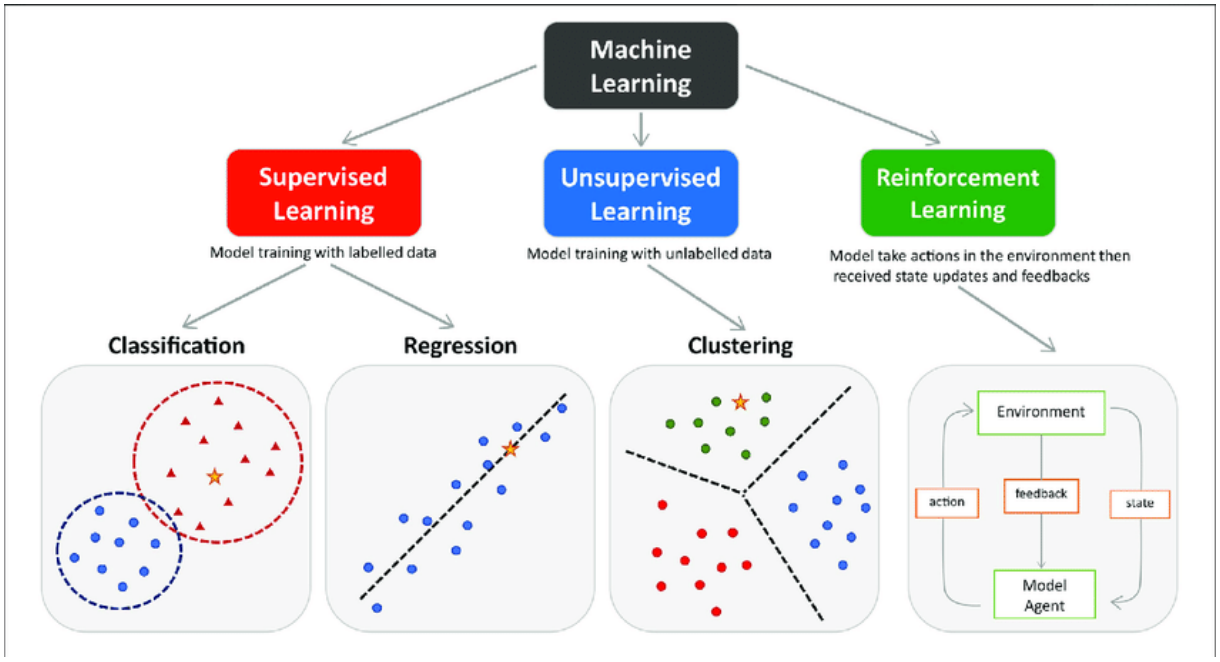

Figure 8: Subdomains of Machine Learning [8]

2. Chapter 2: Theory & Solution

2.1 Introduction:

In this chapter well will cover the theory behind each of the following:

1. Impedance Control
2. Adaptive Impedance Control
3. Genetic Algorithms

A solution is presented in this chapter, the proposed solution includes the implemented neural network, genetic algorithm and policy function suggested in order to train iteratively.

2.2 Impedance Control System:

Impedance control is a dynamic control system which interferes in the control of dynamic systems, impedance control reduces stiffness in these systems, articulated robotics and a variety of dynamic systems benefit massively from low stiffness control as it increases safety in the workplace. Every impedance control system consists of three factors multiplied by a gain each which are:

1. The stiffness factor consisting of the displacement multiplied by the stiffness gain Ks
2. The damping factor consisting of the velocity multiplied by the damping gain Kd
3. The acceleration factor consisting of the velocity multiplied by the inertial gain Ki

$$F_{impedance} = K_i \ddot{e}_t + K_d \dot{e}_t + K_s e_t \qquad (1)$$

Where $e_t = F_{desired} - F_{current}$ is the error between the desired state and current state.

The error between the two values is differentiated to obtain the damping factor, the error is then differentiated a second time to obtain the acceleration factor. The gains Ki, Kd, and Ks are predefined gains tuned specifically for a system under specific conditions, this renders regular impedance control useless in unknown environments.

2.3 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. As mentioned in chapter 1, adaptive impedance control reduces interaction force massively compared to its predecessor as it is directly related to environment conditions. Over the decades, researchers have invented several approaches such as VIC, VIL and VILC explained in chapter 1.

Artificial Intelligence is implemented more and more in the present, generally in one of two of its forms: Supervised learning and Reinforcement Learning, both add intelligence to gain definition further lowering interaction force, supervised learning predicts future interactions

based on previously train on examples while reinforcement learning does not require a dataset, instead actively training the Markov agent on real time interaction with the environment.

However, as can be noticed in supervised learning, it requires a large dataset of labeled training examples, reinforcement learning while it does not require a dataset it also does not rely on a neural network in its training but only a Markov decision tree which by itself can get very smart but is only practical in close environments where the decision tree can grasp the entire situation.

A neural network is a large constellation of neurons similar to that of the human brain, a neural network can learn just as well as a human could when trained correctly, but training a neural network requires a dataset, a solution could arise in mixing the two technologies, combing the pros in reinforcement learning with the pros in supervised learning would unlock brand new highs in predictive and adaptive impedance control.

2.4 Proposed Solution: Genetic Algorithms:

Genetic algorithm training is the process of generating many neural networks and actively training them in real time on real data, the training is done through generations and the best performing neural networks are mated and combined, the new version of the neural networks is duplicated once again in a new generation where the same process is repeated. 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).

Genetic algorithms combine the flexibility of supervised learning and real time training aspect of reinforcement learning resulting in a smarter and more efficient adaptive impedance control system solution.

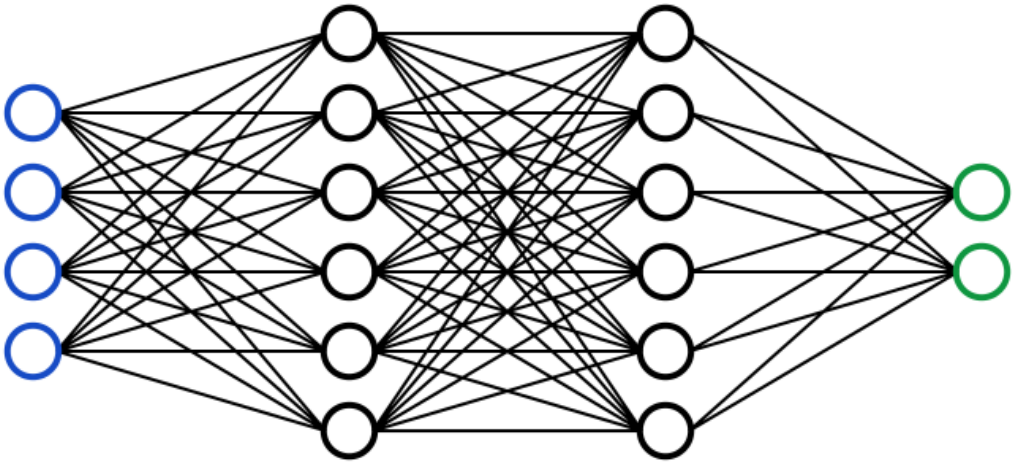

Figure 9: Deep Convolutional Neural Network [9]

2.4 Conclusions:

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 algorithms (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: Simulated Implementation (2 DOF Robotic Arm)

3.1 Introduction:

In this chapter we develop a simulation of an impedance controlled 2 DOF robotic arm in Python, 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 is injected into the impedance control included in the dynamic system as in Figure 10.

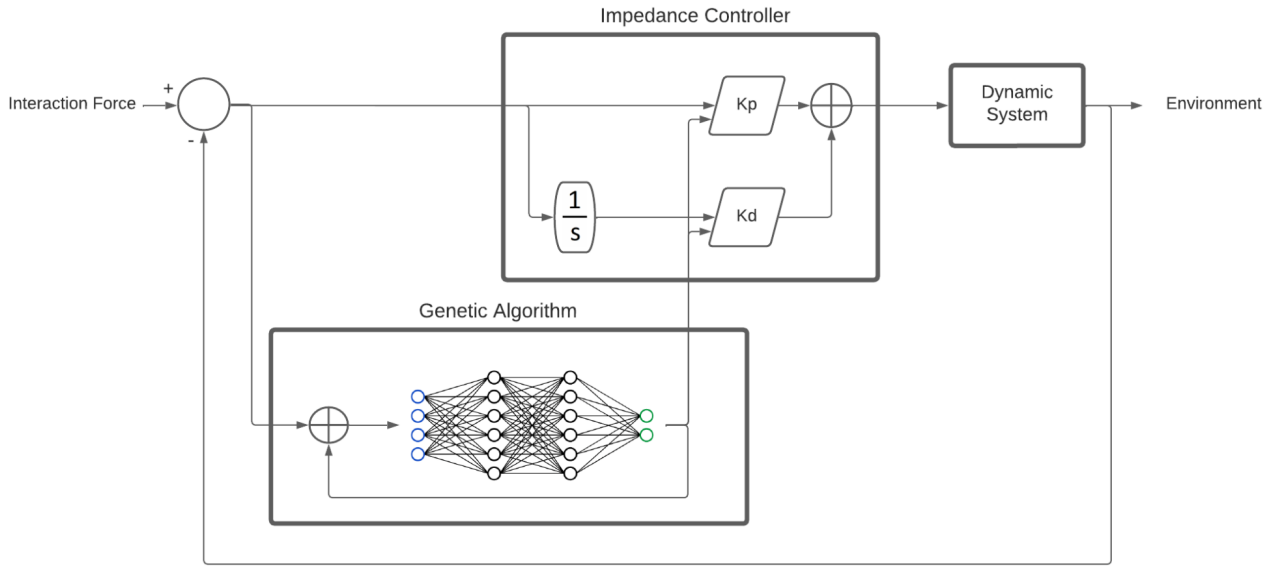

Figure 10: Training System Loop Schematic

3.2 Kinematics:

The kinematics of the robotic arm are the same to that by Tuna Orhanli [10] where the first link tilts by theta1 from the horizontal axis with length L1 and mass M1, and the second link tilts by theta2 beginning from theta1 of the first link with length L2 and mass M2.

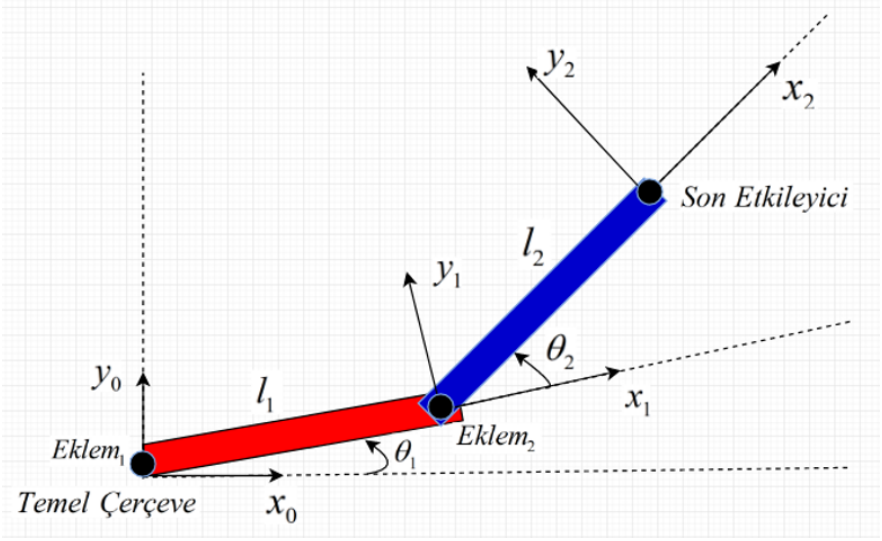

Figure 11: Kinematic System of 2 DOF Robotic Arm [10]

The forward kinematic of point P1 (x1, y1) is a Pythagorean relation between L1 and theta1 resulting as follows:

$$x1 = L1cos(the1) \quad (2)$$

$$y1 = L1sin(the1) \quad (3)$$

Furthermore, the kinematics of point P (x2, y2) being the end effector follows the same method

$$x2 = L1cos(the1) + L2cos(the1 + the2) \quad (4)$$

$$y2 = L1sin(the1) + L2sin(the1 + the2) \quad (5)$$

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 Vx1, Vy1, Vx2 and Vy2 are obtained by differentiating the relationships in equations (2), (3), (4) and (5) as follows:

$$Vx1 = -L1the1^{\bullet}sin(the1) \quad (6)$$

$$Vy1 = L1the1^{\bullet}cos(the1) \quad (7)$$

$$Vx2 = -L1the1^{\bullet}sin(the1) - L2sin(the1 + the2)\,(the1^{\bullet} + the2^{\bullet}) \quad (8)$$

$$Vy2 = L1the1^{\bullet}cos(the1) + L2cos(the1 + the2)\,(the1^{\bullet} + the2^{\bullet}) \quad (9)$$

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

$$V = [-L1the1^{\bullet} sin(the1) - L2sin(the1 + the2)\,(the1^{\bullet} + the2^{\bullet}),$$

$$L1the1^{\bullet} cos(the1) + L2cos(the1 + the2)\,(the1^{\bullet} + the2^{\bullet}),$$

$$0 \qquad] \qquad (10)$$

The relationship between the angular velocities and V is derived by factoring the angular velocities theta1dot and theta2dot in equation (8) into a 2x1 vector as follows:

$$V = [-L1sin(the1) - L2sin(the1 + the2)\,,\; -L2sin(the1 + the2),$$

$$L1cos(the1) + L2cos(the1 + the2)\,,\; L2cos(the1 + the2),\; * [\, the1^{\bullet}, the2^{\bullet}\,]$$

$$0 \qquad , 0 \qquad] \qquad (11)$$

Where the jacobian matrix is:

$$J = [-L1sin(the1) - L2sin(the1 + the2)\,,\quad -L2sin(the1 + the2),$$

$$L1cos(the1) + L2cos(the1 + the2)\,,\quad L2cos(the1 + the2)\;\;,$$

$$0 \qquad , \quad 0 \qquad] \quad (12)$$

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 System:

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.

The Langrangian method is used to obtain the differential equation of the dynamic system by first determining the kinetic and potential energies acting on the robotic arm as follows:

<u>Kinetic Energy</u>:

$$K1 = 1/2 * M1 * (Vx1^{2} + Vy1^{2}) \qquad (13)$$

$$K2 = 1/2 * M2 * (Vx2^{2} + Vy2^{2}) \qquad (14)$$

$$K = K1 + K2 \qquad (15)$$

Potential Energy: g is gravity

$$U1 = M1 * g * y1 \quad (16)$$

$$U2 = M2 * g * y2 \quad (17)$$

$$U = U1 + U2 \quad (18)$$

Lagrangian:

$$L = K - U \quad (19)$$

Dynamic System Equations:

In order to obtain the differential equations we must obtain the Euler equations as follows:

$$E_1 = dL/d\,the1 - d\,(dL/dthe1^{\bullet})/dt \quad (20)$$

$$E_2 = dL/d\,the2 - d\,(dL/dthe2^{\bullet})/dt \quad (21)$$

The result from which is factored by $the1,\ the2, the1^{\bullet},\ the2^{\bullet},\ the1^{\bullet\bullet}\ and\ the2^{\bullet\bullet}$ forming the dynamic system of the robotic arm consisting of:

$$M * [the1^{\bullet\bullet},\ the2^{\bullet\bullet}] + B + G = 0 \quad (22)$$

Where M is the 2x2 inertia matrix:

$$M = [(M1 + M2)L1^2 + m2 * L2^2 + 2M2 * L1L2cos(the2),\ M2 * L2^2 + M2 * L1L2cos(the2);\ M2 * L2^2 + M2 * L1L2cos(the2),\ M2 * L2^2\ ]; \quad (23)$$

And B is the coriolis and centrifugal forces 1x2 matrix:

$$B = [\ -M2 * L1L2 * (2 * the1^{\bullet} * the2^{\bullet} + the2^{\bullet\,2}) * sin(the2)\ ;\ -M2 * L1L2 * the1^{\bullet} * the2^{\bullet} * sin(the2)\ ] \quad (24)$$

And G is the gravitational forces 1x2 matrix:

$$G = [\ -(M1 + M2) * g * L1 * sin(the1) - M2 * g * L2 * sin(the1 + the2)\ ;\ -M2 * g * L2 * sin(the1 + the2)\ ] \quad (25)$$

The interaction force matrix Fd of shape 2x1 and the converted impedance control equation (1) 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).

$$Fd = [\,Fx;\ Fy\,] \quad (26)$$

Tuning the dynamic system's equation (22) into the following equation (27):

$$M * [the1^{\bullet\bullet}; the2^{\bullet\bullet}] + B + G = J^{T} * F_{Impedance} + Fd \qquad (27)$$

The impedance control's torque is considered an internal torque while the interaction force Fd is an external torque. In order to obtain the differential equations of the dynamic system, equation (27) is solved for $the1^{\bullet\bullet}$ and $the2^{\bullet\bullet}$ by multiplying both sides by the inverse of the inertia matrix M:

$$[the1^{\bullet\bullet}; the2^{\bullet\bullet}] = M^{-1} * (-B - G + J^{T} * F_{Impedance} + Fd) \qquad (28)$$

<u>Gravity Compensation</u>:

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:

If $T = J^{T} * F_{Impedance}$ (29) is the internal torque equation of the system then the gravity compensation equation is:

$$T = M * [the1^{\bullet\bullet}; the2^{\bullet\bullet}] + B + G - Fd \qquad (30)$$

This gravity compensation's torque (30) must be combined with the impedance control's torques (29) 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.

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 as seen in figure 12 which two DOF robotic arm on a realistic physics environment, a deep convolutional neural

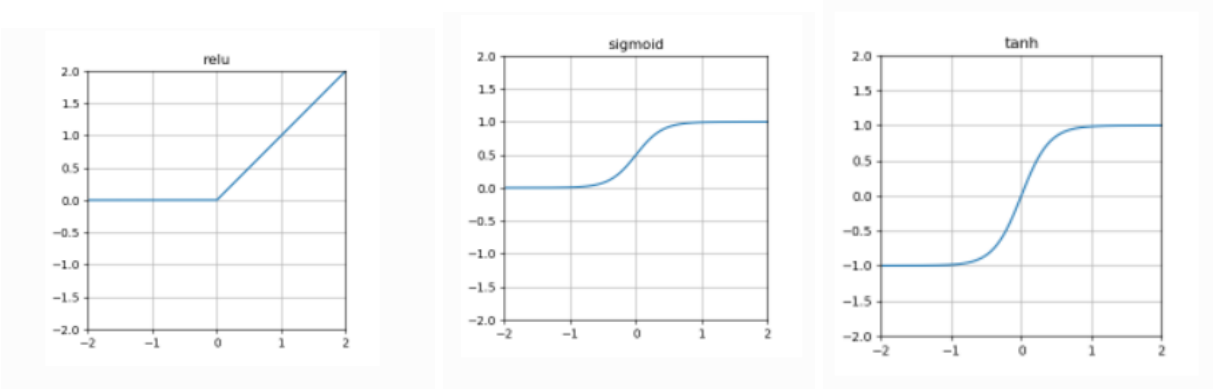

Figure 12: Common Machine Learning Cost Functions [11]

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.

3.5 Programming & GUI:

Matlab is a suitable platform for developing linear and non-linear dynamic systems, execution time is fast and reliable but lacks in graphical interactivity with the user such as canvases and events which are necessary in both a simulation and communication with the arduino in the physical implementation, genetic algorithms are also underdeveloped due to small community support.

Python is a well known programming language which offers both active support in genetic algorithms (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 within Python.

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

Library Imports:

As noticed in figure 13, the required libraries for the simulation 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.

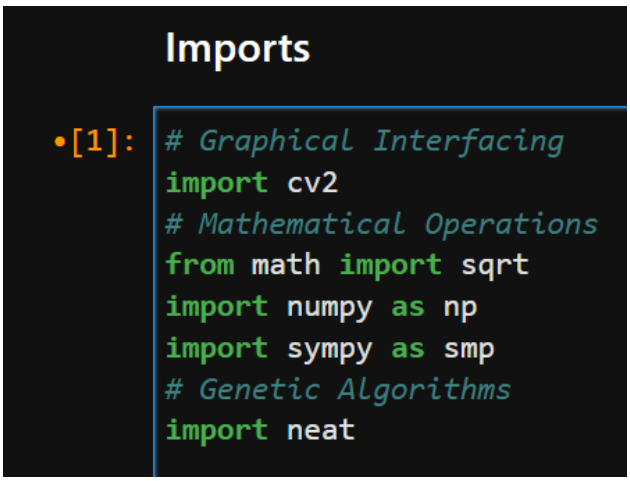

Figure 13: Imported libraries Code

Symbol Declaration:

As noticed in figure 14, the necessary symbols used in the dynamic system's differential equations and the impedance controller's symbols are declared using Sympy.

Figure 14: Declared Equation Symbols Code

Dynamic System:

As noticed in figure 15, the equations (2), (3), (4), (5), (23), (24), (25), (26) and (28) 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 15: Dynamic System, Jacobian & Forward Kinematics Code

Gravity Compensation:

As noticed in figure 16, the equation (30) 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 16: Gravity Compensation determination Code

Impedance Controller:

As noticed in figure 17, the equation (30) 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 17: Cartesian Impedance Controller Code

Symbolic Expression to Function Conversion:

As noticed in figure 18, the aforementioned equations are converted from symbol expressions into callable Python functions for our simulation.

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 18: Symbolic to Function Conversion Code

Graphical Interface:

As noticed in figure 19, the graphical interface is developed using OpenCV and the frames are declared using Numpy as matrices of pixels where the theta 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 simulation 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 Fd are displayed in the top left of the simulation.

```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 19: Graphical Interface Function Code

Impedance Controller Execution:

As noticed in figure 20, 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 20: Impedance Controller Function Code

Dynamic System Execution:

As noticed in figure 21, 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 theta angles, the period dt is predefined which allows for real time integration alongside the simulation.

```
    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 21: Dynamic System Execution Code

Genetic Algorithm:

As noticed in figure 22, 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 22: Genetic Algorithm Initialization Code

As observed in figure 23, 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.

```python
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 23: 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 (test values are discussed in chapter 5)
- **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 24, 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 24: Time limit Code

3.6 Conclusions:

Dynamic system simulation is a lengthy process involving realistic period selection and differential equation execution.

Using Python and Jupyter Notebook, we were able to develop the simulation and the graphical interface which will be useful in chapter 5 and experimentation and results, analyzing the interaction force based on time change is essential for an evaluation.

Simulating real objects using dynamic systems allows for grasping the realistic results from applying a revolutionary controller with little capital. However, once the simulation is proven to perform acceptably, a real implementation can be confidently implemented.

4. Chapter 4: Experimentation

4.1 Introduction:

In chapter 4, we will be building 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.

Servo Motors:

For a robust 2 DOF robotic arm, the smaller SG90 and MG90s servos seen in figures 25 and 26 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.

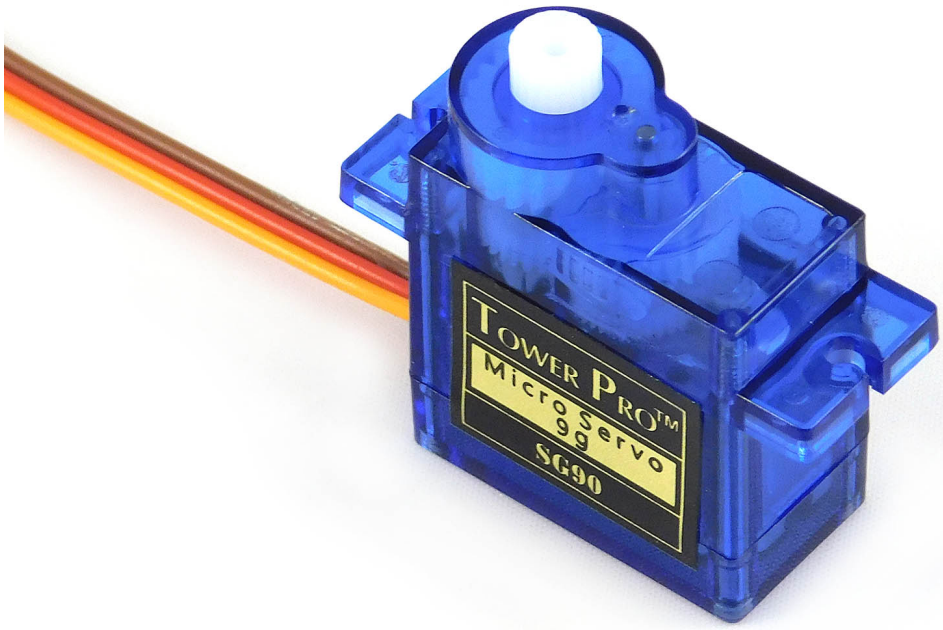

Figure 25: SG90 Servo Motor [12]

Figure 26: MG90S Servo Motor [13]

The MG996R robust servo motors observed in figure 27 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 27: MG996R Servo Motor [14]

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 28 senses up to only 10NM with a price tag of 825$ for one single torque sensor, the cheapest variant seen in figure 29 is still at a minimum of 200$ for a single unit.

Figure 28: Torque Sensor [15]

Figure 29: Cheaper Torque Sensor [16]

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 30, this sensor offers a range of measurable forces from a minimum of 0.1kg up to a maximum of 10kg. Its minimum required physical strain of 0.1kg 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.

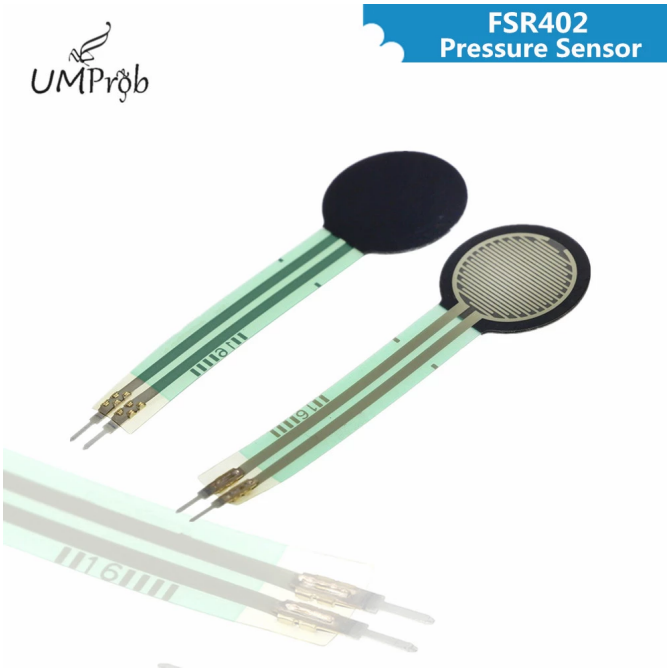

Figure 30: FSR402 Flex Pressure Sensor [17]

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 31 is used in this experiment.

Figure 31: Arduino Nano Microcontroller [18]

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 32.

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 32: Creality Ender 3 Pro V2 [19]

4.3 Circuit Diagram:

The mentioned materials are mounted in the following configuration seen in figure 33, 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 5kohm in order to reduce the current draw by the sensors as they do not require high current.

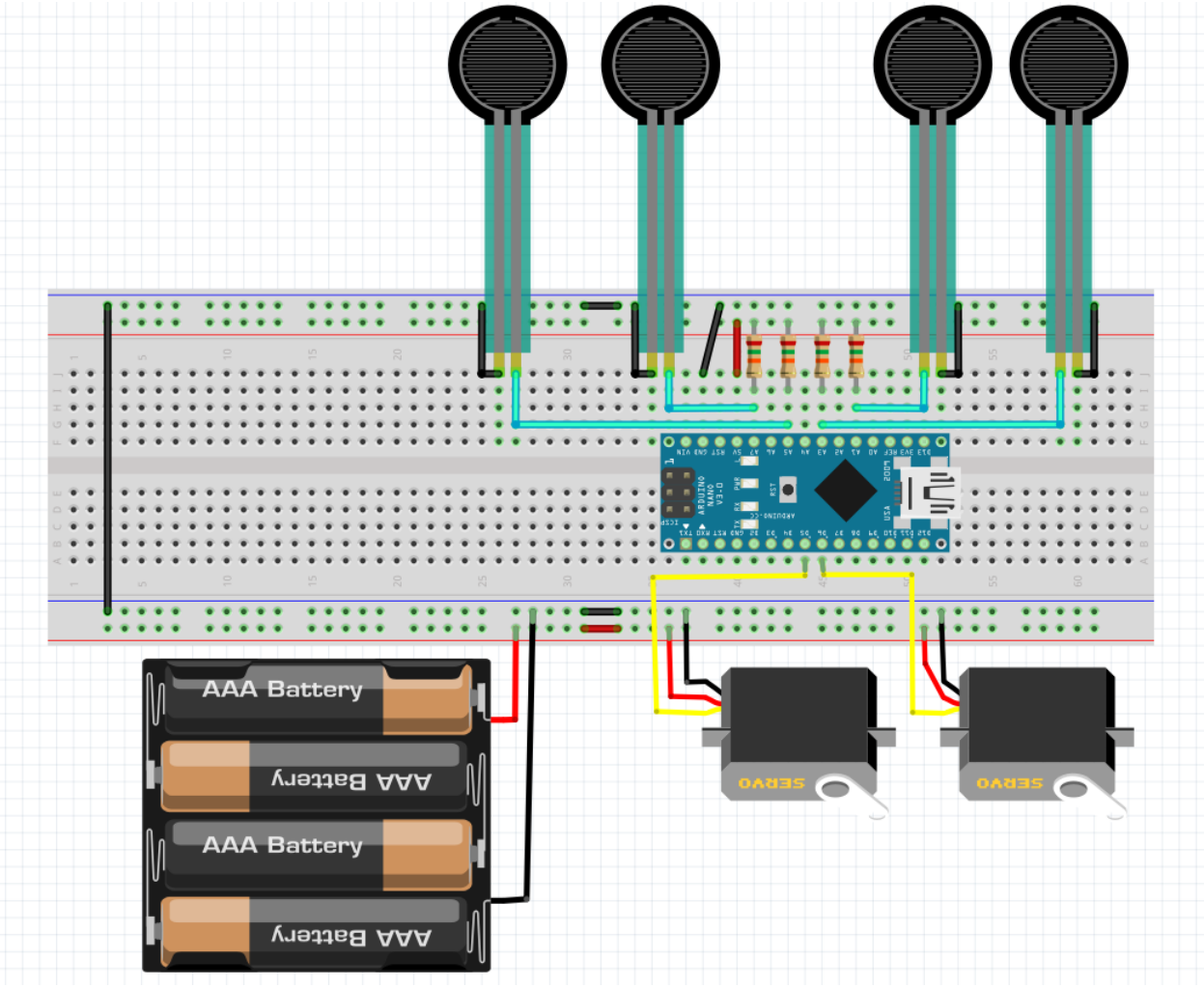

Figure 33: 2 DOF Robotic Arm Circuit Diagram

4.4 3D Modeling & Printing:

1 - 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.

Main Base:

As observed in figure 34, the main base is a circular inboxing which houses the first MG996R servo motor, it is responsible for varying theta 1 from our dynamic system.

The four long extending feet in figure 35 serve for stability and to keep the robotic arm upright when fully extended horizontally. The four L shape pinners also seen in figure 35 serve

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

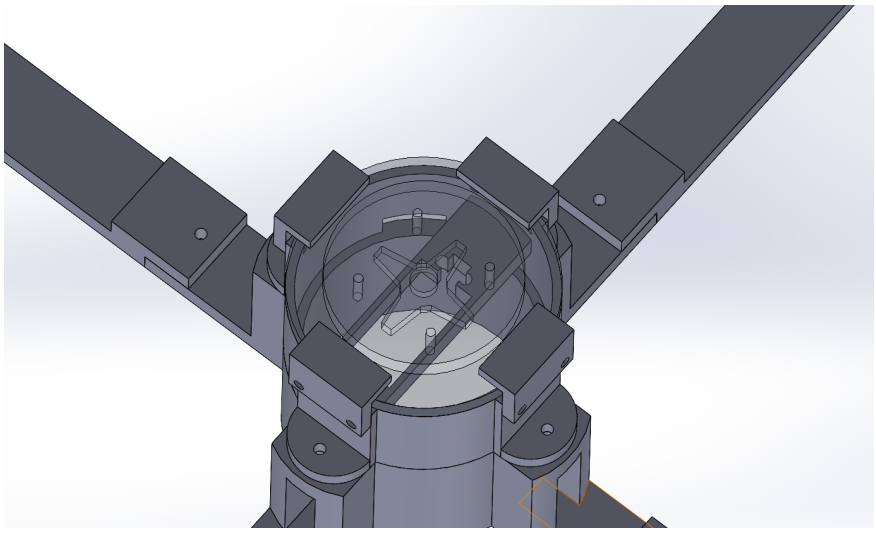

Figure 34: Main Base Housing

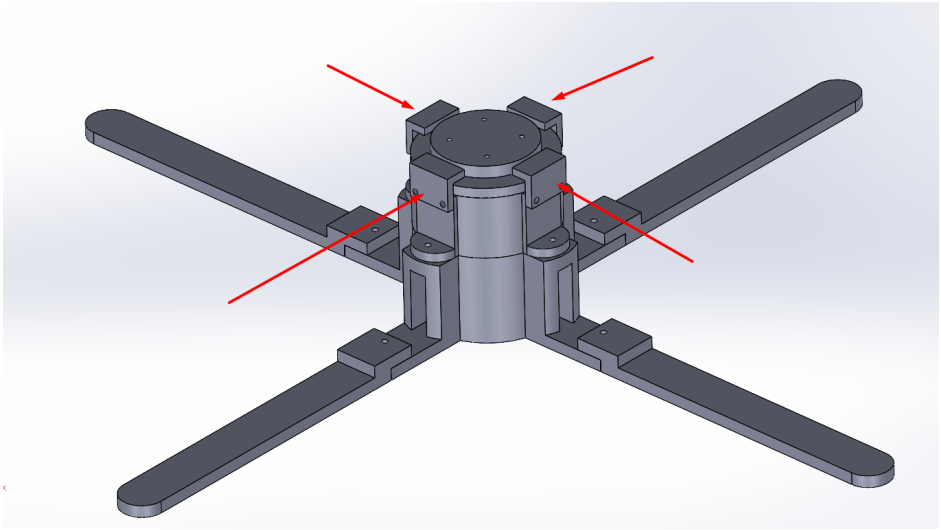

Figure 35: Main Base Stability Structure

Forearms:

In figure 36, 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.

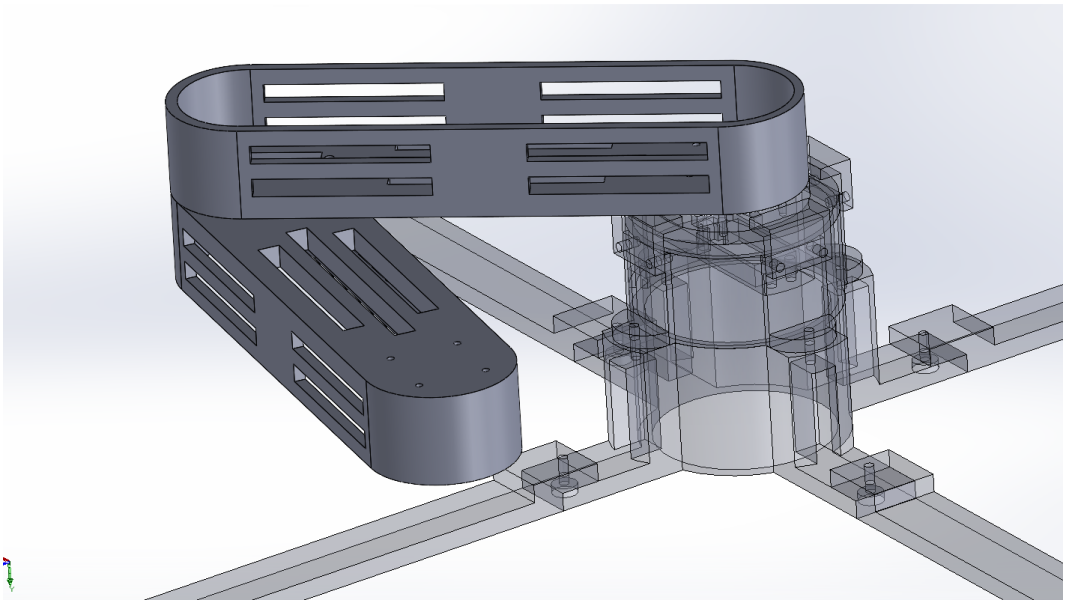

Figure 36: Two links of robotic arm

End Effector:

As observed in figure 37, 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 38.

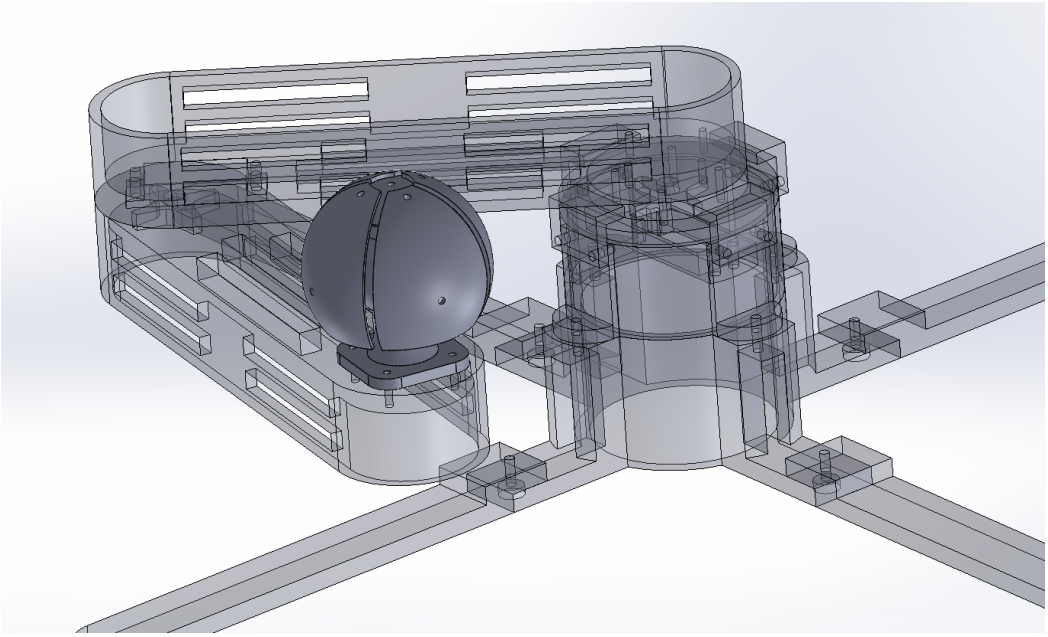

Figure 37: End Effector Positioning

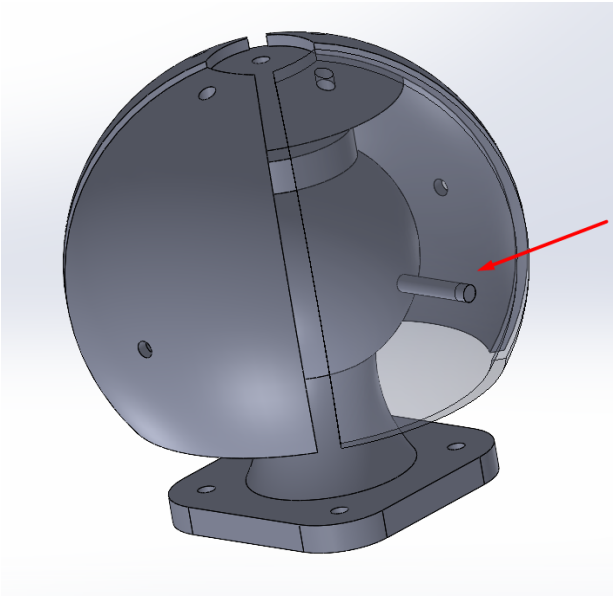

Figure 38: End Effector Mechanism

2 - 3D Printing:

The 3D printing of the parts required little support as it was designed for ease of 3D printing as observed in figures 39, 40 and 41. 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.

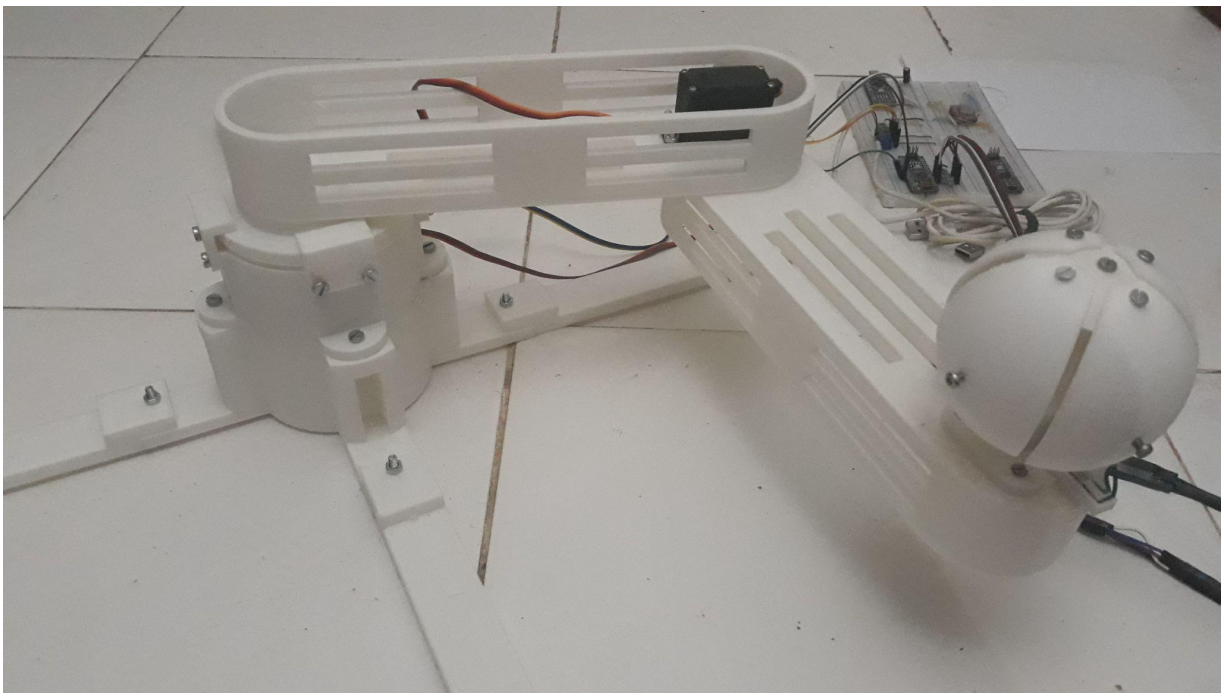

Figure 39: 2 DOF Robotic Arm

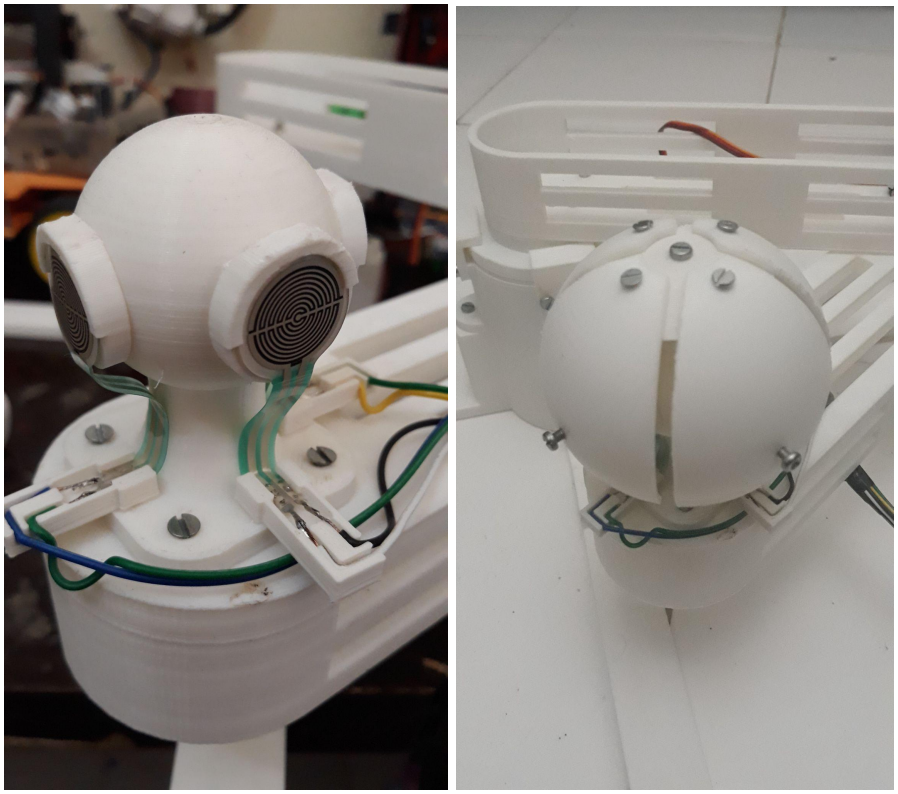

Figure 40: End Effector of 2 DOF Robotic Arm

Figure 41: MG996R Servo Motors Installation

4.5 Programming:

The neural network which is trained on the simulation in chapter 3 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 42, 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.

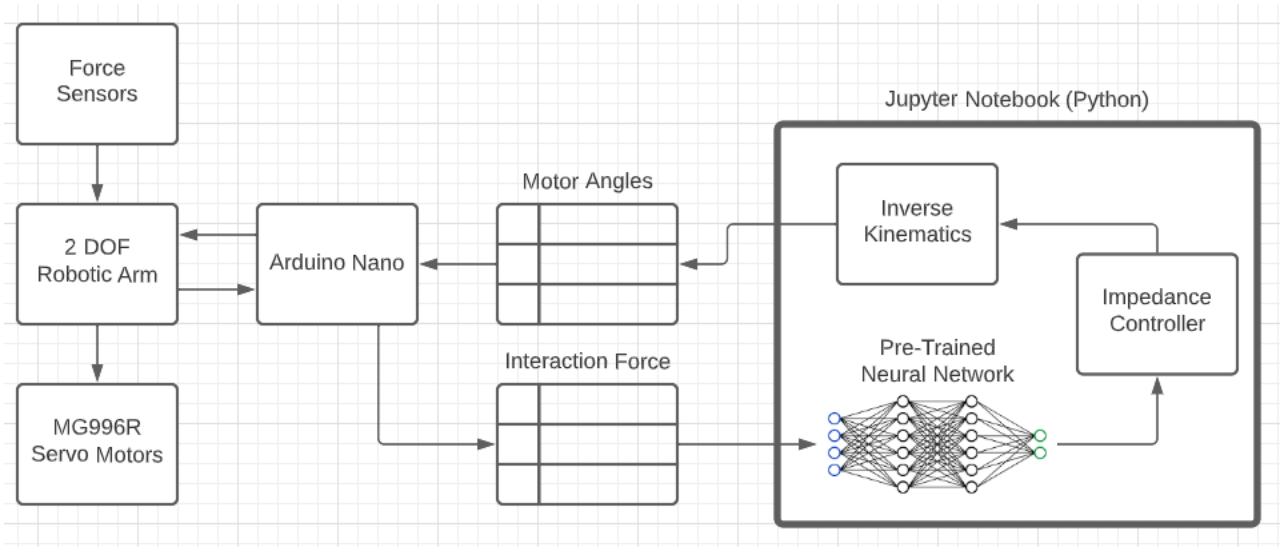

Figure 42: 2 DOF Robotic Arm Diagram

1 - 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 43, 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 43: 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 44.

```
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 44: 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 45.

```
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 45: Go to Coordinates Function

2 - Python:

Serial Communication:

Python uses the pyserial library which allows to send the decided motor angles to Arduino via Serial Communication Protocol, and receive force sensor reports as in figure 46.

```
# 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 46: Serial Communication Function

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 47.

```
# 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 47: Impedance Controller Formula

4.6 Conclusions:

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.

After having done both the simulation and physical implementations, how will these two systems perform? And does the neural network perform in hardware just as well as in software with few sensors only? That's what will be discussed in chapter 5.

5. Chapter 5: Results & Discussion

5.1 Introduction:

In this chapter, we perform a test on both the simulated robotic arm and the physical robotic arm and establish a comparison.

5.2 Simulation Results:

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

First case: (Fd=50N or approximately 5.1kg force) 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 48, 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 Kp and Kd, 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).

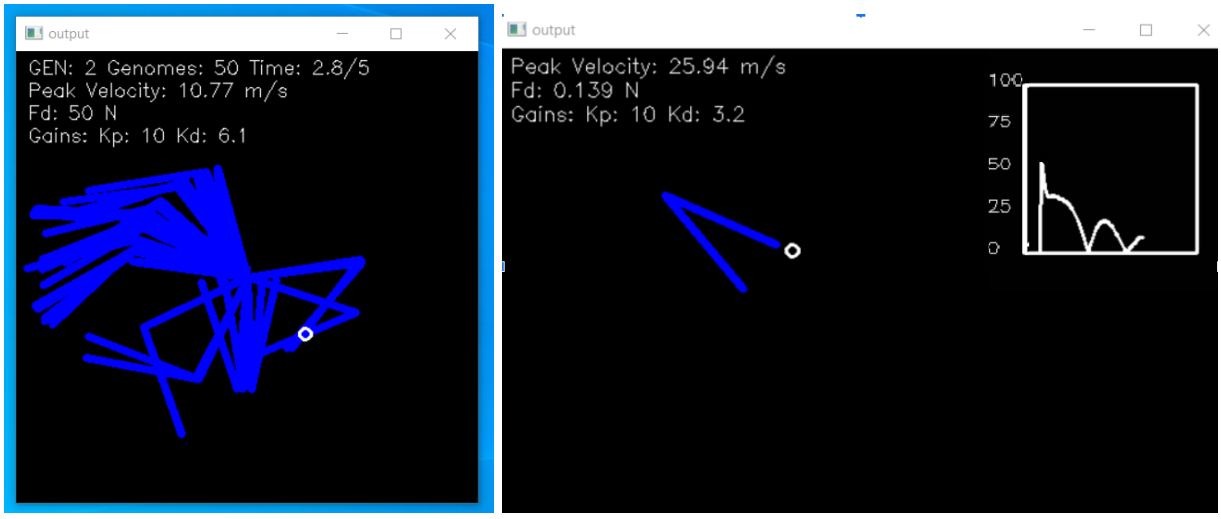

Figure 48: Test case 1 2nd generation training & transient response

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 0, the initial interaction force of 50 N is introduced. As seen in the graph, the system is highly oscillatory 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 49

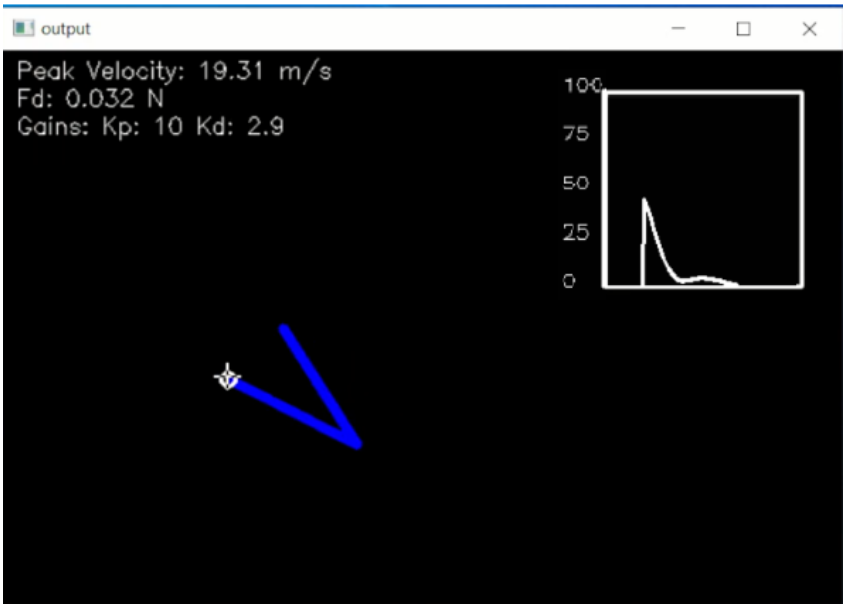

Figure 49: Test case 1's last surviving genome 50 generations of training

The interaction force decays rapidly as 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 50.

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.

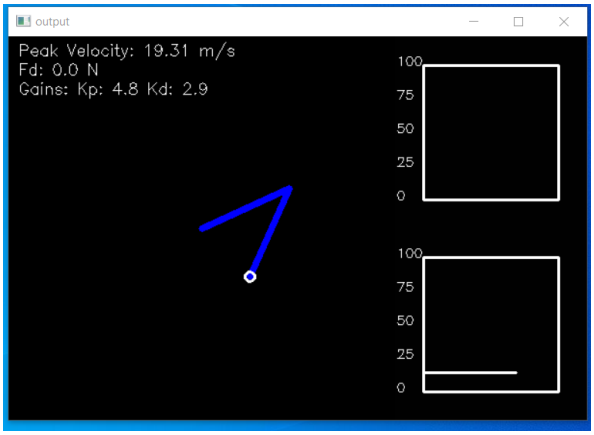

Figure 50: Test case 1 steady state

Second case: (Fd=5N or approximately 0.51kg force) The second test case is applied in a similar manner as observed in figure 51, 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.

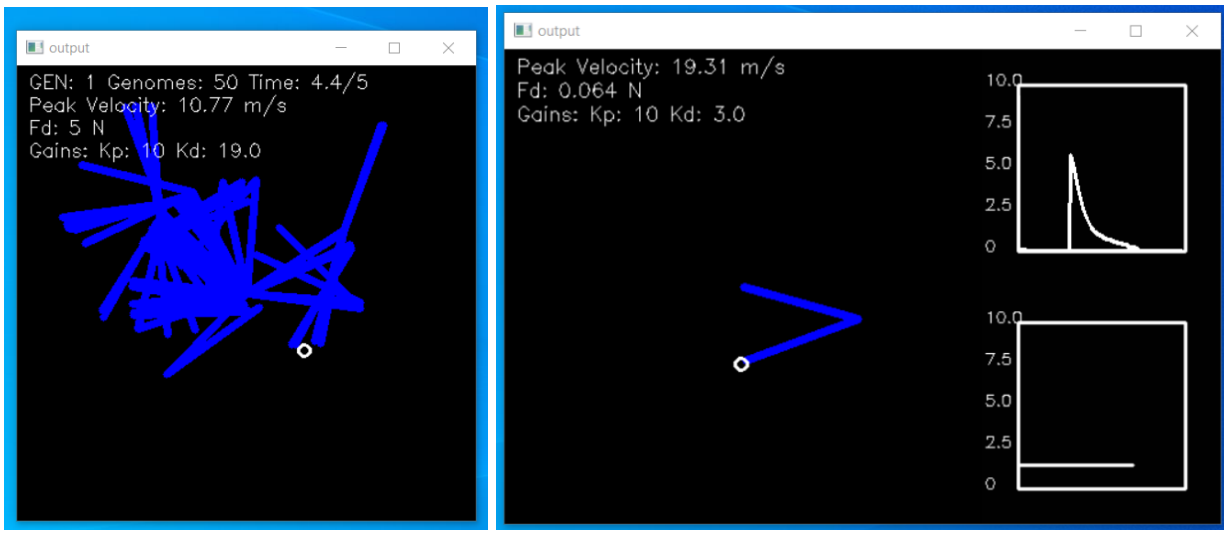

Figure 51: Test case 2 training & transient response

Third case: (Fd=0.5N or approximately 0.051kg force) The third test case is also applied in a similar manner as observed in figure 52, 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.

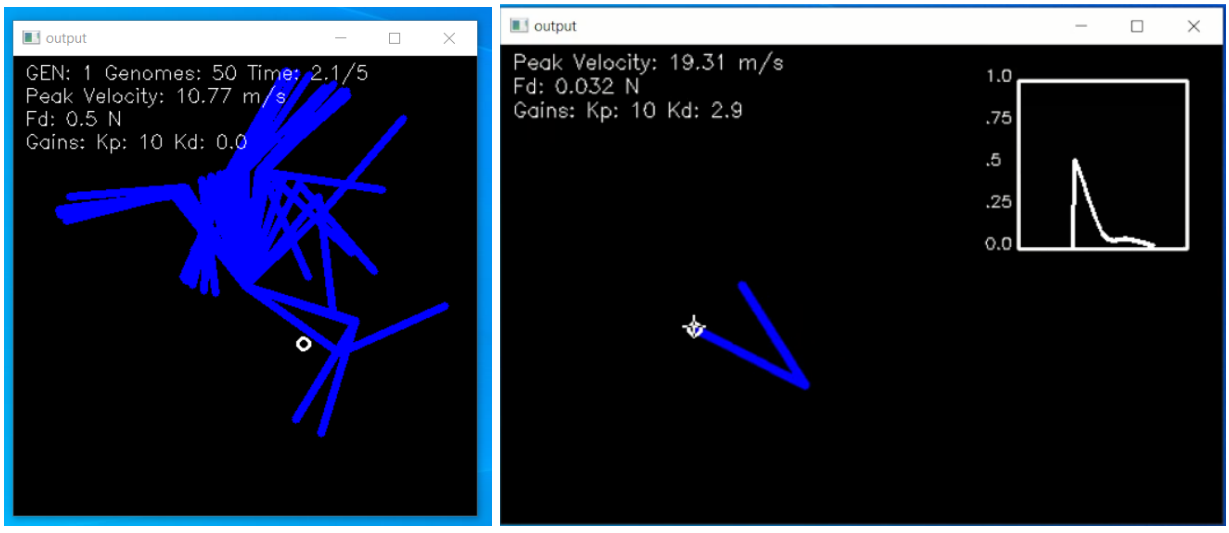

Figure 52: Test case 3 training & transient response

5.3 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 53.

First case: (Fd=50N or approximately 5.1kg force) As observed in figure 53's force graph in the top right hand side, the sensor is objected to 51% of its maximum allowing pressure (5.1kg of force equivalent to 50 Newton), 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.

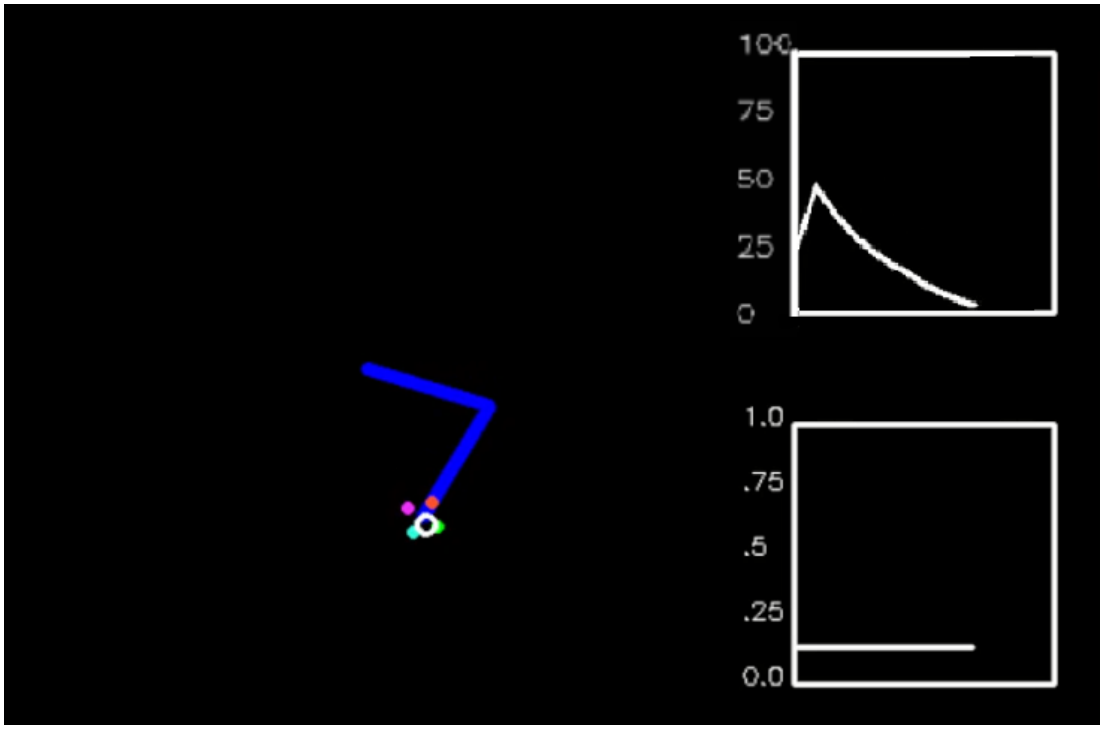

Figure 53: Test case 1 transient response

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

Second case: (Fd=5N or approximately 0.51kg force) In a similar manner, the robotic arm is influenced by approximately 0.51kg of force (equivalent to 5N of applied force), the robotic arm displays comparable transient response with smooth and non-stiff interaction as observed in figure 54.

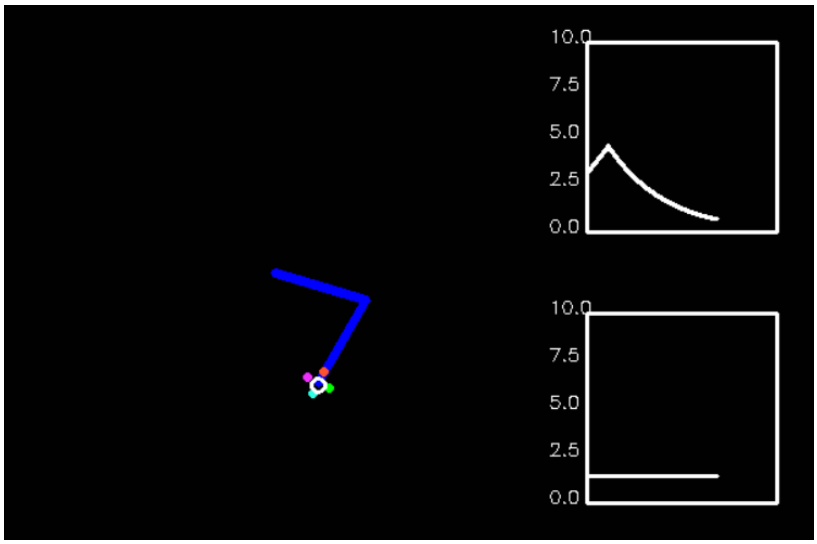

Figure 54: Test case 2 transient response

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

Third case: (Fd=0.5N or approximately 0.051kg force) In a similar manner , the robotic arm is influenced by approximately 0.051kg of force (equivalent to 0.5N of applied force), the robotic arm displays no reaction as observed in figure 55, this is due to the minimum of 0.1kg required interaction force on the FSR402 sensors used in this operation.

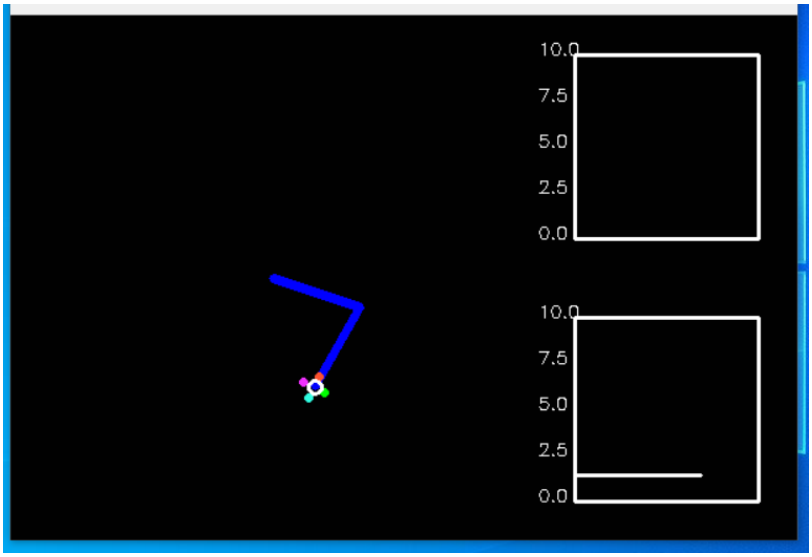

Figure 55: Test case 3 transient response

5.4 Conclusions:

- 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 (0.051kg), 5N (0.51kg) and 50N (5.1kg) of applied force and it performed adequately.
- Due to the minimum of 0.1kg (0.98N) of applied force implied by the affordable FSR402, the third test case is unable to be tested on the real robotic arm.
- 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).

References:

[1]https://healthcaremarketexperts.com/en/news/joanna-szyman-for-pmr-robotics-the-future-of-surgery/

[2]https://www.bbc.co.uk/bbcthree/article/4a466b39-8b03-4717-bde8-822760b430fa

[3]https://www.theverge.com/2021/2/2/22261932/boston-dynamics-spot-enterprise-self-charging-scout-web-based-control-software-robotic-arm

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

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

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

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

[8]https://www.researchgate.net/figure/The-main-types-of-machine-learning-Main-approaches-include-classification-and_fig1_354960266

[9]https://medium.com/swlh/an-introduction-to-neural-networks-de70cb4305f9

[10] Tuna Orhanli (2022). Forward Dynamics of Planar 2-DOF Robot Manipulator (https://www.mathworks.com/matlabcentral/fileexchange/69756-forward-dynamics-of-planar-2-dof-robot-manipulator), MATLAB Central File Exchange. Retrieved May 21, 2022.

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

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

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

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

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

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

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

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

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