RoboCon

Gaming Robot

Faisal Fazal-Ur-Rehman

Student no. 10538828

Supervisor: Dr Mario Gianni

PROJ324 Individual Project

School of Engineering, Computing, Electronics & Mathematics

University of Plymouth

May 2020

Abstract

The use of robots has become inevitable for industrial use, but it also has a big market for commercial, research and entertainment use. This report is concerned with deriving a concept by design and experiment of a complete robotic system that is low-cost, replicable and can play board games with human opponents. It provides a solution by design as evidence to minimise the cost of the hardware, ease reproducibility, maximise robot's structural durability and minimise hardware modification for the addition of new games to the system. This report provides a scalable software solution by use of distributed computing between PC and Raspberry Pi. This report alludes the use of computer vision to capture the physical state of the game, It provides solution for locomotion using kinematics and straight-line trajectory and it explores machine learning algorithm to provide solution for the game in virtue of training, analysing and implementation of Deep Q Network.

Index

Glossary

DQN	**Deep Q Network**
RPI	**Raspberry Pi**
PC	**Personal Computer**
ML	**Machine Learning**
UR	**Universal Robots**
GUI	**Graphical User**
RR	**Revolute Revolute**
NN	**Neural Network**
QL	**Q Learning**
MCU	**Microcontroller Unit**

1 Introduction

The board game industry had an estimated market of 7.2 billion US dollars in 2017 and is expected to increase by 4.8 billion US dollars making it 12 billion US dollars by 2023 [1]. Although this is a growing market there are no low-cost commercial robots in the market. This however does not mean that there is no activity in this area. Currently, majority of the robots used for playing board games are expensive industrial and commercial robots or low-cost robots which are specifically designed for one game and therefore are limited to a single game only.

Some of the robots used for this purpose are the NAO robot, high-end robot manipulators like the Universal Robots and some individual projects like student projects or homemade robots.

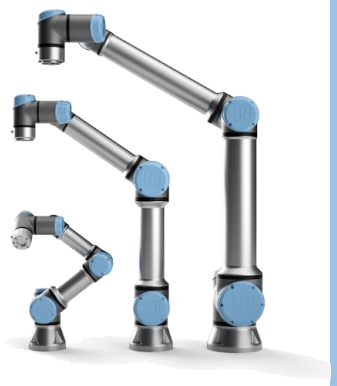


Figure 1 – Universal Robots
source: https://www.active8robots.com/news/cobot-month-ur-e-series/

Universal Robot's UR series is a collaborative robot (Cobot) series. The UR series has 4 models, UR3(e), UR5(e), UR10(e) and UR16(e), where UR3 is the cheapest and currently priced at nearly £17500.

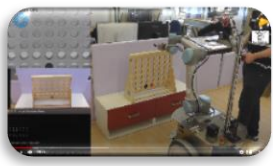
These robots (cobots) are very impressive in their performance, precision and scalability. They have been used across the industries from food and agricultural packaging to scientific and research purposes [3].

A UK based company Moley Robotics have incorporated UR robots for automated kitchens [4]

These cobots are very impressive but they do not fall anywhere near the low-cost range and neither are they designed to be.

Videos of games with UR Robots

Connect 4



Blackjack

Jenga

Figure 2 – NAO Robot
source: https://www.softbankrobotics.com/emea/en/nao

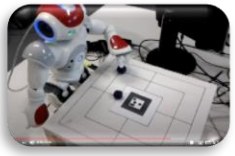
NAO Robot is an autonomous, programmable humanoid robot developed by Aldebaran Robotics and currently owned by SoftBank Robotics [2]. NAO Robot is a fantastic robot for research and learning. It could also be a great robot to play board games against human opponents. However, NAO Robots are expensive, currently priced at around £6500 which is still a little excessive to be classed as affordable for students, hobbyists or an average person.

Videos of games with NAO Robots

Connect 4

Tic Tac Toe

Nine Men's Morris



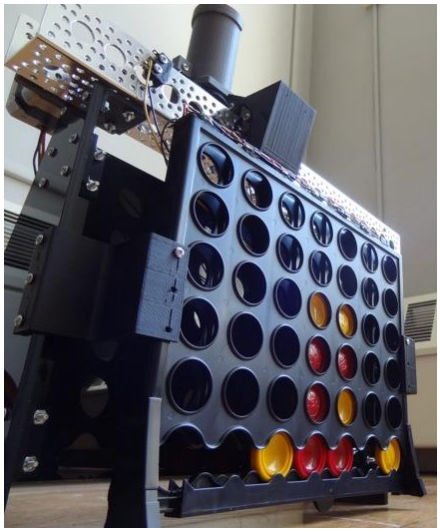


Figure 3 – Connect 4 Robot by MIT Student
source:http://patrickmccabemakes.com/hardware/Connect_Four_Robot/

Connect 4 Robot made by MIT student for a “Microcomputer Project Laboratory” course. The design of the robot is compact, requires only 2 motors and runs very smoothly. The algorithm used to solve the game is Negamax algorithm which is a variant of the Minimax algorithm. The Negamax algorithm searches the game tree to a specific depth. It does this recursively by calling the Negamax algorithm on the children of each node in the tree [5].

The robot has no vision system and makes use of photo interrupter sensors to detect human player’s moves [5]. The photo interrupters emit light from one end and receive it from the other end. Any light blockage triggers a switch. This is a robust system to detect human opponent’s moves but is not extendable to other games.

This is a great robot to play connect 4, however it was specifically designed to play connect 4 only and cannot be extended to play other games.

Video

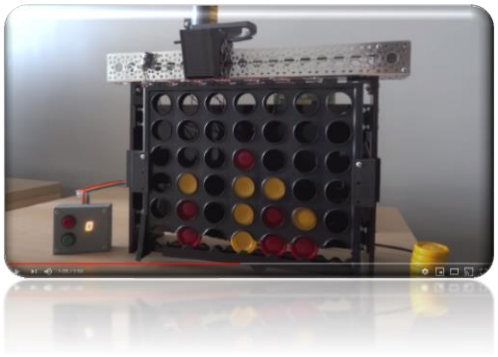


Figure 4 – Chess Robot [6]

This homemade chess robot is a cartesian robot. Like a 3D printer, this robot has three linear joints, one in each x, y and z plane. This makes locomotion models simpler than robots with revolute joints as there is no need to solve kinematics.

The robot makes use of reed switches to detect the 64 states on the chess board. Reed switches are electromagnetic switches that get activated when a magnetic field moves towards the switch [6].

The design is fantastic for the game in question and is the best in terms of cost and scalability in this list so far. However, scaling a cartesian robot still requires modification to most of its structure, e.g. to play standard size Connect 4 game both legs would require adjustment to increase the height so that the board would fit. For a Monopoly game the base and the top panel would require adjustment as a Monopoly board is much bigger than a board of chess.

Video

The robots listed above all solve only one of the two problems, they are either low-cost and replicable or they can play various board games with little modifications to the hardware. This is where RoboCon wins. RoboCon is a proof of concept in which the Connect 4 game is what the system needs to solve. Connect 4 was chosen as it is a complex game to solve with 4,531,985,219,092 total number of possible situations after n turns [7] but not as complex as Chess or Go. It is also one of the few board games with an upright board.

The next section is the design section, which is the main body of the report and is divided into two parts, Hardware and Software which then divides into further sub-sections.

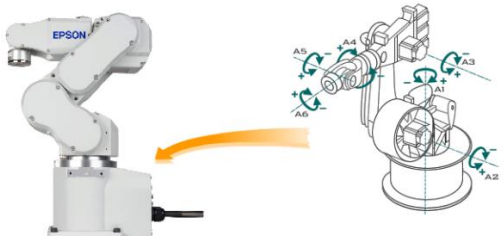
2 Design

2.1 Hardware

The hardware design for this project had to satisfy some challenging requirements. These requirements arise due to the nature of the project, it requires the robot to be large enough in size so that it would be able to reach far enough for larger board games like monopoly and high enough to be able to reach the top of upright boards like connect 4. The robot's design must also be able to accommodate various board games with very little to no hardware adjustments. The robot would require sensor(s) that can detect the state of various games i.e. it must not be specific to a game. The robot must also be reasonably durable. The challenging part however is to achieve all of this with a small budget. Larger structures are heavier and require torquier more expensive motors, whereas smaller structures will have smaller workspace. Weaker motors with large structures in an articulated type robot could work with low-cost doubled up motors but would greatly affect the durability of the motors as the motors would almost always be under strain.

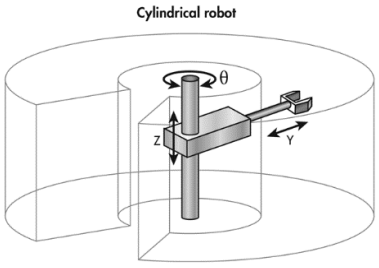
Some of the known robot solutions / designs are as follows:

- **Articulated**
  Consists of 2 or more revolute joints. The arm is connected to the base with a twisting joint. The links are connected by revolute joints.



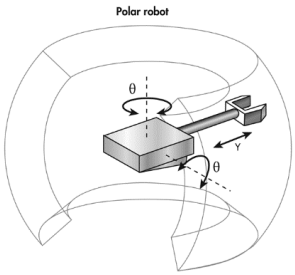
source: http://www.robotpark.com/academy/all-types-of-robots/stationary-robots/robotic-arms-articulated-robots/

- **Cylindrical**
  Consists of at least one revolute joint and one prismatic joint. The arm is connected to the base with a twisting joint. The workspace of the robot is cylindrical.



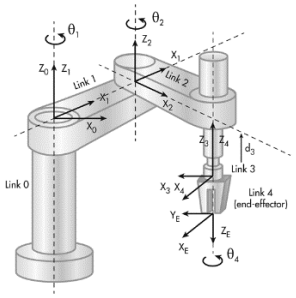
source: https://www.machinedesign.com/markets/robotics/article/21835000/whats-the-difference-between-industrial-robots

- **Polar**
  The arm is connected to the base with a combination of two perpendicular revolute joints and one linear joint. This formation makes a spherical workspace.



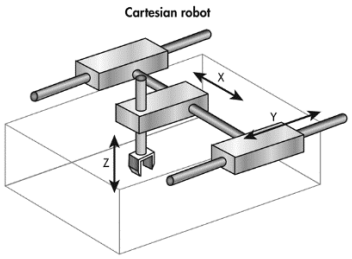
source: https://www.machinedesign.com/markets/robotics/article/21835000/whats-the-difference-between-industrial-robots

- **SCARA**
  Consists of two or more revolute joints that are all in parallel formation and one linear joint connecting the arm with the end-effector. This formation makes a cylindrical workspace.



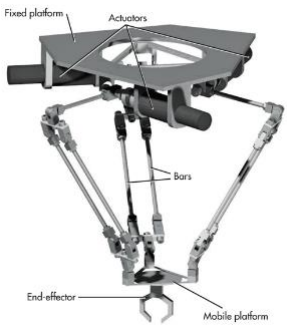
source: https://www.machinedesign.com/markets/robotics/article/21835000/whats-the-difference-between-industrial-robots

- **Cartesian**
  Consists of three linear joints one in each axis, x, y and z. The end-effector usually has a revolute joint. This formation covers a rectangular workspace.



source: https://www.machinedesign.com/markets/robotics/article/21835000/whats-the-difference-between-industrial-robots

- **Delta**
  Consists of three revolute joints all connected at the base of the arm. Forms a parallelogram between the three links. All three links are connected at the base and the end-effector. This formation creates a dome-shaped workspace.



source: https://www.machinedesign.com/markets/robotics/article/21835000/whats-the-difference-between-industrial-robots

Let us analyse the designs against our requirements:

- **Articulated**
  The design of this robot requires the motors to bear the weight of the links. This might work by adding gear cogs between the motors and the joint.

- **Cylindrical and Polar**
  Both of these designs would restrict the workspace of the robot by the length of the linear actuator and the longer the length of these actuators, the more expensive they are.

- **SCARA**
  This design can support the weight of the arm with minimal load on the motors. This is due to the parallel formation of the joints which allows the weight to be held by a shaft and bearings combination which takes almost all the weight off the motors when the robot is at rest. This increases the life of the motors, making the robot more durable and it allows the

use of low-cost, low-power and less torquier motors. The workspace of this robot is limited by the length and height of its links which can be modified to a required length / height.

- **Cartesian**
  The structure of this design can hold almost all its weight. However, this design would require the game board to be placed right under it. When playing a game like monopoly with a group of people and a robot, it would be preferable that the robot is not in the way all the time. This design also requires the robot's resting surface (e.g. the table) to be the size of its workspace regardless of the game, e.g. to play a game of chess would require the same size table compared to a game with a larger board. Even if these draw backs were ignored, what if there was a new game that needed to be added to the system, but this new game had a larger board than the current workspace of the robot. This would require not only the panels of the robot, but the linear actuators (or the worm gear used for the linear actuator) to be modified. If we compare this to the modification that we would require for a SCARA robot, the SCARA robot would only need amendment to its links, or if we can get away with it, to only one of its links.

- **Delta**
  The weight of this robot is almost always held by the motors. This design also requires an over the top base panel in the centre of its workspace. For this project, it would be preferable to avoid this.

After considering the options listed above, it was decided that the SCARA robot was the best solution for this project. However, the SCARA design defines the main arm only. The end-effector and, when required, the game fixture needs to be designed separately to complete the system. This is an advantage if we can design the joints to allow easy replacement of end-effector and the game fixture for different games. For this reason, the design of RoboCon is divided into three parts, the main arm, the end-effector and the game fixture.

2.1.1 RoboCon Design 1

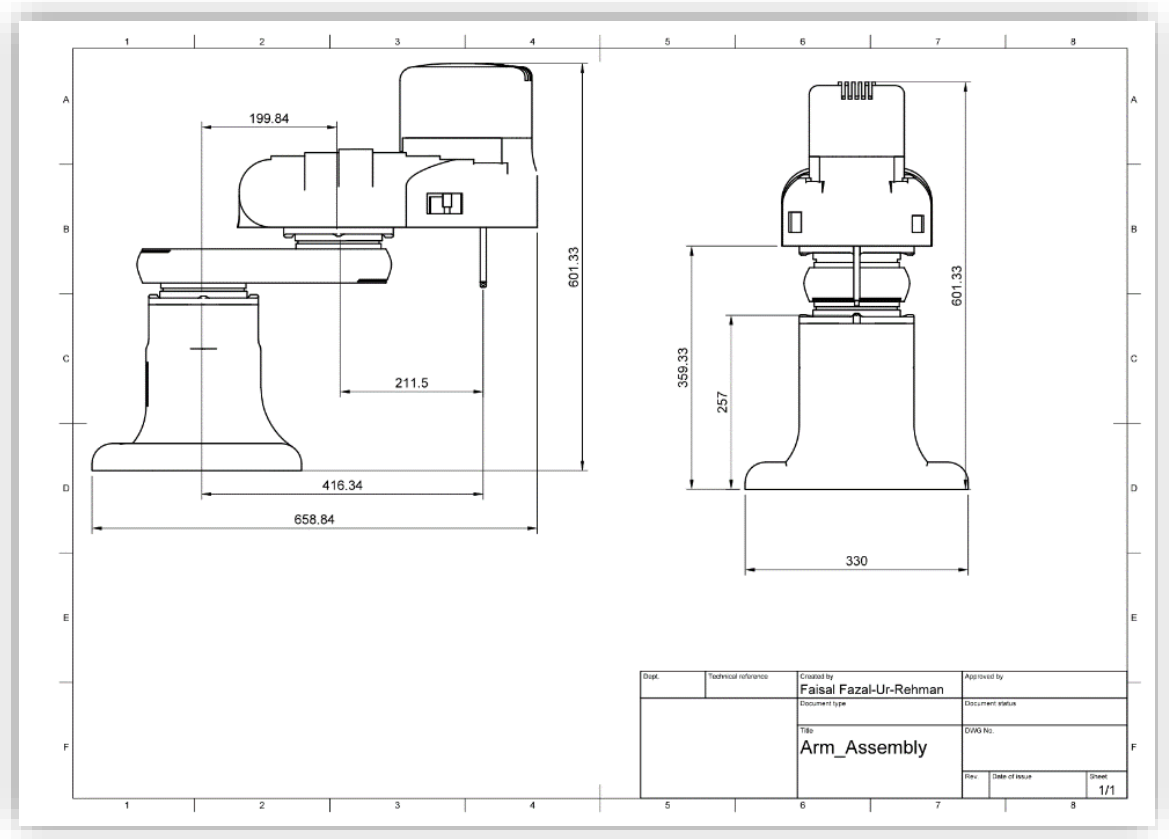


Figure 5 – Design 1 Main Arm Drawing

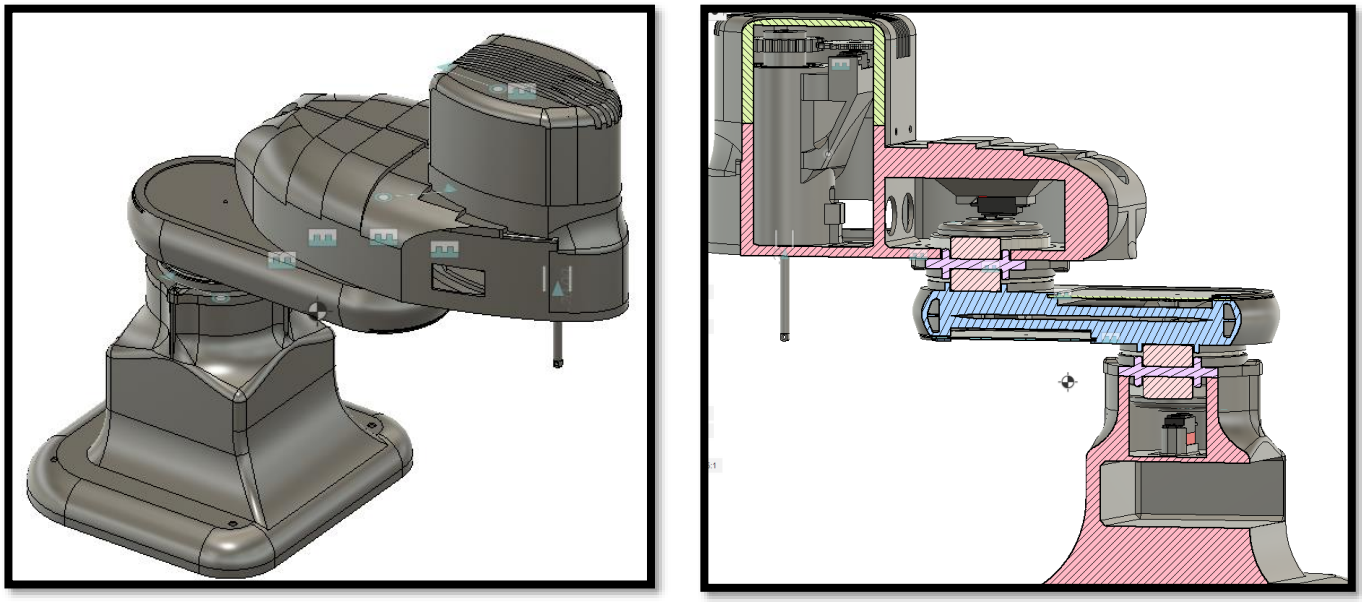


Figure 6 – Design 1 Main Arm

This was the first design for the main arm, but it became apparent, before printing it, that this was unnecessarily bulky. However, this provided some good ideas of what to take forward in the next design and what to avoid.

2.1.2 RoboCon Design 2

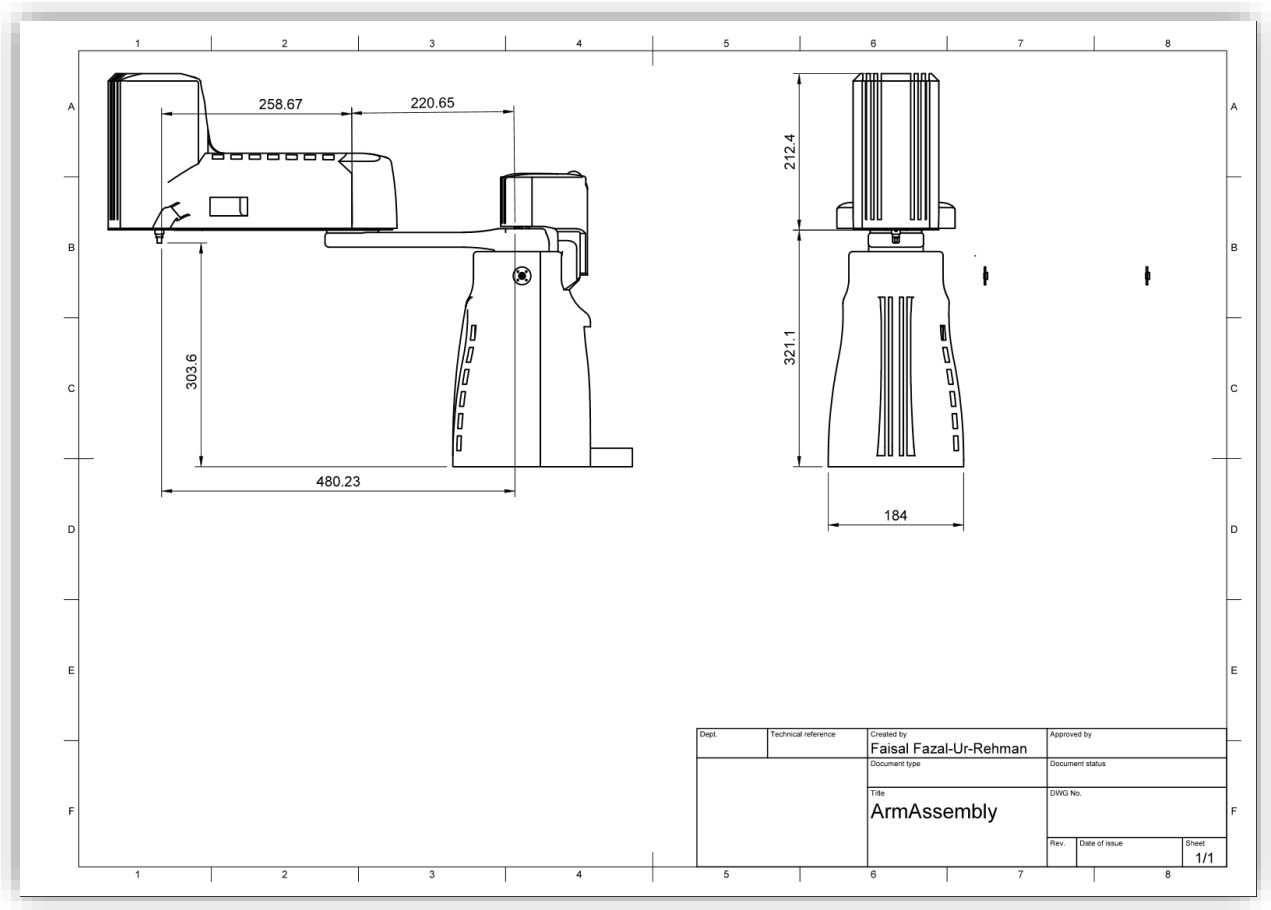


Figure 7 – Design 2 Main Arm Drawing

Figure 8 – Design 2 Main Arm with Covers

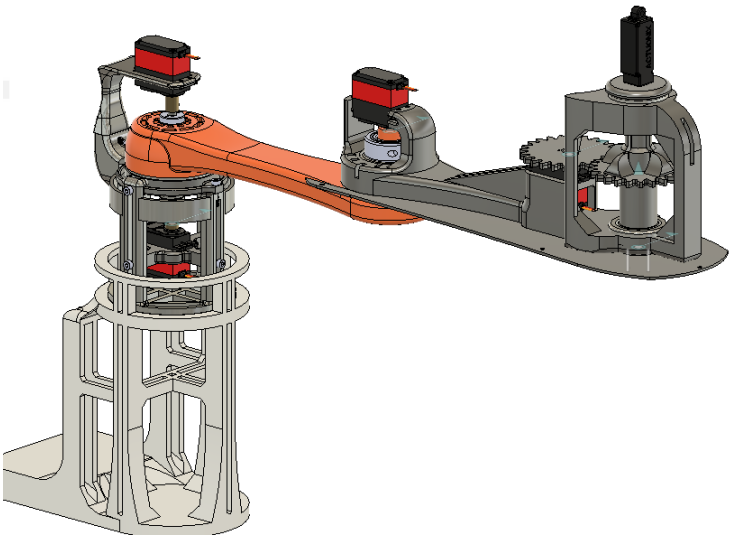


Figure 9 – Design 2 Main Arm without Covers

The second design was based on the first design with some upgrades, like the doubled-up servos for the first joint, however the second design has a different approach to the structure of the robot. Where the first design covers itself, i.e. all the components go in the cavities, the second design is a skeleton structure inside and has separate covers that go on top. This allows a great reduction in

structural weight as the structure now needs to be designed to take the load only regardless of what the outer shell / covers looks like. This minimises the surface area of the robot and therefore minimising the weight, printing cost and printing time of the robot. This also reduces the load on the motors and the torque required by the motors.

This design was a great improvement on the first design; however, it was still not quite there yet. Unfortunately, this time the mistakes in the design were realised after printing it. The links were unnecessarily long and although the robot worked, it did not make much sense to put unnecessary strain on the robot's structure and the motors.

2.1.3 RoboCon Design 3

2.1.3.1 Main Arm

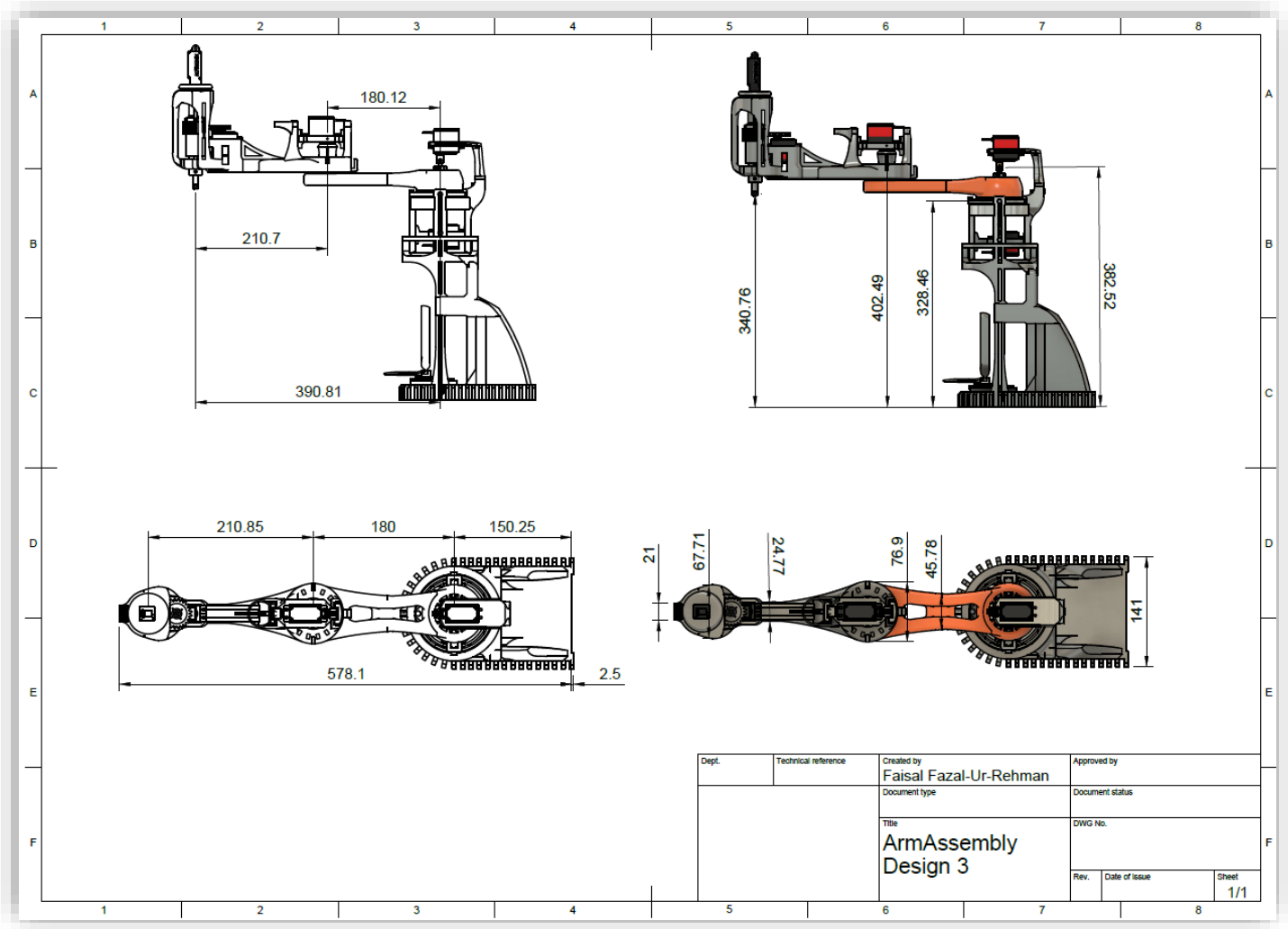


Figure 10 – Design 3 Main Arm Drawing

Figure 11 – Design 3 Main Arm

The third design of the main arm was an amendment to the second design. Some of the amendments were as follows:

- **Base**
  The height of the base was increased to accommodate for the height of the Connect 4 game board. Slot added for the camera fixture (camera fixture designed separately). Slot added for the game fixture.
- **Link 1 (Orange)**
  Link 1 is now shorter in length and was trimmed down by cutting holes for reduction in weight.
- **Link 2 (Black)**
  Link 2 was redesigned from scratch to shorten the length, trim out any excess areas, strengthen the motor holder and for ease of access to areas that require fixtures to be fitted.
- **Cable Holder**
  Added cable holders all around to help keep the wiring clean and safe.

2.1.3.2 End-Effector

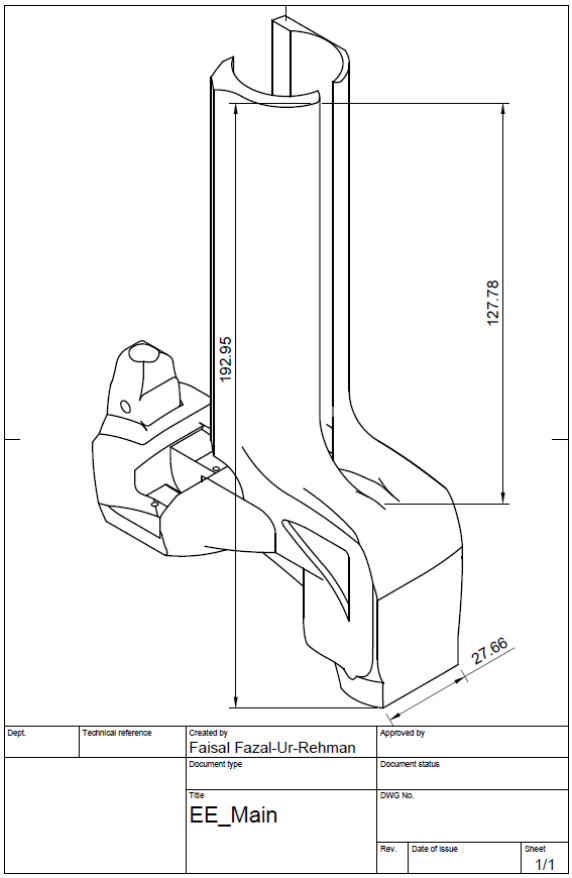


Figure 12 – End-Effector Drawing

Figure 13 - End-Effector

The end-effector is designed to fix directly on to the linear actuator with a single screw. This makes it very easy to design and fix any number of different end-effectors. For this project, the end-effector was designed as a disc dispenser. This is a very simple but effective design and only requires a single 9g servo. The dispenser has a disc holder on top.

The discs stack up on top of one another

Figure 14 - End-Effector Disc Holder

Servo pushes the discs down the slide

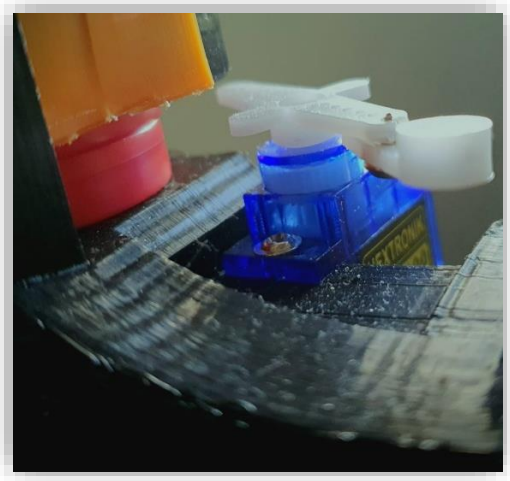


Figure 15 - End-Effector Servo Motor

2.1.3.3 Game Fixture

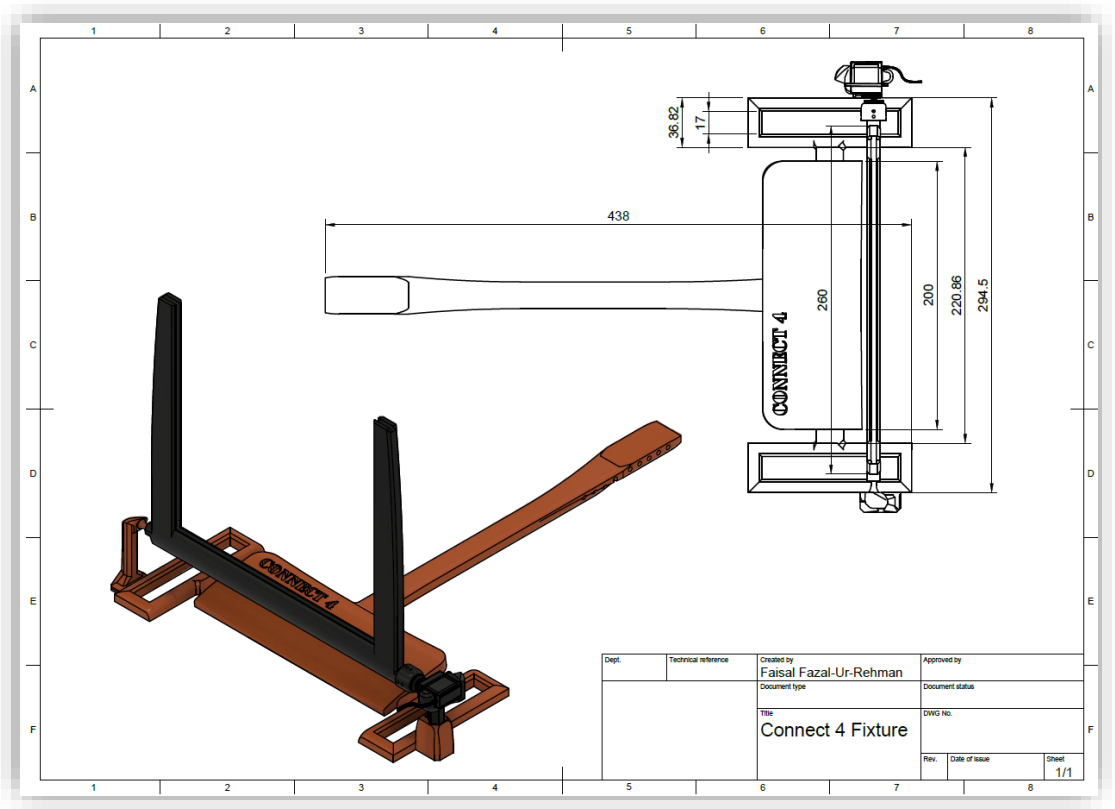


Figure 16 – Connect 4 Game Fixture Drawing

The game fixture slots in at the bottom of the main arm and is secured with a single screw. Like the end-effector, the game fixture is designed separately to allow development of fixtures for different games. For this project, a Connect 4 game fixture was designed. This holds the Connect 4 board in place and provides a screen add-on that secures on the game fixture. The screen add-on has a 9g servo which rotates the screen holder. The screen is used to help with image processing by blocking the view behind the Connect 4 board.

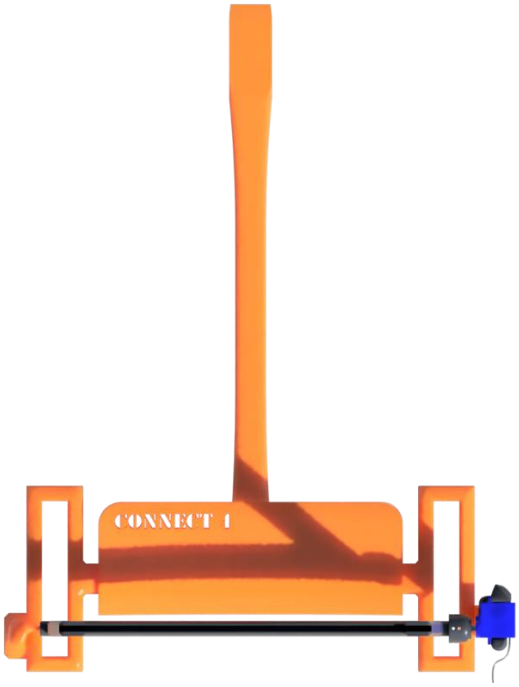


Figure 17 – Connect 4 Game Fixture

2.1.3.4 Full Assembly

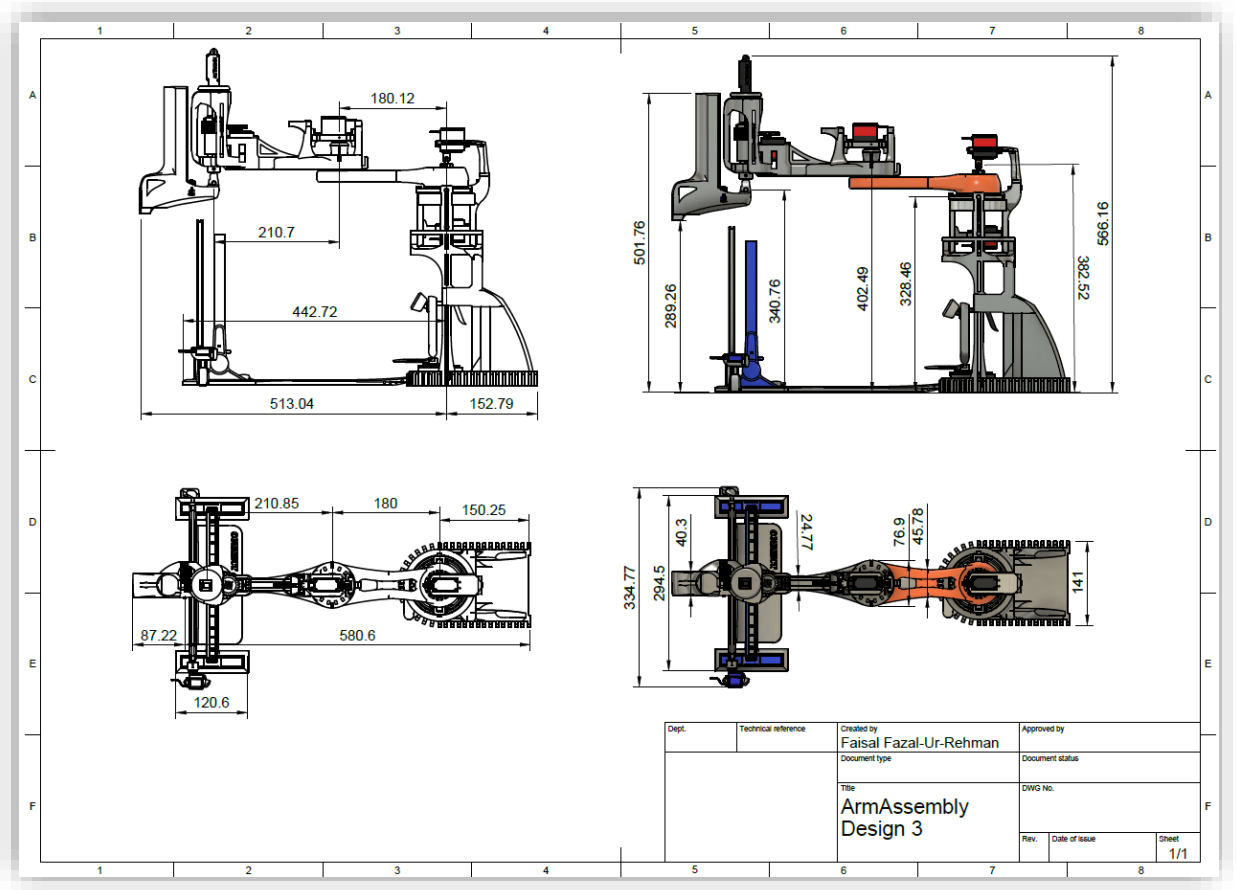


Figure 18 – Design 3 Full Assembly Drawing

Figure 19 – Design 3 Full Assembly

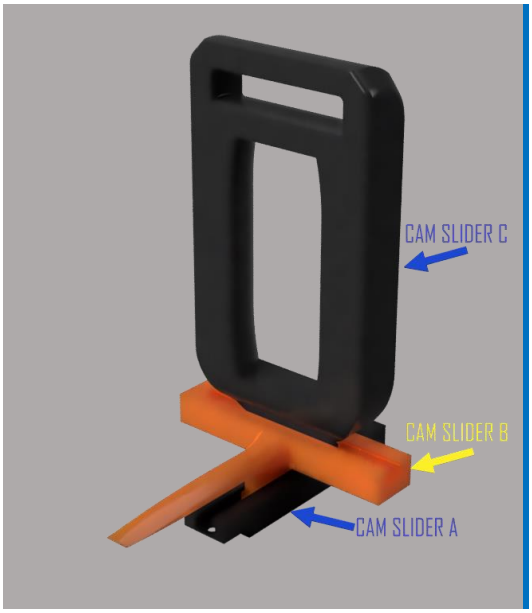
This is the current full hardware design of RoboCon. It works well. It has very little play for a 3D printed model of this size. The average current drawn is 500mA (0.5A) and maximum observed current draw is 700mA (0.7A). For the size of the robot, it would be expected to draw a lot more current than it does. However, this was one of the aims of this design (SCARA), which was to let the structure bare most of its own weight.

Other than the advantages from the SCARA style design, the robot was specifically designed in parts, e.g. the base of the arm is a 2-part design:

Figure 20 - Design 3 Base A

Figure 21 - Design 3 Base B

This saves reprinting base A (Figure 20) as we can just reprint base B (Figure 21) to increase or decrease the height of the robot. Likewise Link 1 (orange) can be adjusted and reprinted to increase or decrease max length of the robot. This can save a lot of time and printing cost and easily allows changing the workspace of the robot if required.



The robot also has a camera fixture which is designed separately and slides right onto the base of the Main Arm.

Similar to the base of the Main Arm the camera fixture is made up of 3 parts, Cam Slider A, B and C. The sliders allow to axis adjustment for the camera. To adjust the camera fixture for another camera only Cam Slider C needs to be adjusted. For this project, Cam Slider C was designed to hold a Logitech c270 webcam.

Key hardware design features:

- Low cost (see section 2.1.3.5)
- Low power
- Off the shelf components makes it replicable
- Easily switch end-effector for different games
- Easily switch game fixture for different games
- Easy switch camera fixture for vision system
- Easy plug-in to add up to eleven more motors (6V) for end-effector and/or game fixture
- Easily increase robot's workspace
- Design makes mathematical models like kinematics and trajectory easier (2D rather than 3D)

2.1.3.5 Bill of Material (BoM)

Description	Type	Quantity	Price / Unit £	Price Total £
3D Printer Filament Spool	PLA	4	18	72
JX PDI-6221MG	Servo Motor	3	11.4	34.2
Actuonix L16 Actuator 140mm	Linear Actuator	1	66.14	66.14
Thrust Bearing 30mm (Inner) x 70mm (Outer)	Bearing	2	10	20
Ball Bearing 30mm (Inner) x 62mm (Outer)	Bearing	1	4	4
Servo Extension cable (Pack of 10pcs)	Cables	2	3.99	7.98
Raspberry Pi 3b+	MPU	1	34	34
ADAFRUIT 16-CHANNEL PWM / SERVO HAT FOR RPI	Motor Driver	1	15	15
Logitech c270	camera	1	30	30
Connect 4	Game	1	12	12
Total Cost				£ 295.3

Table 1 - BoM

2.2 Software

The basis of the structure for the software is for it to be extendable and expandable. Extendable by developing separate classes where possible, as this allows reusing some of the existing code with new game codes. Expandable in size, if the code for a new game is very large and / or would require lots of processing power then the software structure should facilitate this, e.g. MPU1 (like Raspberry Pi) may not be able to handle all the tasks on its own and therefore a solution was required to use a second MPU (MPU2) that can take the load off of MPU1. For this to work a communication system was needed that would allow communication at run-time between the machines. For this project, a raspberry pi and a PC were used to communicate through TCP/IP sockets.

The software is divided into 6 sections, locomotion & motor driver, game environment, game solver, sockets, GUI, and vision system. The locomotion & motor driver and a socket server runs on the RPI and the rest runs on PC.

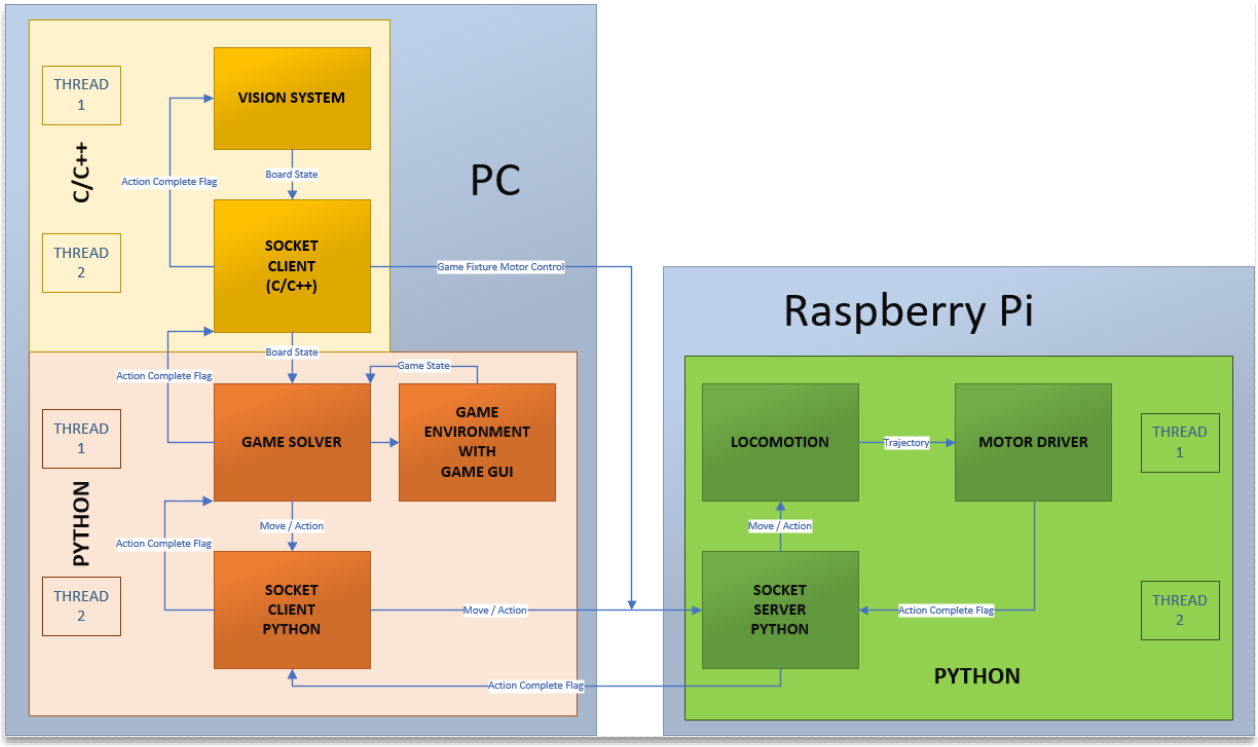


Figure 22 - Software Top Level Flow

2.2.1 Source Code

Please refer to [15] and section 6 for source code.

2.2.2 Locomotion & Motor Driver

Locomotion and motor driver were written in python and run on the Raspberry Pi. A RoboControl class was written for locomotion models and inherits a MotorDriver class that is responsible for setting up and driving the motors through the motor shield (ref 2.1.3.5). When an instance of the RoboControl class is declared it initiates the motors through the MotorDriver class and sets the pose to home position. The RoboControl.setPose(x,y) method takes x and y coordinates as arguments and moves the end-effector to the given coordinates, drops the Connect 4 disc and returns back to the home position. Currently this is the only method required to control the robot as it performs all the necessary tasks required in the following manner:

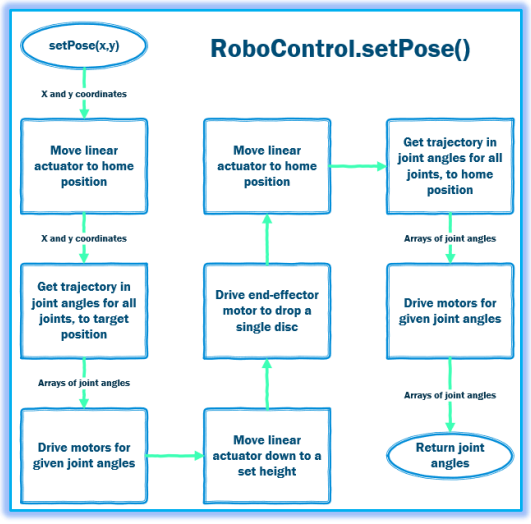


Figure 23 - Flowchart for setPose method of RoboControl Class

As the setPose method performs all the necessary tasks, there is a lot going on within this function, especially in the “get trajectory” part. The get trajectory part calls the sLineTraj method of the RoboControl class, which takes the target x and y coordinates as arguments. This method has three main parts as follows:

1. **Check coordinates fall within the workspace of the robot**
   This protects the system from crashing. To do this an algorithm was created that takes the Pythagoras of the x and y target coordinates, to find the hypotenuse distance, and limit them to the total distance of link 1 + link 2:

```
self.L1 = 0.17996
self.L2 = 0.21085
self.WORKSPACE = self.L1 + self.L2
#----------------------------------------
#Sets limit on cartesian coordinates
def setCoordinateLimits(self,x,y):
    xy = math.sqrt(x**2 + y**2)     #calculate hypotenuse
    OFFSET = 0.001                  #adjustment step size

    while(xy >= self.WORKSPACE):    #keep iterating while hypotenuse is greater than the max length of the arm
        if (abs(x) > abs(y)):       #adjust the x coordinate if greater than y
            if (x > 0):
                x -= OFFSET
            else:
                x += OFFSET
        else:                       #adjust the y coordinate if greater than x
            if (y > 0):
                y -= OFFSET
            else:
                y += OFFSET

        xy = math.sqrt(x**2 + y**2) # re-calculate hypotenuse

    return x,y
```

2. **Calculate a straight line**
   In this part, the algorithm takes the starting position and the target position and calculates given number of equal steps in a straight line between the two positions. The steps to do this are as follows:
   a Find gradient of the line between two positions:

$$m = \frac{y_{target} - y_{start}}{x_{target} - x_{start}}$$

   b Calculate step size:

$$step\ size\ y = \frac{|y_{target} - y_{start}|}{steps\ required} \quad \text{or} \quad step\ size\ x = \frac{|x_{target} - x_{start}|}{steps\ required}$$

   c Store steps in arrays:
   If step size y was used:
   trajectory pose Y (tpY) = all the steps between Y_{target} and Y_{start} of size “step size y”
   trajectory pose X (tpX) = $\frac{tpY - y_{start}}{m} + x_{start}$ (for all points in tpY)
   If step size x was used:
   trajectory pose X (tpX) = all the steps between X_{target} and X_{start} of size “step size X”
   trajectory pose Y (tpY) = Y_{start} + m*(tpX - X_{start}) (for all points in tpX)

3. **Calculate Inverse Kinematics**
   In this part the inverse kinematics is calculated for each point in tpX and tpY (from section c). The joint angles returned by the inverse kinematics method, for each of these points, are then stored in an array. The inverse kinematics for a two link (RR) robot is a known solution and is given by the following equations:

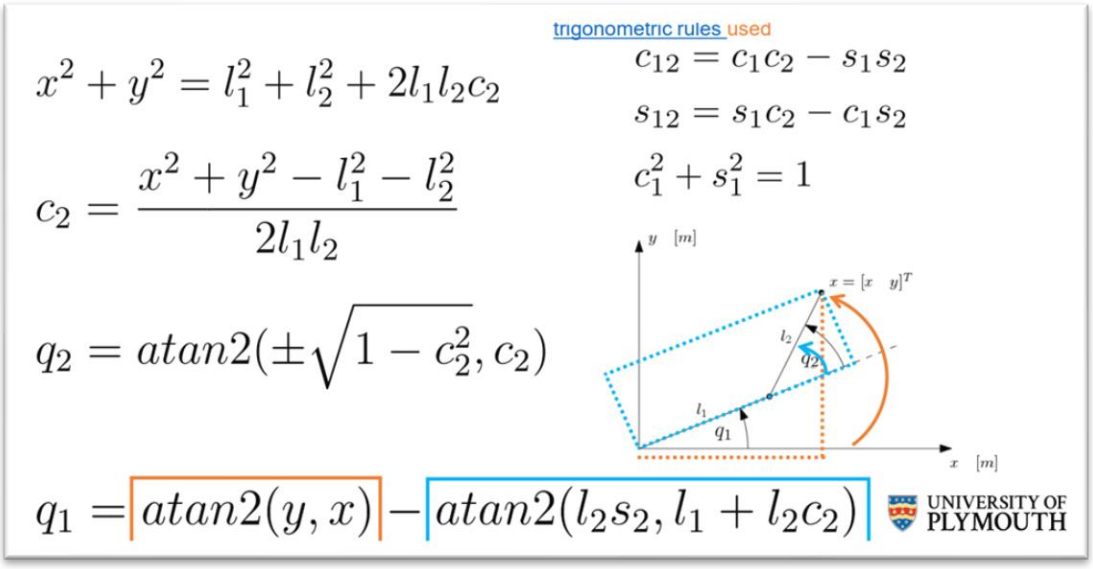


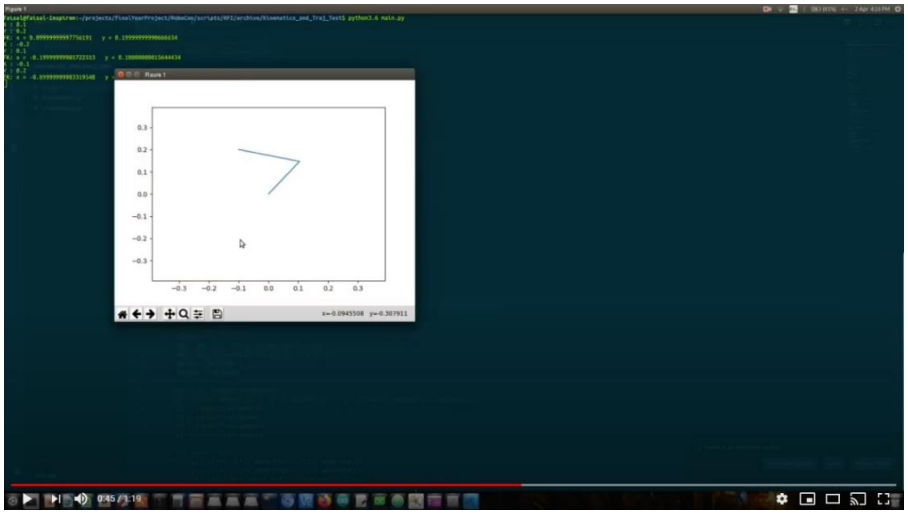
Figure 24 – Inverse Kinematics Solution [8]

The trajectory arrays of all relevant joints are then returned by the sLineTraj method.

2.2.2.1 Testing

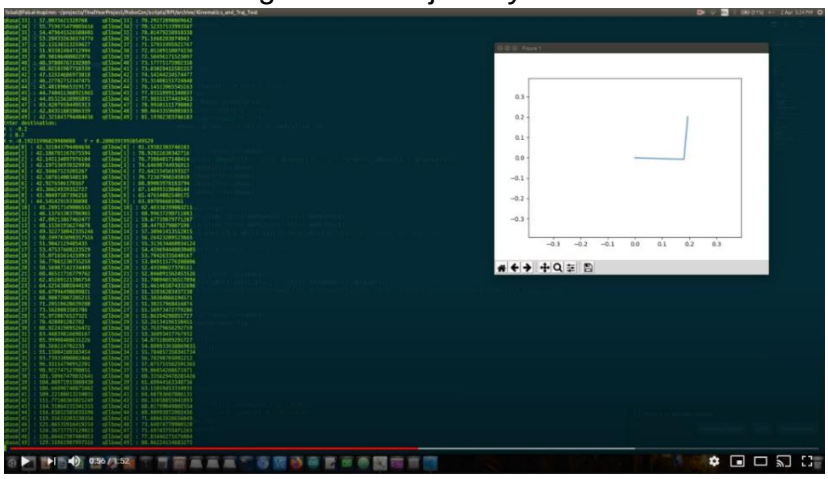
Every function that was created was individually tested and then the class was tested as a whole for its purpose at the end.

- RoboControl Inverse Kinematics Test



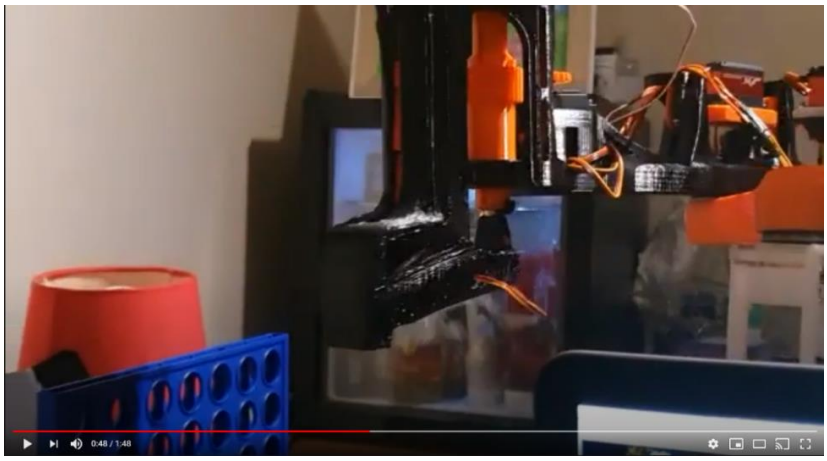
Video 1 - RoboControl Inverse Kinematics Test

- RoboControl Straight Line Trajectory Test



Video 2 - RoboControl Straight Line Trajectory Test

- RoboControl and MotorDriver Test



Video 3 - RoboControl and MotorDriver Trajectory Test

2.2.3 Game Environment

The game environment is written in python and runs on the PC. The game environment is split into two classes, Connect4 class and Connect4Env class. The Connect4 class describes the game rules and provides the GUI of the game board with state updates. It also checks if the game has finished and whether red or yellow has won the game or if it is a draw. The GUI for the board is written with the pygame library and is also part of the Connect4 class. The Connect4Env class inherits the Connect4 class and works like a bridge between the Connect4 class and the DeepQNetwork class (see section 2.2.4). It is concerned with providing information to the DeepQNetwork class rather than describing the game itself. It is written to provide the same structure as the OpenAI Gym library environments to facilitate the use of these with ease. It provides a step method for RoboCon and a separate step method for the opponent. The step methods take action as argument and return four variables, new_state, reward, done and valid.

The new_state represents the state of the board after the last action taken and is a numpy array of 42 integer values, where each integer value can be 0, 1 or 2. The value 0 represents an empty slot on the board, 1 represents a yellow disc and 2 represents a red disc on the board.

Reward is an integer variable which is used for training DQN and the values represent the reward given for the action taken. It is related to the Q-learning section of the DQN. The values can be set depending on the game being learnt and when the reward is given may differ, e.g. the reward

values that were used for connect 4 training was 100 reward for winning the game, -200 for losing the game, 50 for a draw and 0 during the game. This means that no reward is given until the end of a game, but this can be changed.

The done variable is a Boolean variable. It is “False” (value = 0) if the game has not finished or “True” (value = 1) when the game has finished.

The valid variable is an extra variable returned by the function, that has been added compared to the normal Gym environment structure. This is used to safeguard against an invalid action taken. If the step method receives an invalid action it will return the valid variable as “False”.

Some extra features were also added to the Connect4Env class. The added features help with setting up the Deep Q Network e.g. it provides the number of required outputs for the Neural Network in the Deep Q Network, it also provides the required number of inputs needed for the Neural Network.

A counter move method was also created which can be used to force a player to always counter a winning move when possible. When used by the opponent, this greatly improved DQN training performance.

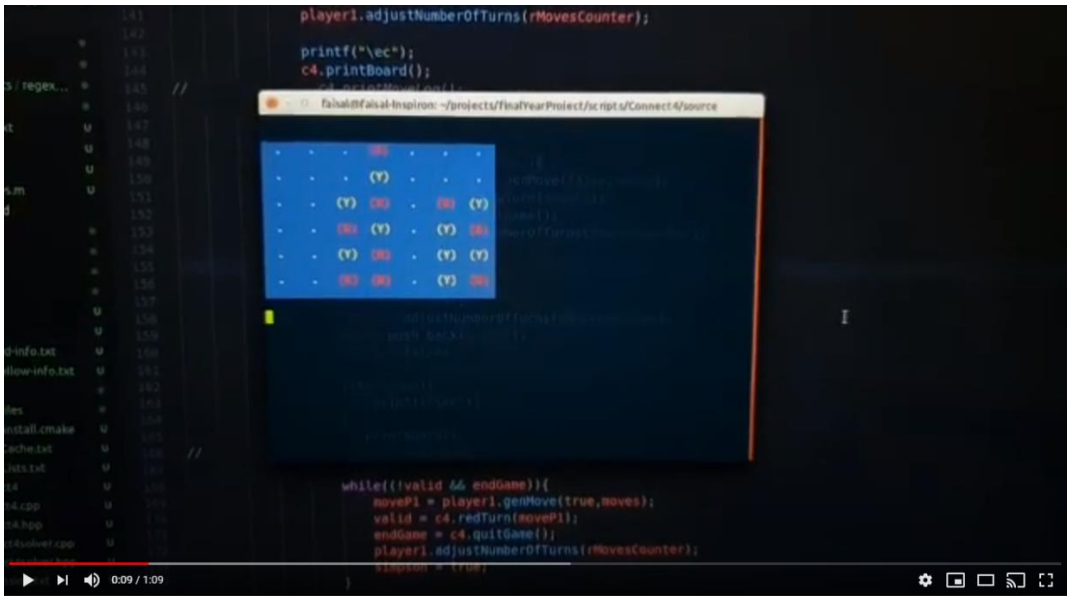
2.2.4 Game Solver

The game solver is the heart of the software for this project. This is what makes the robot worthy of being a player against humans. As if the robot cannot compete against the humans then it loses the purpose of being a gaming robot.

2.2.4.1 Lookup Table Solution

The first approach taken was a naïve one, as it developed a game environment and solver in C++ that would populate a lookup table with sequences of games that it has won. This looks like a simpler and effective solution at first. However, this is neither a simple nor an effective solution. The game sequences were saved in files. Each game sequence was an array of moves played in that game. Two separate batches of files were created, one where the red player played the first move and second batch with sequences where yellow player played the first move. The game sequence being used had to be checked after every move, as the opponent cannot be restricted to play the same game sequence. This meant not only checking the sequence after every move but if a new sequence was required, it needed to check if another sequence even existed, that matches the current state of the game and if not, start playing randomly and save this new sequence if this game was won. This would be sufficient, however a connect 4 game has 4,531,985,219,092 total number of possible situations [7], which is an astronomical amount of sequences, especially to iterate through every time the current sequence becomes invalid. Also, one sequence per line was saved and to save all the possible situations it would have to be saved across hundreds if not thousands of files, which would then need to be iterated through. Not a smart solution.

A lookup-based game solver was created, however Dr Mario Gianni (project supervisor) advised to find a smarter solution and suggested to investigate Deep Q Networks.



Video 4 - Lookup Table

2.2.4.2 Deep Q Network

Deep Q Network was something new for me. I had some knowledge of Neural Networks and Q-learning, but it took a lot of research, reading and learning to understand Deep Q Networks intrinsically. This is because I learnt QL with known and deterministic states but connect 4 has too many states for this. For connect 4 the states can be considered as continuous states and due to this the DQN is a good option.

2.2.4.2.1 Theory

A Deep Q Network is a NN with QL algorithm connected at its output. It uses the output from the NN as the Q-values of the Q-table. Before looking at the DQN it-self it's easier to understand it by first looking at NN and QL with discrete states (see reference [13]). QL in the DQN helps with the future values where as a NN is concerned with the present only. A NN can take actions that it has never seen before whereas if an unknown state is presented to a QL algorithm it takes a random action.

NNs are difficult to optimise and for non-critical applications they are generally considered acceptable at 90% and above accuracy. Some of the features we can adjust that effect the performance of a NN, and therefore a DQN, are as follows:

- **Training Data**
  Is the data used for training the DQN. It plays a big part in the performance of the NN. In general, the more generalised this data is the better the training results tend to be. For this reason, a memory replay technique is used. The DQN for this project generates training data as follows:
  - For every move, the action taken, state of the game before the action was taken, state of the game after the action was taken, reward after the action was taken and whether the game is finished, is saved as a python list in a memory pool (let's call this a single "data point"). This is true for all the actions taken during a training session.
  - First, it plays a given number of actions without training. Let us say first 1000 actions taken will be played without training the DQN. This will give us 1000 data points to train from.

- For every action taken, after the first 1000 actions, we train our DQN for a given number of episodes (let us call this a mini-training session). However, the training data used is not the whole memory pool but a minibatch of randomly selected data points from the memory pool. A new minibatch is used for every mini-training session.

This helps randomise and therefore generalise the training data. It also generates training data from scratch.

- **NN Activation Function**
  The activation function sits at the end of each layer and defines the output of its layer. There are many types of activation functions, like sigmoid, Rectified linear unit (ReLU), TanH, SoftPlus etc. The sigmoid activation function compresses the output of the layer between 0 and 1 and forms an s-shaped curve. The TanH activation function compresses the output of the layer between -1 and 1 and like sigmoid it forms a s-shaped curve. The ReLU activation function adjusts the output between 0 and ∞ and forms a ramp shape. The ReLU activation function was used for the DQN:

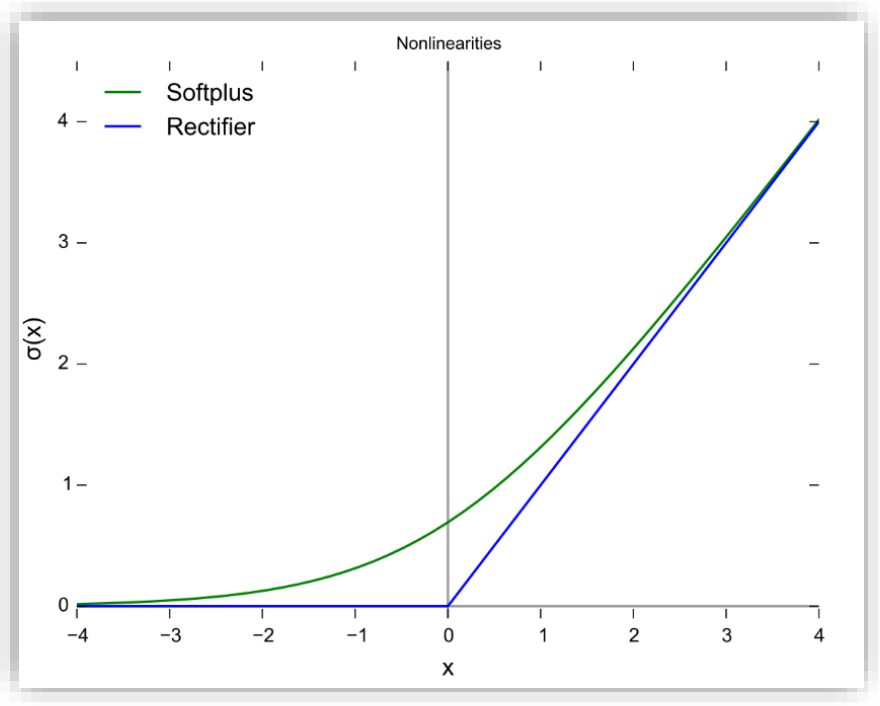


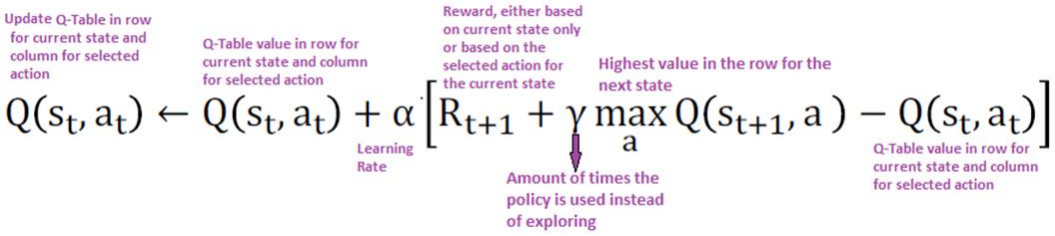
Figure 25 – Rectified Linear Unit (ReLU) [10]

- **NN Weight Initialisation**
  The aim of this is to prevent layer outputs from exploding or vanishing. If either of these occur, loss gradients become too large or too small to flow backwards beneficially, and the network takes longer to converge, if it is even able to do so at all. There are many weight initialisation methods used, like Xavier, He, Glorot etc, they are however very easy to use with Keras [11]. As the Relu activation was used in the layers, the He Initialisation was utilised. With the He initialisation method, the weights are initialized keeping in mind the size of the previous layer which helps in attaining a global minimum of the cost function faster and more efficiently. The weights are still random but differ in range depending on the

size of the previous layer of neurons. This provides a controlled initialization, hence the faster and more efficient gradient descent [12].

Apart from the ones mentioned above, there are many features of a NN that we can play around with to tune the NN, like number of hidden layers, size of the hidden layers (number of neurons in a layer), types of layers (dense (fully connected), convolutional, pooling etc), learning rate, number of training episodes, size of training etc.

The Q-learning part in a DQN plays a big part but only makes use of the update equation from the Q-Learning algorithm. The Q-table update equation in Q-learning is as follows:



Whereas the Q-table update equation used in a DQN only makes use of half the equation used for Q-learning:

$$Q^{new}(s_t, a_t) = R_{t+1} + \gamma * \max Q(s_{t+1}, a_t)$$

This is a small but very power equation. From right to left of the equation:

- $\max Q(s_{t+1}, a_t)$ - Grabs the maximum number from the NN outputs
- γ - Makes the number smaller (γ is < 1)
- R_{t+1} - Adds the reward value. For this project the reward value is 0 during the game, -200 for a lost game, +50 for a draw and +100 for a win.

$Q^{new}(s_t, a_t)$ is propagated back through the NN and adjusts the weights of the NN. It rewards winning game sequences or game sequences that resulted in a draw but punishes lost game sequences. This means that it nudges the weights to provide outputs with higher values for actions that would result in winning the game. The Q-table update equation steers the NN for results guided by rewards.

A DQN usually also makes use of 2 separate NNs. One which is trained for every action taken and the second NN which copies the weights of the first one after a given number of episodes. This helps reduce the noise in the network. The model that is actually used is with the second NN.

2.2.4.2.2 Software

The DQN software was written in python using Keras with Tensorflow, Tensorboard and OpenCV. There are two parts for the DQN software, one set of scripts are for training and testing the DQN and the other one is for using it with the rest of the software.

After trying various structures for the DQN, (different number of layers, layer sizes etc) the best results were with the following structure:

- 4 fully connected layers.
- 256 neurons per layer (layer width)
- ReLU activation for first 3 layers with linear activation in the final layer.
- He weights initialisation with normal distribution.
- Learning rate of 0.00001

- Gamma / Discount Factor of 0.99
- Max Memory Pool of 100,000
- 2000 number of actions to take before training starts
- Minibatch size of 200
- Decaying Epsilon for greedy function

The DQN was trained in multiple sessions rather than in one long session, this allowed variation to the training sessions for better results. The changes made between sessions were, number of iterations, use of greedy function (whether to always use Q-Table for decision or not), difficulty of the opponent (random, use DQNs Q-table or use Q-table and use counter moves).

The two set of scripts for DQN are the same except for the training scripts have extra functionality for analysing training sessions. The training scripts provide two ways to analyse the training performance of the DQN, by use of live graphs with Tensorboard and by creating a video of the training session in the background using OpenCV.

Tensorboard is a part of the Tensorflow library and is a fantastic analysis tool for DQN training. It pushes the logs created onto a server and produces graphs during training. The logs are saved to the given path. The logs that were created contained the following:

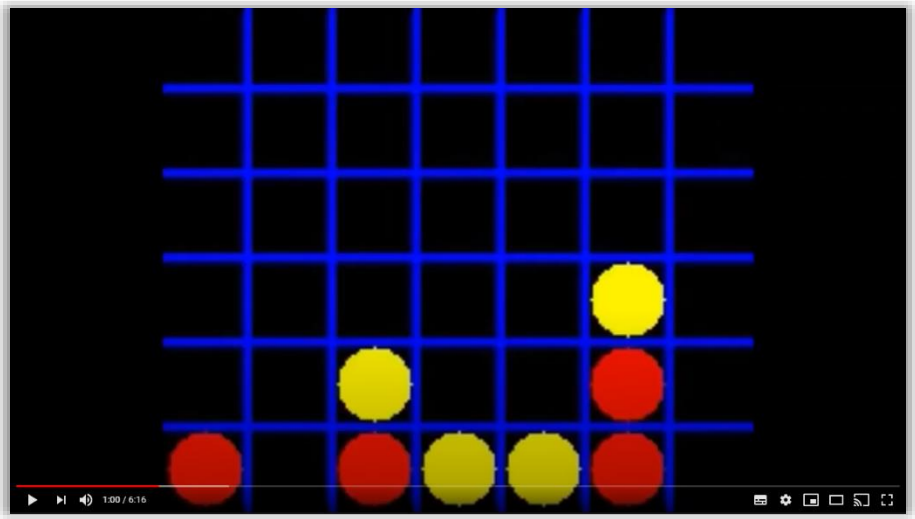
- Loss
- Minimum Reward
- Maximum Reward
- Average Reward
- Epsilon, to check the decay time

These graphs played an integral part in improving the performance of the DQN.

While training the DQN there were some issues. The average reward seemed to shoot up very quickly to the max value and remained stable up there. The DQN was learning well, but there was not much difference in the way it was playing compared to when it was set to play with random moves. During my Project Progress Demo with Dr Mario Gianni (Project Supervisor). He suggested that I observe the training sessions, as at the time I was only checking the graphs and testing the model at the end. However, observation during training was a tricky thing to get around. The following options were looked at:

Option	**Issue**
Turn GUI on during training.	Average training speed was 3 to 4 full games per second, so direct observation of the GUI was a no go. Due to the speed recording the GUI on screen would not work either.
Turn on the GUI during training and slow the training down.	This allowed direct observation of the training with the GUI on or turn on screen recording while the GUI was turned on as well and leave it to train and check the video recordings as or when it was needed. However, this slowed the training right down as the minimum delay that could be added with reasonable time to look at each move was 0.5 second per move! Opposed to 3 to 4 full games per second.
Find a way to record a video in the background	As the game's GUI is developed with pygame library, at first, it was considered that the pygame might provide an option to record the

without slowing down the training.	GUI in the background, but there did not seem to be any options available to do this. As there did not seem to be an existing solution to do this, it was decided to create a new algorithm. To do this the algorithm was created with OpenCV library, that creates its own representation of the game and given the current state of the game it creates an image (which is just an OpenCV Mat variable) with the current state of the game. These images are stored in an array during the training session at the end the stored images are stitched together to create a video of the training session. This works very elegantly, as by observation it has not slowed down the training speed and allows the observation of the video at any time. This also allows a record to be kept of the training sessions in case a previous training session needs to be looked at.



Video 5 - Deep Q Network Training

By observing the training videos it was realised that the games being played by the DQN were the same for every training session, e.g. training session 1 game 1 was the same as training session 2 game 1 and training session 1 game 2 was the same as training session 2 game 2 and so on. Due to this and a suggestion from Dr Gianni, the part of the code that was generating the training data was investigated and it was realised that randomising of samples, while fetching the minibatches, were being seeded. This randomises the samples but in the same sequence every time. After fixing this issue the results were much better when playing against the DQN. Now the DQN was starting to play some counter moves and although, most of the time, it seemed to like playing vertically it was at least trying to win.

Let us look at the flow charts for the DQN. Figure 26 shows the flow diagram of the train function and Figure 27 shows the DQN thread routine flow diagram.

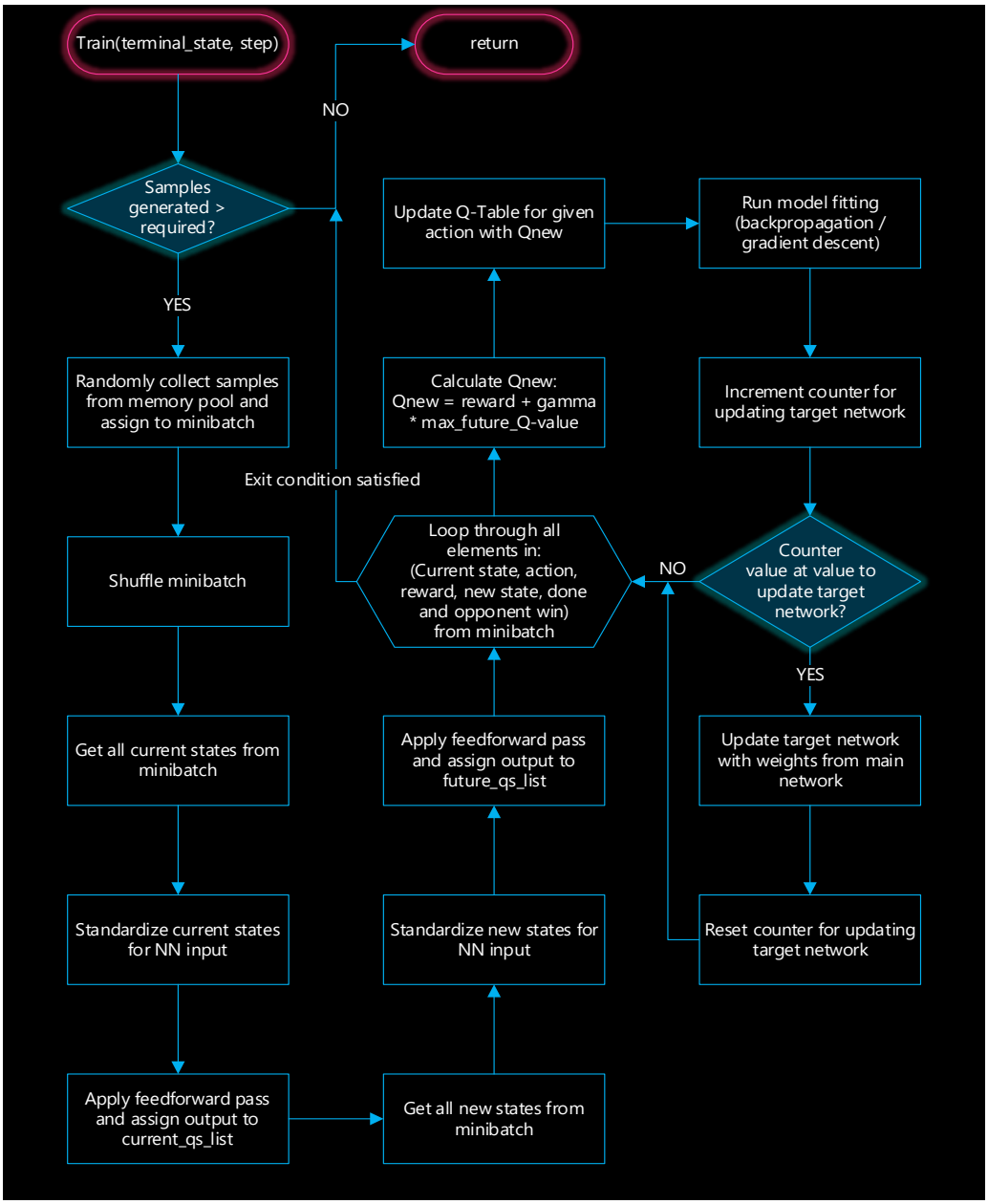


Figure 26 - Deep Q Network Train Method Flowchart

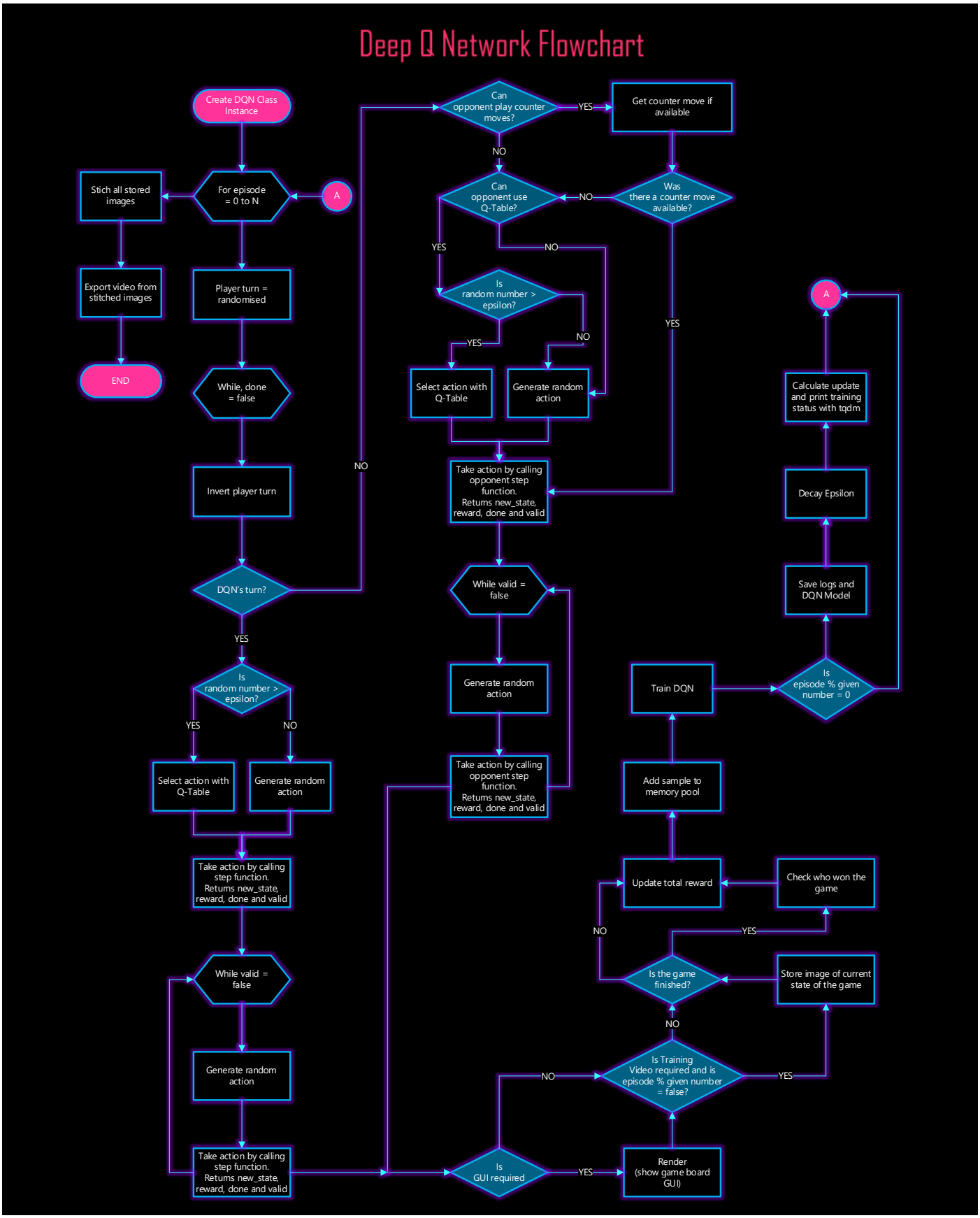


Figure 27 - Deep Q Network Main Flowchart

2.2.5 Sockets

Sockets were completely new for me. Although I have used ROS publishers and subscribers before I had no idea that these were based on sockets. It involved lots of reading and learning through various forums and articles online for me to understand enough about them to be able to use them in my project.

Sockets were required for two reasons, to communicate between C/C++ and python scripts and to communicate between PC and Raspberry Pi. There were some other options to communicate between C/C++ and python code, one option was to use Cython, another one was to communicate through files. However, learning a new language (Cython) would have taken much longer and some python features would have been lost. Initially a file communication system was developed between C/C++ and python but during my Project Progress Demonstration Dr Mario Gianni advised me against it. Communicating through files use hard drive and is not the most elegant solution. Communication between PC and Raspberry Pi had to be considered as well and communicating through files would not be a feasible solution for this. A serial communication system between PC and RPI would have been one option but I wanted to keep the PC side and the RPI side as physically independent as possible, also if I had one solution for both problems then that is a preferred option. These were the reasons why sockets were the best solution.

Sockets work in a Server, Client setup and for communicating between PC and RPI, it had to be decided which one would be the server side and which one would be the client. For this I did some research and made the decision based on the following

- The server sets up a port and waits for client(s) to join. A server can connect with multiple clients.
- If the server shuts down the client would shut down too but not the other way around.
- Client can connect to server based on hostname.

It was decided that RPI would be more suitable to be the server side due to the following, and based on the points above:

- If RPI was the client, connection errors would require restarting the scripts on the RPI.
- If RPI is the server, no amendments to hostnames will be required if using a different pc.

There are three server, client combinations in this project:

A. To communicate between C/C++ and python, PC to PC.
B. To communicate between C/C++ and Python, PC to RPI and back.
C. To communicate between Python and Python, PC to RPI and back.

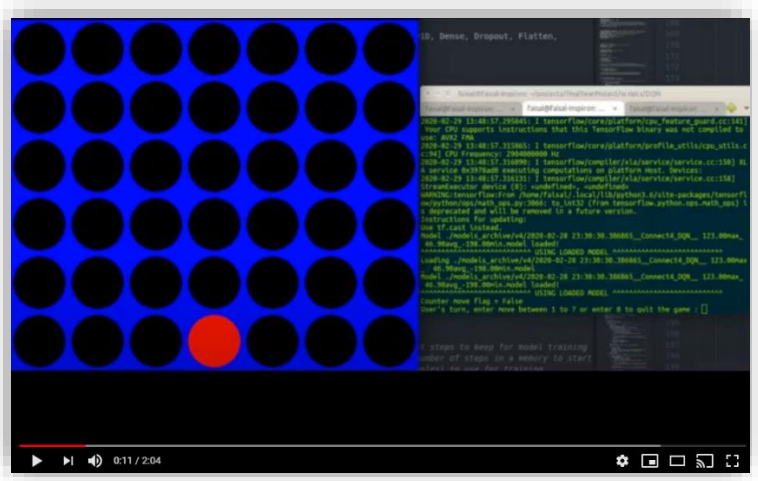
This covers all bases of communication required for the current setup of this project. Currently A is used for communicating between vision system (C/C++) and the rest of the Python code. B is used to communicate between vision system (C/C++) and the motor on the game fixture for the back screen. C is used to pass the action generated by the DQN to the code on the RPI to move the robot and play the action received.

The sockets written in C/C++ use the Boost library and for python, the standard socket library was used. Although this took some time to get working it works like a charm. To practice and test the socket scripts very simple server and client scripts we written in C/C++ and Python and they were tested in all combinations (A, B and C) that were needed.

2.2.6 Graphical User Interface (GUI)

GUI used for this project has two parts. One is the controls part, written in python using PyQt library but was created as a separate project [14]. The other one is part of the game environment

class Connect4. Written in python using pygame library this provides a live game board which is updated during the game with updates from the DQN for red disc and updates from the vision system for yellow discs. To test this, the internal game updates were used for both red and yellow discs from the DQN before using it with the vision system (external) updates (ref Video 6).

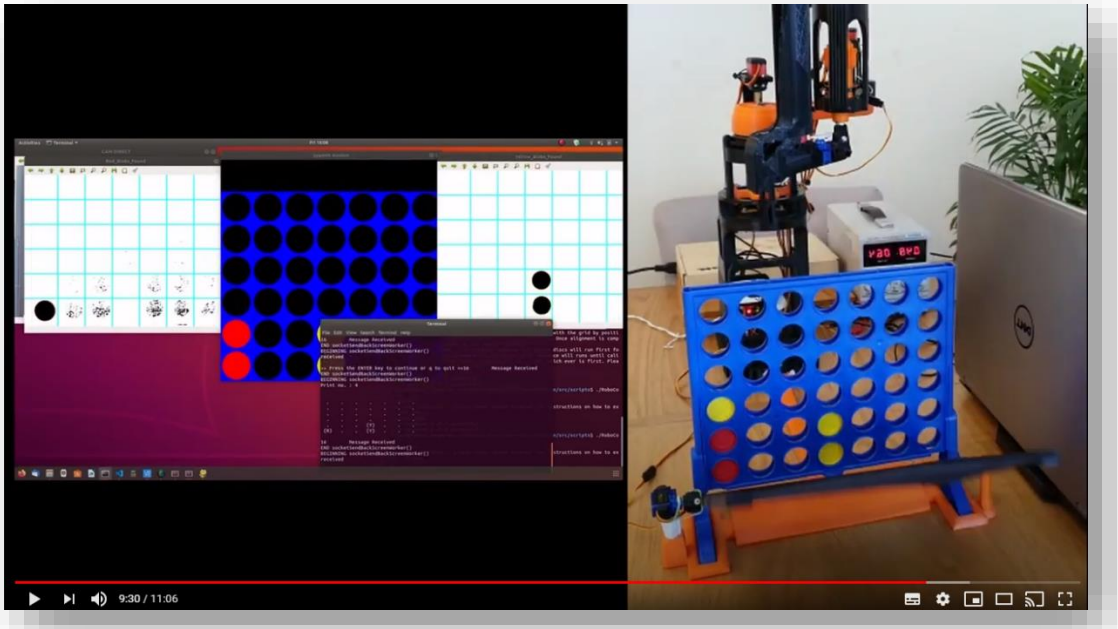


Video 6 - Testing Connect 4 Game Board GUI

2.2.7 Vision System

The vision system used in this project was developed as a separate project [14]. However, for this project the vision system class had to be adjusted to use in a multi-threaded environment and with the sockets class RoboSocket, which was developed as part of this project.

3 Video Demonstration



Video 7 - Final Demonstration

4 Further Improvements

Some improvements that would make the system more robust and user friendly are as follows:

- Although the vision system used in this project was written as a separate project, it is currently the weakest link of this project. This project can benefit from stereo vision. This would allow the use of vergence and / or disparity mapping algorithms to extract depth perception from the images. This would have particularly helped with the connect 4 game, as the board is used in an up right formation and has holes in it, and due to this the camera pics up the colours behind it. With the help of depth perception, it would be possible to ignore any colours detected farther than the board. This may also be possible with a single camera and the current setup but by adding a 9g servo on the camera fixture as this would allow us to move the camera to take a second picture at a known horizontal distance which would then allow us to either use vergence or disparity mapping.

  The vision system could also be used to scan the workspace while the robot is moving and stop the robot if an unexpected object appears within the workspace to make the robot safer to use.

- The motors used are servo motors with a potentiometer inside. This potentiometer is easy to hack and can be used for reading the voltage across it. This would allow us to close the loop and use the feedback with a PID for consistency.

- Using an op-amp circuit, we can detect the current drawn by the motors and therefore it would be possible to limit the torque applied by the motors. This would be very useful to avoid crushing objects with the robot unintentionally. Although I have not fully investigated this, but I did spend some time thinking this over and came up with the following circuit that seems to work on a simple simulation:

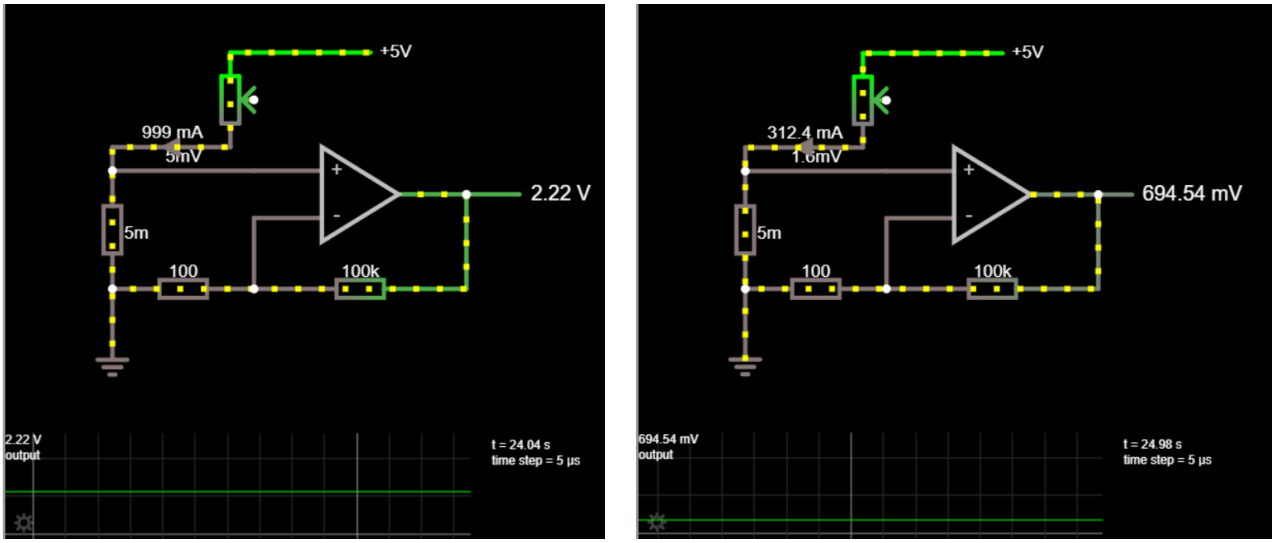


Figure 28 - Op-Amp Circuit to Detect Current Drawn by a Motor

The output needs to be in voltage so that it can be directly plugged into the ADC of an MCU. The 5mΩ resistor needs to be small to reduce the overall voltage drop. I bought the resistor in Figure 29 from Farnell, however I have not built or tested the circuit yet.

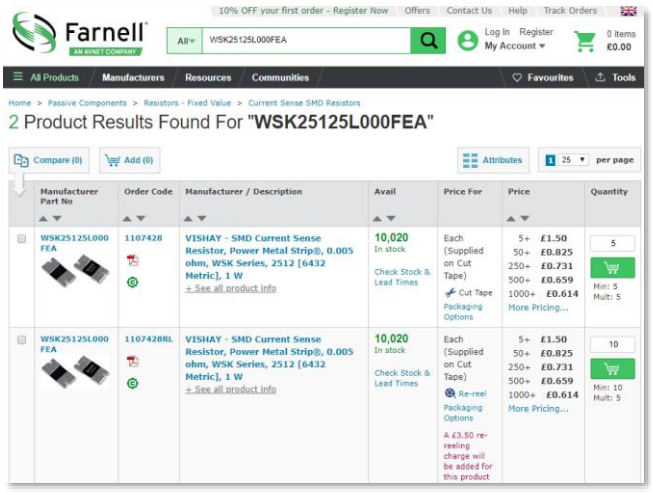


Figure 29 - 5mΩ Resistors (Farnell)

- The Deep Q Network needs to be optimised. Due to time and lack of resources I was not able to improve the performance of the DQN to a level that I was hoping for and as much as this project is not solely based on machine learning it is something that would make this project shine.

5 Conclusion

The software and hardware design of the robot provides a model that is low-cost and was built with 3D printed parts and off the shelf components which makes it replicable. The model developed can play Connect 4 games and provides easily switchable end-effector design, which can be switched by replacing a single screw and a game fixture slot to add new games. It also provides an extendable and expandable software design to help facilitate adding new games to the system. The current system works well however a few improvements mentioned in section 4 would make this project more robust and user friendly.

6 Software Operating Instructions (SOI)

6.1 Dependencies

The software was developed on Linux Ubuntu 18.04 LTS (PC) and Raspbian (RPI) using the following:

Application / Library	Version
PC	
Cmake	3.5
Python	3.6.9
Tensorflow	1.15.0
Keras	2.3.1
Pygame	1.9.6
Tqdm	4.45.0
NumPy	1.18.2
OpenCV (python)	4.2.0.34
Matplotlib	3.2.1
Boost (C++)	1.65.1
Raspberry Pi	
Python	3.7.3
adafruit-circuitpython-servokit*	1.1.1

Table 2 - Software Dependencies

* May require additional libraries, please follow the link for details:
https://learn.adafruit.com/adafruit-16-channel-pwm-servo-hat-for-raspberry-pi/

Please check reference [14] for vision system dependencies.

6.2 Setup

The following instructions assume that all dependencies listed in section 6.1 have been satisfied.

6.2.1 Get Repository

- First get a copy of the source code from Github:
  - ssh:

```
$ git clone git@github.com:Faisal-f-rehman/RoboCon.git
```

  - https:

```
$ git clone https://github.com/Faisal-f-rehman/RoboCon.git
```

6.2.2 Raspberry Pi

- In the RoboCon folder there are two sub-folders, PC and RPI. First Copy the RPI folder onto the Raspberry Pi. Then open terminal (ctrl + alt + t) and change directory into RPI folder (this may vary depending on where you have placed the RPI folder on your Raspberry Pi):

```
$ cd RPI
```

- Execute python scripts:

```
$ python3 main.py
```

  - Or (may require sudo):

```
$ sudo python3 main.py
```

6.2.3 PC

6.2.3.1 Setup with bash script

- Open terminal (ctrl + alt + t) and change directory to PC in the RoboCon folder:

```
$ cd RoboCon/PC
```

- Make bash (shell) script executable:

```
$ chmod +x RoboCon.sh
```

- Execute bash (shell) script*:

```
$ ./RoboCon.sh
```

- Enter y on the terminal and press enter (to skip generating build files, enter any other key):

```
$ y
```

* This executes a bash script that generates build files using cmake, builds the files using make, makes another bash script executable located in the scripts (RoboCon/PC/src/scripts/roboCon.sh) folder and executes it. This saves the hassle of changing directories between build and src folders for any subsequent builds and runs.

6.2.3.2 Setup without bash script

For manual setup instead of using the bash script provided, use the following instructions:

- Open terminal (ctrl + alt + t) and change directory to PC in the RoboCon folder:
```
$ cd RoboCon/PC
```
- Create build directory:
```
$ mkdir build
```
- Generate build files:
```
$ cmake ../src/scripts
```
- Build generated files:
```
$ make
```
- Change directory to scripts directory:
```
$ cd ../src/scripts
```
- Make roboCon.sh bash script executable:
```
$ chmod +x roboCon.sh
```
- Execute program:
```
$ ./roboCon.sh
```

6.3 Source Code Explained

6.3.1 PC

6.3.1.1 PC/src/scripts

There are total 13 files in this directory, out of which 7 of them are C/C++ files and 6 of them are python files.

6.3.1.1.1 C/C++

1. **main.cpp**
   Holds the main (entry point) function for the c++ code. It starts all the threads and waits until the joined threads have finished.

2. **vision.cpp and vision.hpp**
   Only thread and concurrency related code was developed in this project, rest of the code was developed in a separate project [14]. Defines the vision system class and holds the vision system's thread routine function in vision.cpp.

3. **guicomms.cpp and guicomms.hpp**
   Developed in a separate project [14], it defines the class that is responsible for communication between the vision system and the GUI for controls.

4. **roboSocket.cpp and roboSocket.hpp**
   Defines a socket class for C++ side of the code. It was written with Boost library and holds the thread routines for two clients:
   - PC to PC (C++ to Python)
   - PC to RPI (C++ to Python)

6.3.1.1.2 Python

1. **main.py**
   Defines all the threads, starts them, and then joins some of these threads and waits for them to finish.

2. **connect4dqn.py**
   Defines the Deep Q Network class and the thread routine for using it. It uses an instance of the game environment class for the game.

3. **connect4env.py**
   Defines the connect4 class which describes the connect 4 game rules and a GUI of the game board. It also defines the connect4env class which inherits the connect4 class and connects the DQN and the connect4 class.

4. **rpiClient.py**
   Defines a socket client for communication between PC and RPI (python to python).

5. **client.py**
   Defines a PC to PC (C++ to / from python) socket client.

6. **RoboConGui.py**
   Developed in a separate project [14], it provides the GUI based controls for the vision system and the game.

6.3.1.2 PC\src\DQN_Training

There are only 3 files all written in python and are used for training the Deep Q Network only.

1. **connect4dqn.py**
   Similar to section 6.3.1.1.2 part 2, with some added features for training like using the gameRecord class (see part 3 of this section)

2. **connect4env.py**
   Same as section 6.3.1.1.2 part 3.

3. **gameRecord.py**
   Defines the gameRecord class which is designed to receive the game state array from the DQN and save images of the game states during a training session and stitches them together at the end and provides a video of the training for observation.

6.3.2 Raspberry Pi

There are total 4 files all written in python and provide the locomotion solution for the robot and communication between RPI and PC.

1. **main.py**
   Defines all the threads, starts them, and then joins some of these threads and waits for them to finish.

2. **controlserver.py**
   Defines a socket server that communicates with roboSocket (section 6.3.1.1.1 part 4) and rpiClient (section 6.3.1.1.2 part 4). This provides communication between RPI and PC.

3. **motordriver.py**
   Defines the motor driver class and is written in python using the Adafruit library listed in section 6.1. It provides an API for driving the motors through the Pi Hat (motor shield).

4. **robocontrol.py**
   Written in python it defines the locomotion class roboControl which inherits the motorDriver class. It receives the connect 4 game slot number from the controlserver (part 2 of this section), calculates a straight line trajectory with help of inverse kinematics and moves the robot using an array of joint angles calculated.

7 References

[1] “Tabletop game industry,” *Wikipedia, the free encyclopedia,* 2020. Available: https://en.wikipedia.org/wiki/Tabletop_game_industry [Accessed: May 12, 2020]

[2] “Nao (robot),” *Wikipedia, the free encyclopedia,* 2020. Available: https://en.wikipedia.org/wiki/Nao_(robot) [Accessed: May 12, 2020]

[3] *Universal Robots.* Available: https://www.universal-robots.com/industries/ [Accessed: May 13, 2020]

[4] *Moley Robotics.* Available: https://www.moley.com/ [Accessed: May 13, 2020]

[5] J. McCabe, “Connect Four Robot,” *patrickmccabemakes.com*, 2014. Available: http://patrickmccabemakes.com/hardware/Connect_Four_Robot/report.pdf [Accessed: May 13, 2020]

[6] N. Maselli, “Homemade Chess Robot,” *Instructables Circuits*, 2017. Available: https://www.instructables.com/id/Homemade-Chess-Robot/ [Accessed: May 13, 2020]

[7] J. Tromp “A212693 - Number of legal 7 X 6 Connect-Four positions after n plies,” *The Online-Encyclopedia of Integer Sequences (OEIS),* 2012. Available: https://oeis.org/A212693 [Accessed: May 13, 2020]

[8] Dr M. Gianni “Introduction to Kinematics”, *ROCO318 - Mobile and Humanoid Robots, University of Plymouth,* 2019. [Accessed: May 19, 2020]

[9] “Layer activation functions,” *Keras*, Available: https://keras.io/api/layers/activations/ [Accessed: May 20, 2020]

[10] “Rectifier (neural networks),” *Wikipedia, the free encyclopedia,* 2020. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks) [Accessed: May 20, 2020]

[11] “Layer weight initializers,” *Keras*, Available: https://keras.io/api/layers/initializers/#available-initializers [Accessed: May 20, 2020]

[12] V. Kakaraparthi “Xavier and He Normal (He-et-al) Initialization,” Available: https://medium.com/@prateekvishnu/xavier-and-he-normal-he-et-al-initialization-8e3d7a087528 [Accessed: May 20, 2020]

[13] F. Fazal-Ur-Rehman “Machine_Learning_Maze_Solver,” *Github*, 2019, https://github.com/Faisal-f-rehman/Machine_Learning_Maze_Solver/blob/master/10538828_AINT351_CW_Report_Dec2019.pdf

[14] F. Fazal-Ur-Rehman “RoboCon (Vision),” *Github*, 2020, https://github.com/Faisal-f-rehman/10538828_RoboConVision

[15] F. Fazal-Ur-Rehman “RoboCon,” *Github*, 2020, https://github.com/Faisal-f-rehman/RoboCon

8 Appendix

8.1 Project Proposal

Student	Name	Faisal Fazal-Ur-Rehman
	Registration Number	10538828
Programme		BEng (Hons) Robotics
Module Code		PROJ324
Proposer		Self
Supervisor		Dr Mario Gianni
Proposal Title		RoboCon – Collaborative Gaming Robot
Objective: The objective of this project is to create a robotic arm that can play Connect 4 / Go 4 with a human opponent 		

8.1.1 Design

The robot will be designed using Autodesk Fusion360 and will be 3D printed using off the shelf 1.75mm PLA. Apart from the 3D printed structure there will be some use of off the shelf products such as bearings, a D-Type shaft and shaft couplers. The SCARA style design has been chosen to allow the maximum weight / force (when static), to be absorbed by the skeleton of the robot arm instead of the motors.

The end-effector shall be designed separately from the main arm and shall be a claw like minimalistic design mechanism. The end-effector should be designed to detach and easily re-attach to the main arm. This will allow addition of other games in the future by changing only the end-effector and software.

8.1.2 Control Equipment and Hardware

The robotic arm shall use 3 (JX PDI-6221mg) servo motors used for 3 rotational joints, 1 for the base joint, 1 between the 2 links (elbow joint) and 1 for the rotation of the end-effector (wrist joint). The robot shall also use 1 linear actuator (Actuonix L16 140mm 150:1), between the end-effector and the wrist joint to allow movement in the z-axis.

The robotic arm will use a Raspberry Pi (RPI) for the following using C/C++ programming language:

- computations of the game's algorithms
- locomotion computations
- To publish joint angle trajectories as a ROS publisher to a ROS subscriber in a python script.

The python script will send these joint angles to a PWM servo motor shield with a chip of its own to free up some processing load from the RPI.

A coin dispenser like mechanism will be designed and 3D printed to dispense the coins in an upright position. This will make it easier for the end-effector to grab the coins, with a simple claw like mechanism.

8.1.3 Testing

Testing shall be carried out for individual tasks as follows:

- Test the design and structure of the main arm after assembly.
- Test inverse kinematics and straight-line trajectory in Matlab.
- Test inverse kinematics and straight-line trajectory on the robot.
- Test the end-effector and disc dispenser individually.
- Test the end-effector and disc dispenser with the main arm (try pick and place).
- Test the Connect 4 game solver own its own, i.e. given a game scenario are the outputs correct?
- If implemented, test the vision system used to check the workspace, in different lighting environments.
- If implemented, check the feedback from the motors tie-up with the expected position of the motors.
- If implemented, check the current from the op-amp circuit is as expected, if the motor is forced to stop while running.
- If implemented, test the Connect 4 game solver created with machine learning, on its own i.e. given a game scenario are the outputs still correct?

Testing shall be carried out on the final product as well. This shall be done by inviting various people to play a Connect 4 game with RoboCon. Ideally this test should be carried out with a reasonable amount of time before the deadline so that there is enough time to make any amendments.

8.1.4 Extras for Robustness and Entertainment Value

After the above has been completed and the robot is able to play the game against a user, some of the following tasks can be implemented for improvement:

- Create a GUI interface with Python for control and to display the score of the game.
- Add a vision system to check the workspace of the robot and flag the robot to stop working if an unexpected object is detected with in the workspace.
- Hack the pot on the motors and read then current position from the pot on the motors to create a feedback loop.
- Design and create an op-amp circuit to measure current for the motors as a safety mechanism to limit the torque implemented by the motors.
- Use machine learning to try and teach the robot connect 4 / Go 4 game and compare the results with the results of pre-defined algorithms.

8.1.5 Estimated Cost

The estimated cost of this Robotic Arm is £275 which does NOT include equipment that maybe required for section 8.1.4 but it does include the following:

- 3D Printing Filament Spools
- Servo Motors JX PDI-6221MG

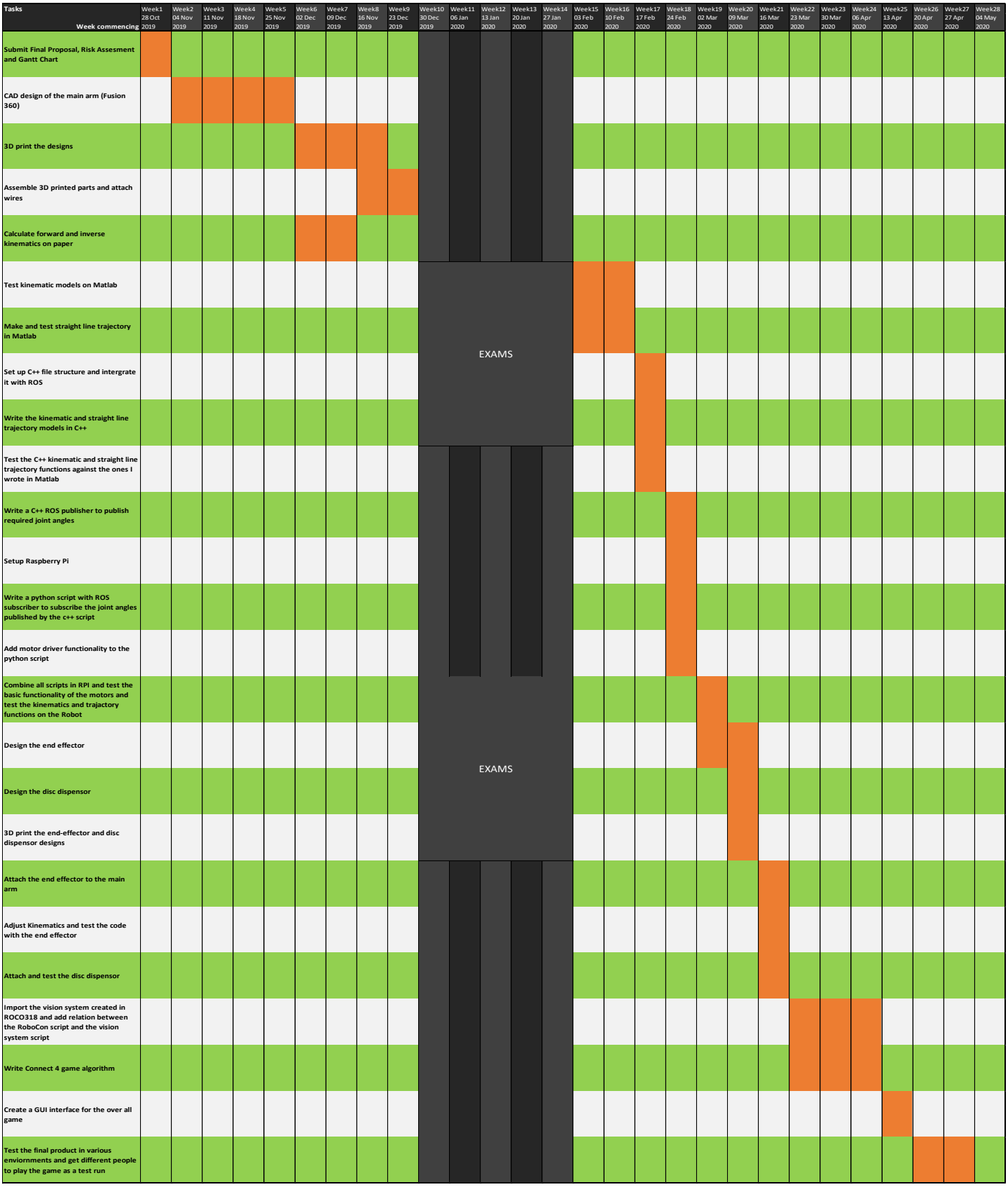
- Linear Servo Motor Actuonix L16 Actuator 140mm
- Thrust Bearings
- Ball Bearings
- Servo Extension cables
- Raspberry Pi 3b+
- Servo Motor Driver
- RPI Camera
- USB Camera

Please note that £250 of this cost has already been covered by the company I did my one year industry placement with.

8.1.6 The deliverables shall be divided as follows:

Tasks	Weight (%)
Project Main Body:	
1) Complete the design of the robot (main arm)	10
2) 3D print and assemble the robot (main arm)	5
3) The robot shall be able to move (main arm)	10
4) The robot shall move to a given position, in X and Y coordinates (main arm)	10
5) The robot shall move to a given position in a straight line (main arm)	5
6) Design the end-effector and the disc dispenser	10
7) 3D print, assemble and attach the end-effector and disc dispenser to main arm	5
8) The robot shall be able to perform pick and place operation	10
9) Create a Connect 4 game solver	10
10) Integrate the game solver with the robot to allow the robot to play the game and test the performance of the robot	5
Project Extras	
1) Create a GUI interface with Python for control and to display the score of the game.	10
2) Add a vision system to check the workspace of the robot and flag the robot to stop working if an unexpected object is detected with in the workspace.	10
3) Hack the pot on the motors and read then current position from the pot on the motors to create a feedback loop.	5
4) Design and create an op-amp circuit to measure current for the motors as a safety mechanism to limit the torque implemented by the motors.	10
5) Use machine learning to try and teach the robot connect 4 / Go 4 game and compare the results with the results of pre-defined algorithms.	10

8.2 Work Plan


Tasks
Week commencing
Week1 28 Oct 2019
Week2 04 Nov 2019
Week3 11 Nov 2019
Week4 18 Nov 2019
Week5 25 Nov 2019
Week6 02 Dec 2019
Week7 09 Dec 2019
Week8 16 Nov 2019
Week9 23 Dec 2019
Week10 30 Dec 2019
Week11 06 Jan 2020
Week12 13 Jan 2020
Week13 20 Jan 2020
Week14 27 Jan 2020
Week15 03 Feb 2020
Week16 10 Feb 2020
Week17 17 Feb 2020
Week18 24 Feb 2020
Week19 02 Mar 2020
Week20 09 Mar 2020
Week21 16 Mar 2020
Week22 23 Mar 2020
Week23 30 Mar 2020
Week24 06 Apr 2020
Week25 13 Apr 2020
Week26 20 Apr 2020
Week27 27 Apr 2020
Week28 04 May 2020
Submit Final Proposal, Risk Assesment and Gantt Chart
CAD design of the main arm (Fusion 360)
3D print the designs
Assemble 3D printed parts and attach wires
Calculate forward and inverse kinematics on paper
Test kinematic models on Matlab
Make and test straight line trajectory in Matlab
Set up C++ file structure and intergrate it with ROS
Write the kinematic and straight line trajectory models in C++
Test the C++ kinematic and straight line trajectory functions against the ones I wrote in Matlab
Write a C++ ROS publisher to publish required joint angles
Setup Raspberry Pi
Write a python script with ROS subscriber to subscribe the joint angles published by the c++ script
Add motor driver functionality to the python script
Combine all scripts in RPI and test the basic functionality of the motors and test the kinematics and trajectory functions on the Robot
Design the end effector
Design the disc dispensor
3D print the end-effector and disc dispensor designs
Attach the end effector to the main arm
Adjust Kinematics and test the code with the end effector
Attach and test the disc dispensor
Import the vision system created in ROCO318 and add relation between the RoboCon script and the vision system script
Write Connect 4 game algorithm
Create a GUI interface for the over all game
Test the final product in various enviornments and get different people to play the game as a test run
EXAMS
EXAMS