# AINT351Z - Learning, Planning, and Control of a Planar Arm Coursework 2021

Student No. 10618407

Monday 6th December, 2021

# Contents

# 1 Training Data Generation

## 1.1 Display workspace of the Revolute Arm

The code in figure 1 intialises the arm that is used. It plots an arm with length L1, and L2 which are shown below. The two joint angles are generated using the 'generateUniformDistribution' shown in figure 2.

```
% number of samples used for training
samples = 1000;

% define lengths of the arm sections and adding into matrix
L1 = 0.4;
L2 = 0.4;
armLen = [L1, L2];

% defining the origin of the plot
origin = [0,0];

%generate 1000 random values between 0 and pi for the arm joint locations
for i = 1:samples
    theta(1, i) = generateUniformDistribution(0, pi, 1, 1);
    theta(2, i) = generateUniformDistribution(0, pi, 1, 1);
end
```

Figure 1: Code for generating the 2DOF arm

These angles are between zero and pi, and are used in the provided 'RevoluteForwardKinematics2D' function. This is used to calculate and return the location of the arm's elbow and end effector, point1 and point 2 respectively. This can be seen in figure 1.

```
function [output] = generateUniformDistribution(min, max, rows, columns)

    output = min + (max - min) .* rand(rows,columns);

end
```

Figure 2: Generate Uniform Distribution Code

Once the data for the arm is generated, the useful range of this arm was outputted using the code in figure 3. The output plot is shown in figure 4.

```
% number of samples used for training
samples = 1000;

% define lengths of the arm sections and adding into matrix
L1 = 0.4;
L2 = 0.4;
armLen = [L1, L2];

% defining the origin of the plot
origin = [0,0];

%generate 1000 random values between 0 and pi for the arm joint locations
for i = 1:samples
    theta(1, i) = generateUniformDistribution(0, pi, 1, 1);
    theta(2, i) = generateUniformDistribution(0, pi, 1, 1);
end
```

Figure 3: Code to plot the workspace of the 2DOF arm

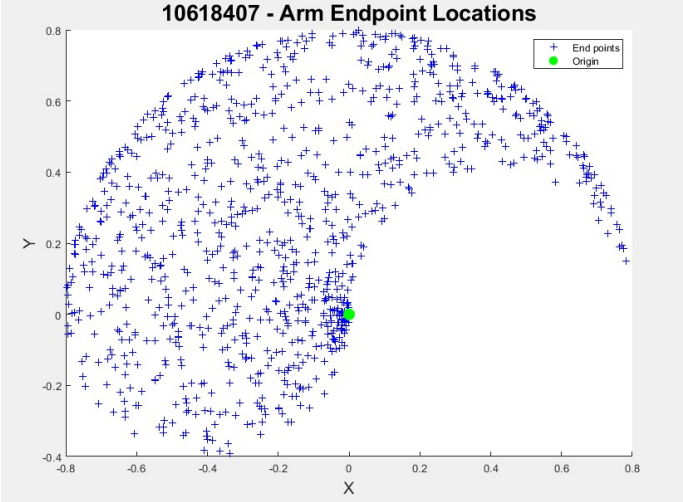



Figure 4: Output workspace of the 2DOF arm

Figure 4 shows the useful range, also known as the workspace, of the arm. This shows all the points that the end effector can reach. This arm is a 2DOF arm with endpoints highlighted with blue crosses and an origin highlighted by a green dot (at 0,0). This workspace is known as a tear-drop swirl pattern. The length from the end point to the origin is 0.8m. With the shoulder joint (origin of the robot), and the elbow joint being limited to range between zero and pi, the possible locations of the end effector are restricted. The shape of the resulting workspace would dictate the possible applications for the robot. It is important to keep this in mind for potential applications. The size of this workspace (approximately $5.8m^2$), will be important when deciding appropriate final scaling of the maze in later tasks.

# 2 Implement 2-layer network

## 2.1 Implement 2-layer network training

```
% used to load in the varibales from the workspace from other files
P1Data = load('Task_1_workspace.mat');

% epsiode variables
iterations = P1Data.samples;
tests = 1000;
alpha = 0.01;
hiddenUnits = 10;

% testing dataset
theta = P1Data.theta;
% training dataset
point2 = P1Data.point2;

% intialising the first pass matricies
a2 = zeros(2,1);
net2 = zeros(3,2);
output = 0;

% differentials with respect to W1 and W2
errGradwrtW1 = zeros(3,2);
errGradwrtW2 = zeros(1,3);

% initialising the wieghts to non-zero values
W1 = generateUniformDistribution(-0.1, 0.1, hiddenUnits, 3);
W2 = generateUniformDistribution(-0.1, 0.1, 2, hiddenUnits + 1);

% Organise training data from P1
x = point2;
xHat = [x; ones(1,iterations)]; % Augmenting the input for bais term
t = theta;

% creating the error and output vectors with a dynamically adjustatable
% length
outputVector = zeros(2, iterations);
errorVector = zeros(2, tests);
averageError = zeros(2, tests);
```

Figure 5: Setting up and importing constants and variables used for training the network

The weights are initialised using the 'generateUniformDistribution' function shown in figure 2. The values are initialised to be between -0.1 and 0.1. $W1$ is set up as a $'hiddenUnits'$ x 3 matrix , and $W2$ is a 2 x $'hiddenUnits + 1'$ matrix. The $'hiddenUnits'$ variable has been used so that the values can be adjusted easily to test different configurations. It is important for the weights to be close, but not equal to, zero. This would cause symmetry in the model which would mean all the inputs to the nodes are also zero. This would mean that no data is being transferred through any of the zero nodes.

```
% implement psuedo code here:
for c = 1:tests
    error = 0; % setting intial error to 0
    for i = 1:iterations

        % using the index of variables to train the data
        X = xHat(:, i);
        T = t(:, i);
        net = W1 * X;

        a2 = 1./(1+exp(-net)); % Calculate internal activations of layer 1
        a2Hat = [a2; 1;]; % Augment a2 to account for bias term in W2
        net2 = W2 * a2Hat;
        output = net2; % Calculate output activations of layer 2

        % calculating the delta terms
        delta3 = -(T-output); % Calculate output layer delta term
        w2bar = W2(:, 1:hiddenUnits);
        delta2 = (w2bar' * delta3) .* a2 .* (1- a2); % Back prop to calculate input (lower) layer delta term

        % calculating the differential terms
        errGradwrtW1 = delta2 .* X'; % Calculate error gradient w.r.t. W1
        errGradwrtW2 = delta3 .* a2Hat'; % Calculate error gradient w.r.t. W2

        % updating weights
        W1 = W1 - alpha * errGradwrtW1; % Update W1
        W2 = W2 - alpha * errGradwrtW2; % Update W2

        % adding to the error term
        error = error + (T - output)' * (T - output);
        outputVector(:, i) = output; %adding the current output to the output vector to allow plotting of the learning
    end
    errorVector(:, c) = error; % adding the error to allow plotting of errors to track training details
end

% averaging the errors for plotting
averageError = errorVector / iterations;
```

Figure 6: 2-layer network with 10 hidden units and a linear output

The figure above shows a neural network that works by defining the weights of the layers, using given training and target samples. The network calculates and recalculates the weights as many times as is defined by the 'tests' variable. In this instance, tests is 1000. After this, the values are used to calculate the internal and output activation layers of the network. The internal activation values are used to train the weight values using back propagation. The output error term is calculated by evaluating the difference between the actual output of the network and the output predicated by the target value of the network. The gradient of the weights is calculated by multiplying the input and output activations by the input values. This gradient is multiplied by the learning rate, alpha, and is taken away from the weights. This is used to refine the values of the weights. These weights are used as the new weights for the next pass of the network. The error term for all the samples for both W1 and W2 is calculated from the learning pass. This can be plotted to see the progress of the network throughout it's learning passes. This can be seen in figure 8 below.

## 2.2 Train network inverse kinematics

The training of the Inverse Kinematics was implemented using a feedforward pass that is shown in the code below.

```
% averaging the errors for plotting
averageError = errorVector / iterations;

% calc output of net
net2 = W1 * xHat;
a2 = 1 ./(1 + exp(-net2));
len = length(a2);
a2Hat = [a2; ones(1, len)];
outputVector = W2 * a2Hat;

armLen = P1Data.armLen;
origin = P1Data.origin;

% calculate end kinematics (tilda is used as point1 isn't used)
[~, point2] = RevoluteForwardKinematics2D(armLen, outputVector, origin);
```

Figure 7: Feedforward pass of the Network

The corresponding error over the number of iterations, and the output work space of the arm trained using the network, can be seen below in figure 8 and figure 9 respectively.

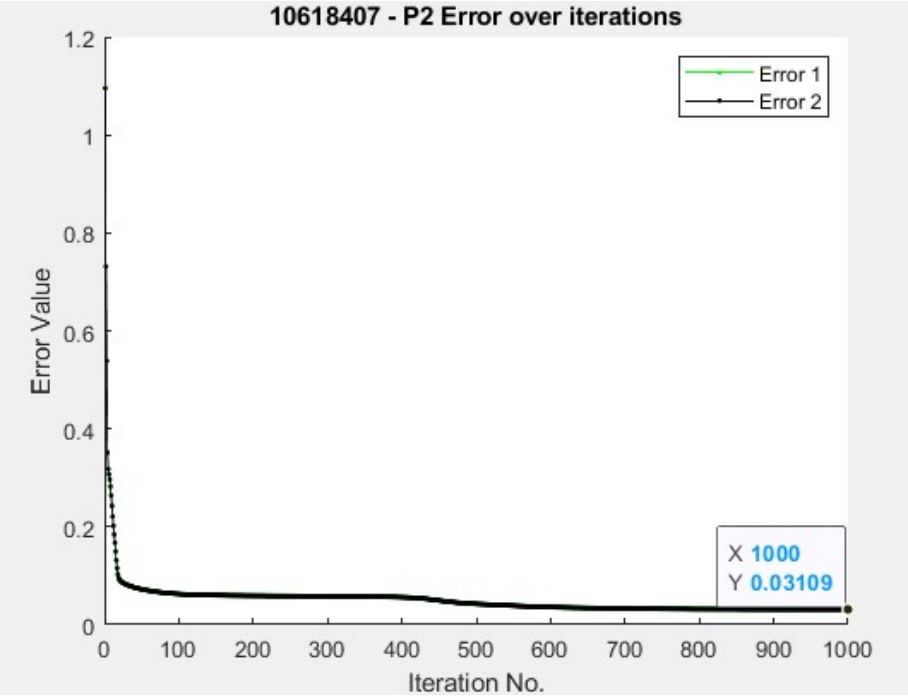



Figure 8: Error over iterations throughout network learning and feedforward pass

It can be seen from the diagram that at 1000 tests, the output error has dropped to 0.03109.

This has been evaluated against other configurations ie. different number of tests and hidden layers. These results are outlined below.

### 2.3 Test and improve the inverse model

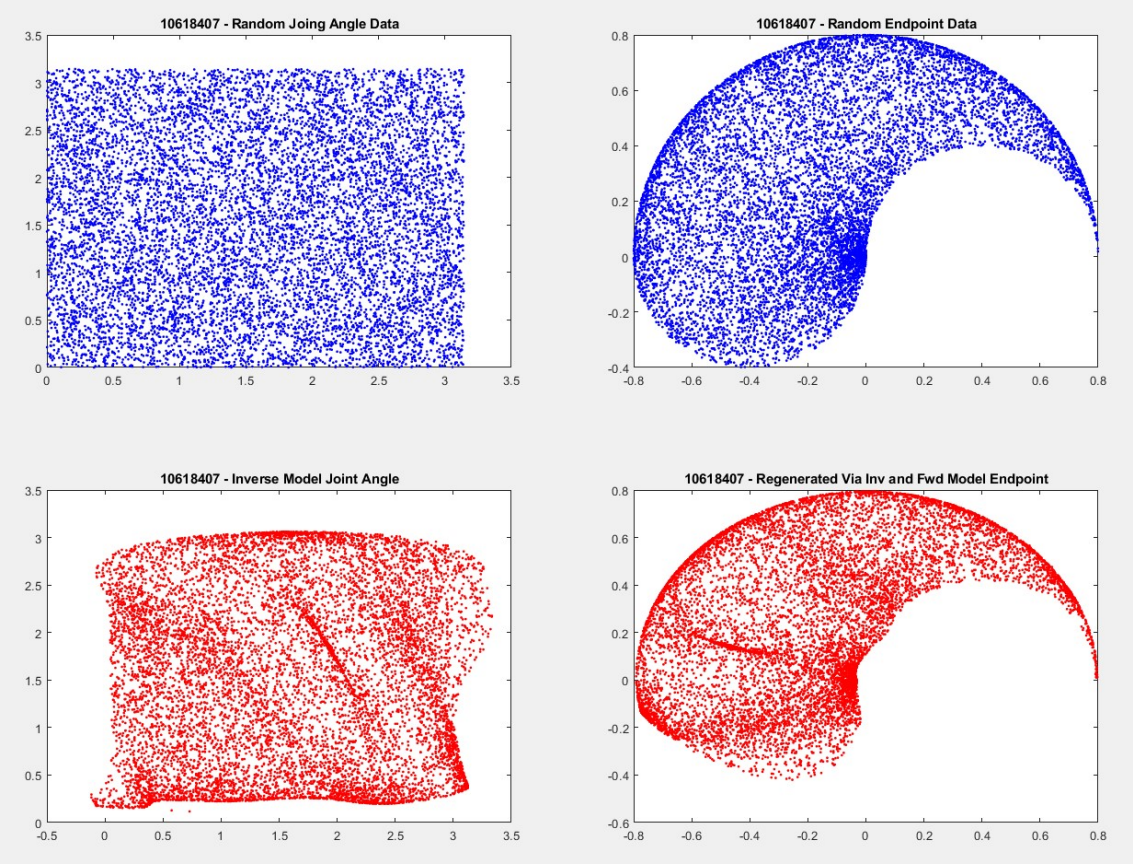



Figure 9: Training workspace of the endpoint of a 2 link arm before and after training

In figure 9, the top set of plots show the attainable locations of the arm along with the angles of each of the points used to get to those locations before the network was trained. The bottom set of these plots show the same data but after the network was trained.

This figure is significant because it shows the overall shape of the robot's workspace has been mostly preserved from the random angles initially used. There are however, some differences in the plot of the arm, primarily between $[0, -0.2]$ and $[0, 0.2]$. This could be for several reasons. I believe that it is primarily because of the size of the data set being used to train the model. Ideally, the model would be trained with a much larger data set which would give a more comprehensive set of data to train the model. This could alleviate those areas which contained gaps in the model.

To make a more comprehensive dataset for training the arm, we need to increase the number of angle samples. To increase the usefulness of the task, and to make the dataset more representative, it would be useful to be able to create more samples inside the larger end of the teardrop. This is where most of the arm's movement will take place. This would create finer granularity in the areas of the robot's workspace that would be most often utilised. This could lead to increased accuracy in the robot's movements within the maze. The data set could be further increased by using more representative data from 'accurate' paths through the maze. This would allow the robot to be able to train of data that is actually representative of the task it would be performing. The majority of the network learning has been genetic pre training of the arm. This allows it to plot joint angles to a given endpoint of the arm. It would but useful to train the arm on more maze solving data. This would allow it to become more effective in performing that specific task. This could allow the arm to perform better when attempting to solve the maze.

The figures below show the output of the neural network with different values for the number of tests run, alpha (learning rate), and the number of hidden nodes.

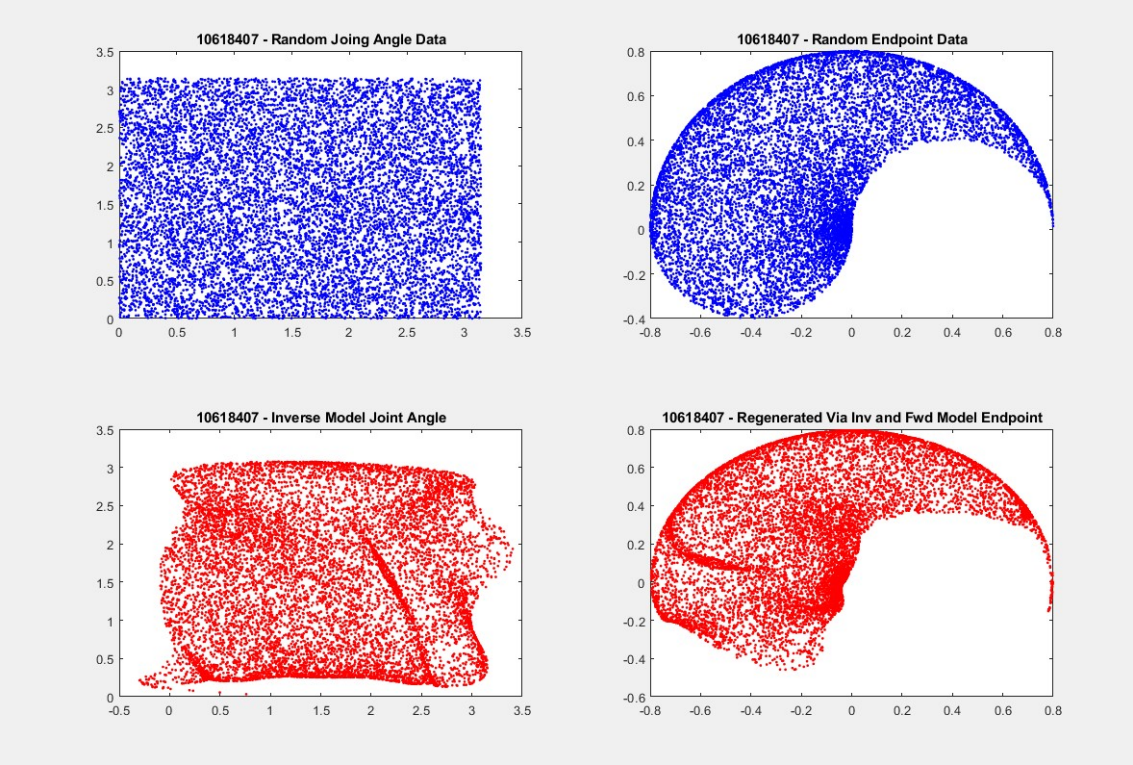



Figure 10: Training Workspace of the endpoint of a 2 link arm before and after training No. of hidden nodes changed from 10 to 100

As seen in figure 10, increasing the number of hidden nodes from 10 to 100 has a detrimental effect on the training of the robot's workspace. The disparity between the two can be seen mainly in the bottom of the larger end of the teardrop. It has lost the curved shape of the teardrop which negatively impacts on the size of the arm's workspace.

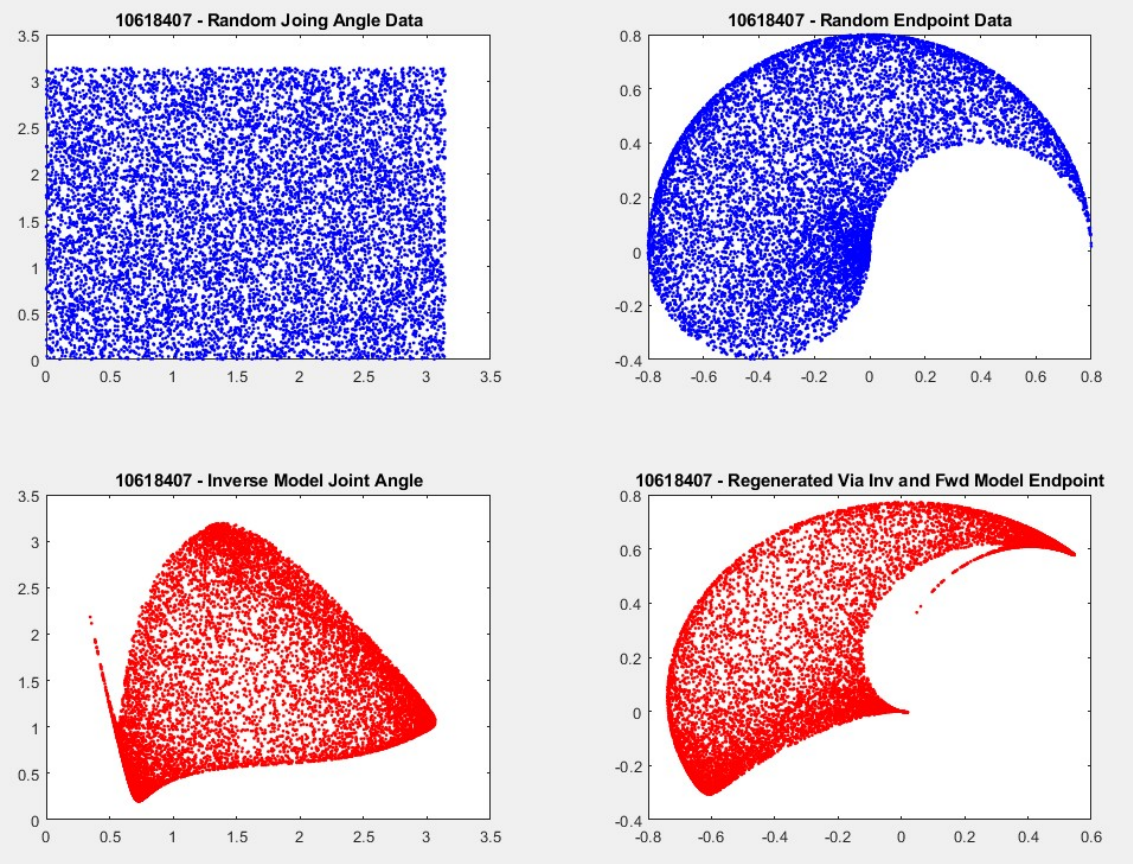



Figure 11: Training Workspace of the endpoint of a 2 link arm before and after training No. of hidden nodes changed from 10 to 2

As seen in figure 11, decreasing the number of hidden nodes from 10 to 2 also has a detrimental effect on the training of the robot arm. Unlike increasing the number of hidden nodes, the disparity between the two plot is far more pronounced. Although there are a few remenents of the previous teardrop shape, it is significantly warped and the workspace is drastically changed. The overall size of the robot's workspace decreased, making it less effective.

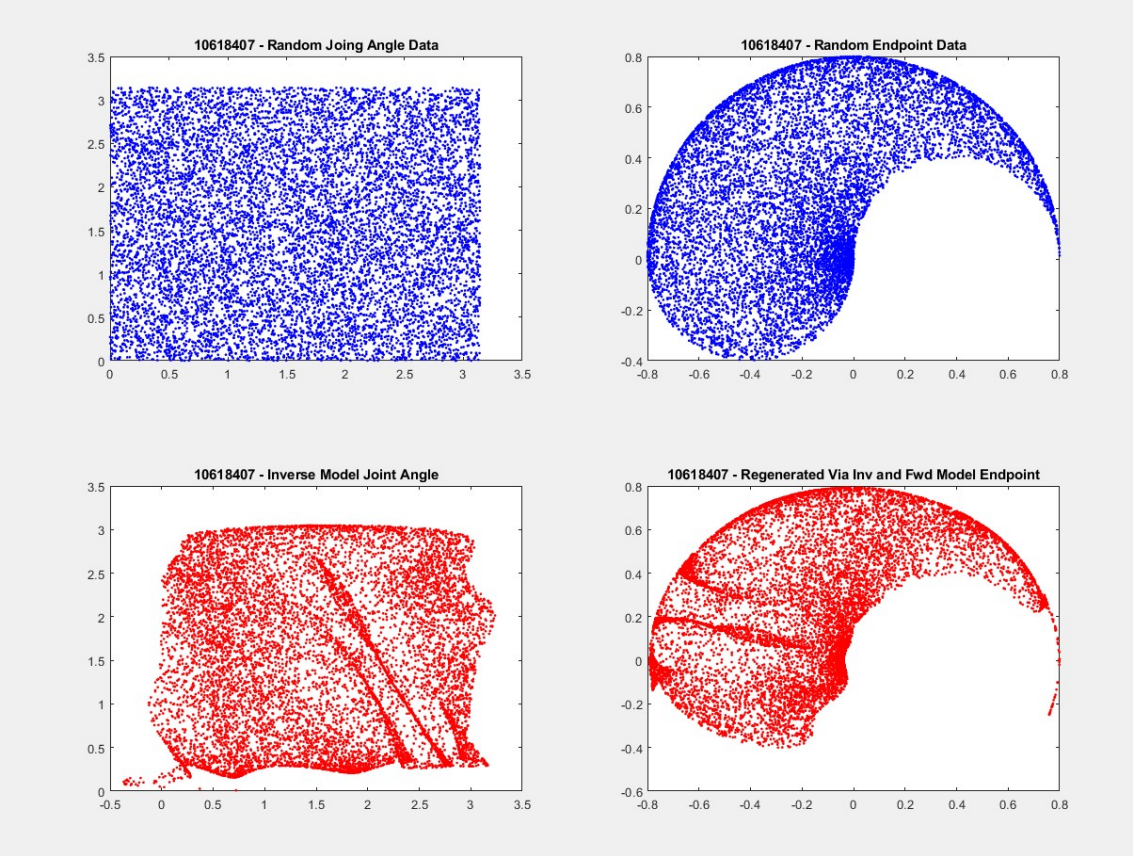



Figure 12: Training Workspace of the endpoint of a 2 link arm before and after training No. of tests changed to 10000

As shown in figure 12, increasing the number of tests leads to a very similar model shown with only 1000 tests, with mostly the same shape as the initial training. Although the area has been more thoroughly tested, there are some invalid results in the bottom left hand of the Inverse Model Joint Angle plot. This could be because the model was overtrained by using too many tests. Despite this, the overall output shape remains consistent to the origin. It takes far longer to train the network on each iteration and the output can be seen as invalid.

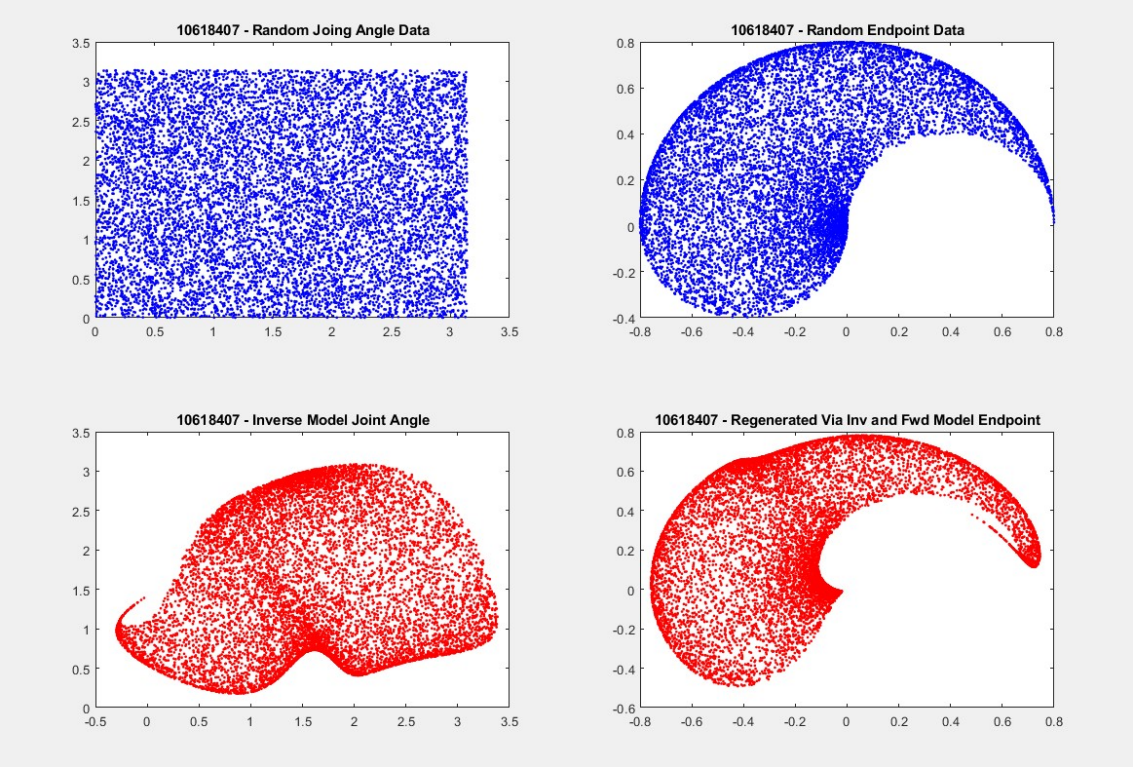



Figure 13: Training Workspace of the endpoint of a 2 link arm before and after training - Alpha changed to 0.0001

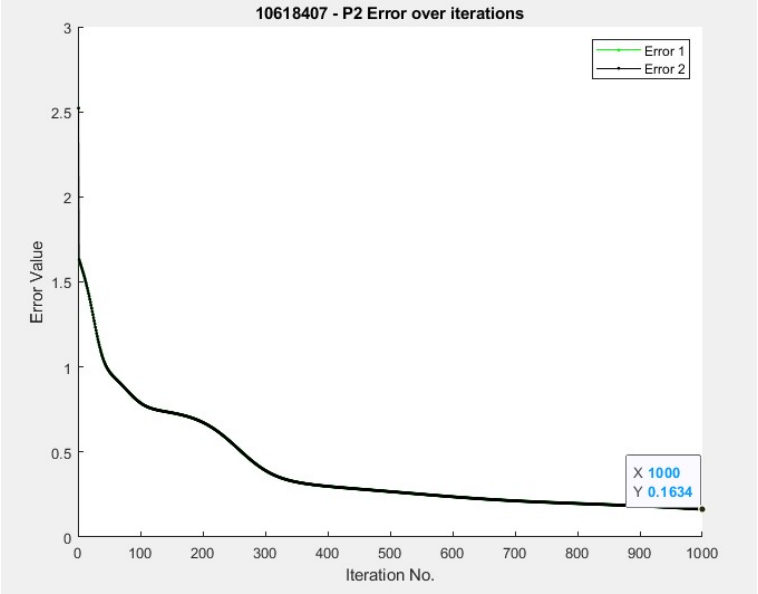



Figure 14: Error over iterations throughout network learning and feedforward pass - Alpha changed to 0.0001

As seen in figure 13 and figure 14, the values of alpha have a very large effect on the neural

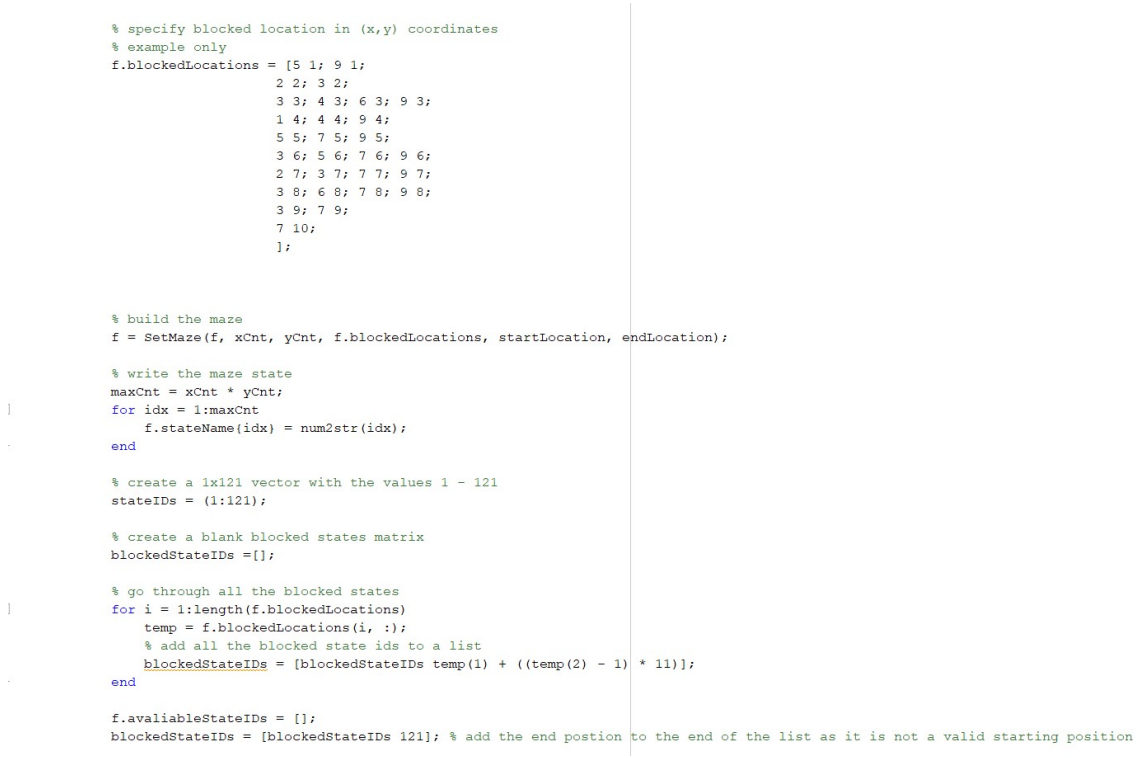
network. Not only is the workspace effected, but so is the error. The workspace is significantly less comparable to the original, it is more comparable to the workspace with only two hidden nodes. Whilst still retaining the original teardrop shape, it has been severely deformed around the top and at the thin end. It can also be seen, much like the workspace of the training with 10000 tests, there are some angles that the arm cannot achieve, outlined in the bottom left hand side of the Inverse Model Joint Angle diagram. This is undesirable as it is training the model with unattainable angles. The ending error can also be seen to be considerably larger than in figure 8. The error resulting from an alpha rate of 0.0001 is over five times larger than the original error.

# 3 Path through a maze

## 3.1 Random start state

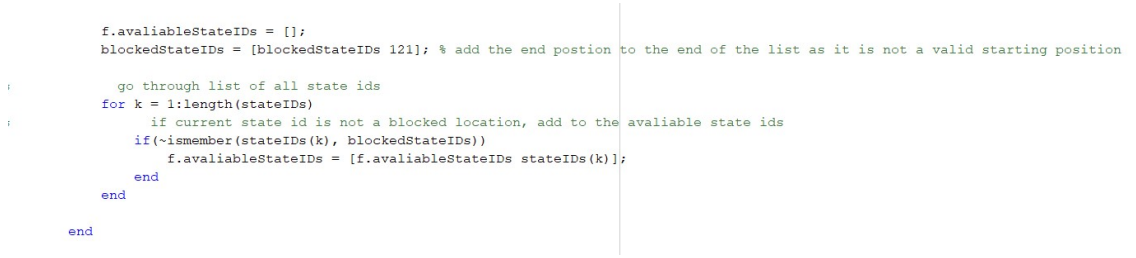
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% function  computes a random starting state
function startingState = RandomStartingState(f)

    % removes state 121 from avaliable state IDs
    if((max(f.avaliableStateIDs)) == 121)
        f.avaliableStateIDs(end) = [];
    end
    % picks a random value from the avaliable state ids list and
    % returns it
    startingState = f.avaliableStateIDs(randi(numel(f.avaliableStateIDs)));

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 15: Random starting state selected from 'avaliableStateIDs' (creation of 'avaliableStateIDs' is shown below)



```
% specify blocked location in (x,y) coordinates
% example only
f.blockedLocations = [5 1; 9 1;
                      2 2; 3 2;
                      3 3; 4 3; 6 3; 9 3;
                      1 4; 4 4; 9 4;
                      5 5; 7 5; 9 5;
                      3 6; 5 6; 7 6; 9 6;
                      2 7; 3 7; 7 7; 9 7;
                      3 8; 6 8; 7 8; 9 8;
                      3 9; 7 9;
                      7 10;
                      ];


% build the maze
f = SetMaze(f, xCnt, yCnt, f.blockedLocations, startLocation, endLocation);

% write the maze state
maxCnt = xCnt * yCnt;
for idx = 1:maxCnt
    f.stateName{idx} = num2str(idx);
end

% create a 1x121 vector with the values 1 - 121
stateIDs = (1:121);

% create a blank blocked states matrix
blockedStateIDs =[];

% go through all the blocked states
for i = 1:length(f.blockedLocations)
    temp = f.blockedLocations(i, :);
    % add all the blocked state ids to a list
    blockedStateIDs = [blockedStateIDs temp(1) + ((temp(2) - 1) * 11)];
end

f.avaliableStateIDs = [];
blockedStateIDs = [blockedStateIDs 121]; % add the end postion to the end of the list as it is not a valid starting position
```

Figure 16: Code to create blocked ids and 'blockedStateIDs'



```
f.avaliableStateIDs = [];
blockedStateIDs = [blockedStateIDs 121]; % add the end postion to the end of the list as it is not a valid starting position

  go through list of all state ids
for k = 1:length(stateIDs)
     if current state id is not a blocked location, add to the avaliable state ids
    if(~ismember(stateIDs(k), blockedStateIDs))
        f.avaliableStateIDs = [f.avaliableStateIDs stateIDs(k)];
    end
end

end
```

Figure 17: Code to create 'avaliableStateIDs'

As shown in the figures above, blocked state IDs are found by adding all the blocked IDs in coordinate form to a big list. The valid state IDs are found by creating a list from 1-121 and any state IDs that are also in the blocked IDs list get removed. This leaves you with just the available state ids which are used to calculate the valid random starting state IDs, amongst other things that are outlined later.

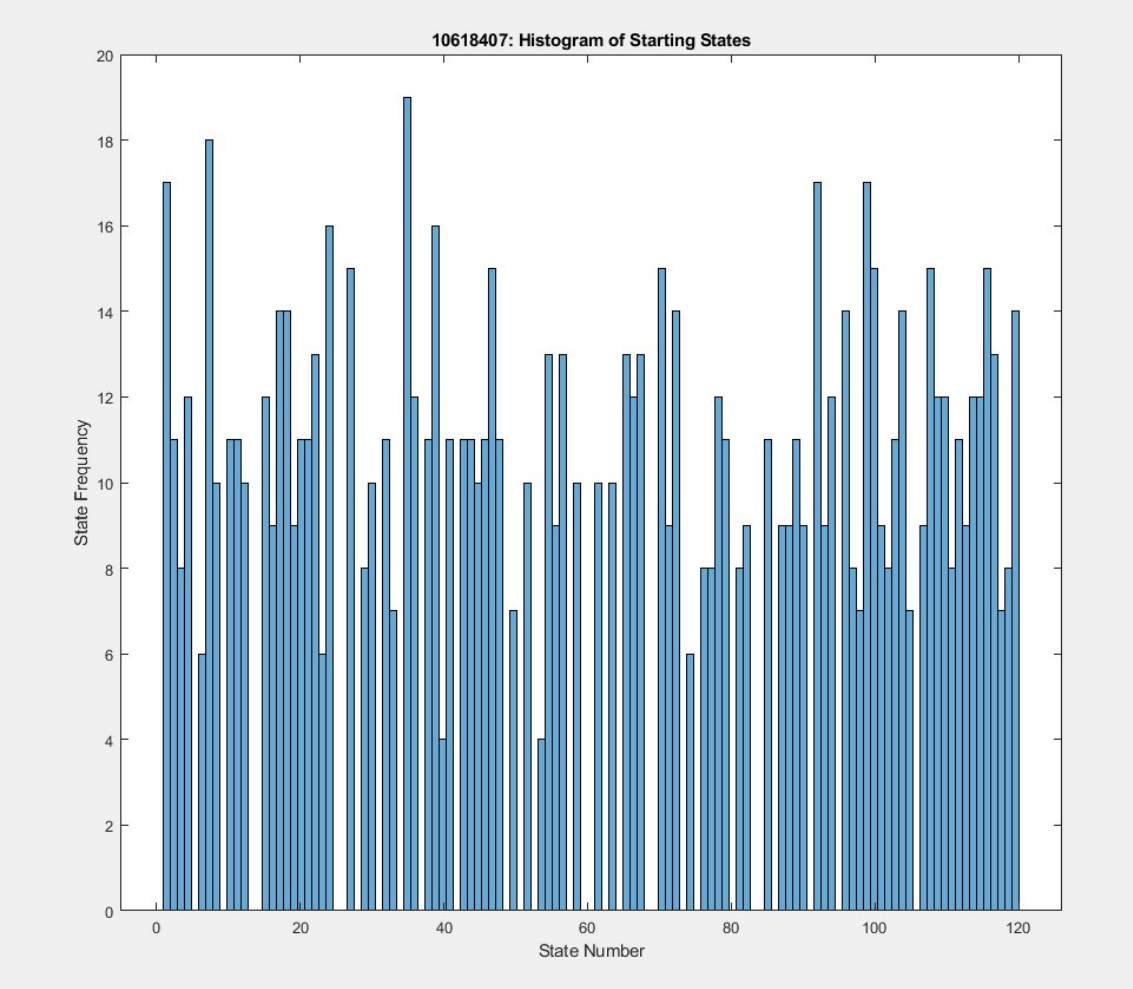



Figure 18: Histogram of 1000 generated starting state ids

The histogram in figure 18 shows 1000 randomly generated values for the maze. Each of the bins in the maze correspond to a state ID. There are gaps throughout the histogram. These gaps coincide with blocked or invalid starting state IDs. The values 1 and 121 are also blank as these are not valid starting IDs for the random starting state selector. The distribution of these values is not uniform, but as the number of random starting state ids increases, the distribution would tend towards a more uniform distribution.

The code to display this histogram is shown below.

```
startingStateHistVals = [];

for x = 1:1000
    startingStateHistVals = [startingStateHistVals maze.RandomStartingState()];
end

figure
histogram(startingStateHistVals, 121)
title('10618407: Histogram of Starting States');
xlabel('State Number');
ylabel('State Frequency');
```

Figure 19: Code to generate the starting state histogram

## 3.2 Build a reward function

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% reward function that takes a stateID and an action
function reward = RewardFunction(f, stateID, action)

    if (stateID == 120 && action == 2)||(stateID == 110 && action == 1)
        reward = 10;
    else
        reward = 0;
    end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 20: Reward function that evaluates current state and proposed action to allocate reward

The reward function is implemented as part of the update Q-Value calculations. The reward is only given if the current state and proposed action leads to the goal state. In that case, the reward is 10. If these conditions are not met, the reward stays at zero. The states and actions to get a reward can be seen in figure 20 and this function can be changed to give a different end state reward. Although not utilised in this setup, it is possible to allocate a small negative reward for each move that is not a rewarding move. This can also be used to train the network.

### 3.3 Generate the transition matrix

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% % build the transition matrix
% look for boundaries on the grid
% also look for blocked state
function f = BuildTransitionMatrix(f)

    % allocate
    f.tm = zeros(f.xStateCnt * f.yStateCnt, f.actionCnt);


    f.avaliableStateIDs = [f.avaliableStateIDs 121]

    % if the state ID in the transition table is valid, that cell is valid in
    % the transition function, if not, keep as all zeros

    % if value +11 is valid add a val + 11 to the first column
    % if value +1 is valid add a val +1 to the second column
    % if value -11 is valid add a val - 11 to the third column
    % if value -1 is valid add a val -1 to the forth column
    % else input the value into the column instead

    for m = 1:length(f.tm)

        % if the value is 121, the maze is complete and not transition function
        if(m == 121)
            f.tm(m, :) = 121;
            break
        end

        % if the sqaure is a valid space, not a blocked space,
        % add it to the transition state table, if it is a blocked
        % space, keep all the values at zero
        if(ismember(m, f.avaliableStateIDs))
            % if state +11 is valid (going north) add state +11 to transition table
            if(ismember(m + 11, f.avaliableStateIDs))
                f.tm(m, 1) = m + 11;
            else %if state + 11 is not valid (not space north) remain in current state
                f.tm(m, 1) = m;
            end
            % if state +1 is valid (going east) add state +1 to transition table
            if(ismember(m + 1, f.avaliableStateIDs) && ~((mod(m, 11) == 0)))
                f.tm(m, 2) = m + 1;
            else %if state +1 is not valid (no space east) remain in current state
                f.tm(m, 2) = m;
            end
            % if state -11 is valid (going south) add state -11 to transition table
            if(ismember(m - 11, f.avaliableStateIDs))
                f.tm(m, 3) = m - 11;
            else %if state -11 is not valid (no space south) remain in current state
                f.tm(m, 3) = m;
            end
            % if state -1 is valid (going west) add state -1 to transition table
            if(ismember(m - 1, f.avaliableStateIDs) && ~((mod(m, 11) == 1)))
                f.tm(m, 4) = m - 1;
            else %if state -1 is not valid (no space west) remain in current state
                f.tm(m, 4) = m;
            end
        end
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 21: Function to build the transition matrix from the size of the maze and the available state ids

The transition matrix (tm) is built in the function above. It starts by adding 121 to the available state IDs list as for the tm, it is a valid state. It then iterates through the length of the empty tm and looks at the current value. If the value is 121, all the values in the tm go back to 121, as it cannot move out that state. The second check is to ascertain if the current ID of the tm is a member

of the available state IDs. If the value isn't a valid state, then do nothing (the values all stay equal to zero). If the state is an available state id, then the function checks if it can move north, east, south, or west and updates the tm as appropriate. If one or more of the north, east, south, or west moves cannot be achieved, the tm puts back in the value of the state id for the relative movement. This can be seen with the comments in the tm function code. This is done for all values in the tm.

### 3.4 Initialize Q-values

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% init the q-table
function f = InitQTable(f, minVal, maxVal)

    % allocate
    f.QValues = zeros(f.xStateCnt * f.yStateCnt, f.actionCnt);

    % rand of the values times rand, + minVal offset
    f.QValues = minVal + (maxVal - minVal) * rand(f.xStateCnt * f.yStateCnt, f.actionCnt);

end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 22: Function to initialise the Q-Values

```
% init the q-table
minVal = 0.001;
maxVal = 0.1;
```

Figure 23: Q-Table initialisation values

The 'InitQTable' function starts by creating a matrix of all zeros, which is the size of the maze (11x11). Values are then randomly created between the 'minVal' and 'maxVal' seen in the figure above. These values are put in a matrix that is the size of the maze and overwrites the QValues table of all zeros.

### 3.5 Implement DynaQ-learning algorithm

```
% test values
state = 1;
action = 1;
e = 0.1;
alpha = 0.2;
gamma = 0.9;
trials = 100;
episodes = 1000;
terminationState = 121;
optimisedQTable = [];

stateplot = [];
pathTaken = [];
```

Figure 24: Initialisation of variables for the Q-Learning

```
maze = maze.InitQTable(minVal, maxVal);
for episodeIndex = 1:episodes
    startingState = maze.RandomStartingState();
    state = startingState;

    isRunning = 1;
    numberOfSteps = 1;

    while(isRunning)

        %greedy action selection
        action = maze.greedyActionSelector(maze.QValues, state, e);

        % resulting state
        resultingState = maze.tm(state, action);

        %sort out reward function
        reward = maze.RewardFunction(state, action);

        maze.QValues = maze.updatedQTable(maze.QValues, state, action, resultingState, reward, alpha, gamma);

        %termination of learning
        if(state == terminationState)
            stepsAcrossTrials(trialIndex, episodeIndex) = numberOfSteps;
            isRunning = 0;
            break % terminate current episode
        end

        % update the steps
        numberOfSteps = numberOfSteps + 1;

        %update the current state
        state = resultingState;
    end
end
```

Figure 25: Main Q-Learning Algorithm (runs in the main function)

The main Q-learning algorithm runs inside of the 'Main_P3_RunGridworld2021.m' file. The initial variables are setup as shown in figure 25 above. The purpose of this main algorithm is to update the Q-Values, which are based on a multitude of different variables. These include the

$\varepsilon$-greedy function, outlined in figure 26, the reward function, the learning rate alpha, and gamma. The while loop keeps the Q-learning updating the Q-Values and traversing the maze until the goal state is reached. Once this occurs, the Q-learning will exit that loop and restart the learning from a new random starting state, with the updated Q-Values from the old run. This loop will continue until it has completed the full episode number of loops, continuing to optimise the Q-Values whilst solving the maze.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Greedy action selector
function action = greedyActionSelector(f, QTable, state, e)

    x = rand; %picks a random value between 0 and 1

    % if value 'x' is greater than 1 - e, pick random action (Exploration)
    if x > (1 - e)
        action = randi([1, 4]);
    else
    % if value 'x' is less than 1 - e, pick the value with highest
    % Q value, that is the action to use
        [~, action] = max(QTable(state, :));
    end
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 26: $\varepsilon$-greedy Action Selector

The $\varepsilon$-greedy function chooses the next action given the current state. There are two possible options for which state the action selector can choose: the action with the highest Q-Value or a random action. This is decided by selecting a random number between zero and one. If the number is greater than one minus $\varepsilon$ the random action is chosen. If the number is less than one minus $\varepsilon$, then the action with the highest Q-Value is chosen. This is to allow for some random exploration for new or not fully optimised Q-Values.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Update QTable
function QValues = updatedQTable(f, QValues, state, action, resultingState, reward, alpha, gamma)

    % finds the maximum action value for a next state
    maxValue = max(QValues(resultingState, :));
    % finds the current Q value for the current state and action
    currentQValue = QValues(state, action);

    % updates Q table
    QValues(state, action) = currentQValue + alpha * (reward + (gamma * maxValue) - currentQValue);

end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 27: Update Q-Table function

The Update Q-Table function is called to update the Q-Values in the Q-Table at the end of each loop. These Q-Values are saved and optimised between each episode, helping the algorithm to learn.

### 3.6 Run DynaQ-learning

```
% Runing Q Learning

for trialIndex = 1:trials
    maze = maze.InitQTable(minVal, maxVal);
    for episodeIndex = 1:episodes
        startingState = maze.RandomStartingState();
        state = startingState;

        isRunning = 1;
        numberOfSteps = 1;

        while(isRunning)

            %greedy action selection
            action = maze.greedyActionSelector(maze.QValues, state, e);

            % resulting state
            resultingState = maze.tm(state, action);

            %sort out reward function
            reward = maze.RewardFunction(state, action);

            maze.QValues = maze.updatedQTable(maze.QValues, state, action, resultingState, reward, alpha, gamma);

            %termination of learning
            if(state == terminationState)
                stepsAcrossTrials(trialIndex, episodeIndex) = numberOfSteps;
                isRunning = 0;
                break % terminate current episode
            end

            % update the steps
            numberOfSteps = numberOfSteps + 1;

            %update the current state
            state = resultingState;
        end
    end
end

optimisedQTable = maze.QValues;
```

Figure 28: Trial loops for Q-learning episodes

A Q-learning trial is where the Q-Table is renewed with a freshly initialised Q-Table. The algorithm is then run again. This allows us to visualise when the program reaches the end state in the minimal number of steps. This could lead to a reduction of needed episodes and constant changes to reach the end state.

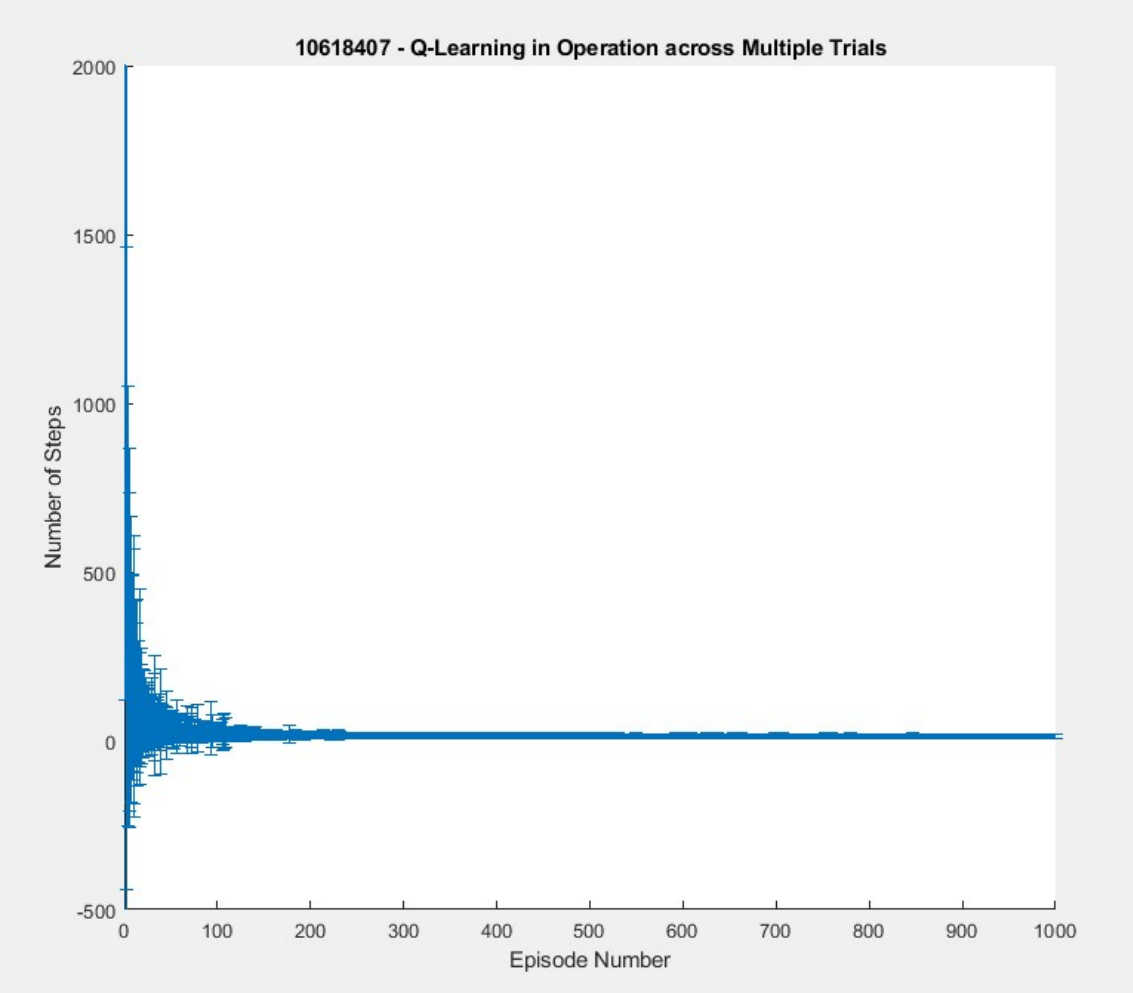



Figure 29: Error plots of mean and standard deviation of the Q-learning over multiple trials

Figure 29 above shows the error plots of the mean and standard deviation of the steps taken by the Q-Learning algorithm against the Episode Number. From Episode 250 onwards, the number of steps seems to remain consistent. This shows that the Q-Learning algorithm has likely found the most efficient way to get to the end state from any starting state.

## 3.7 Exploitation of Q-values

```
isRunning = 1;
state = 1;
finalNumberOfSteps = 1;
statePlot = [];

while(isRunning)

    for stepNumber = 1:21
        pathTaken(stepNumber) = state;

        % create a 2xN matrix, N is the steps

        % plotting x coordinate of pathTaken
        if(mod(state, 11) == 0)
            statePlot(1, finalNumberOfSteps) = 10.5; % this places cross in the centre of the box
        else
            statePlot(1, finalNumberOfSteps) =  (mod(state, 11)) - 0.5; % places cross in centre of the box
        end

        % plotting y coordinate of pathTaken
        statePlot(2, finalNumberOfSteps) = ((state - (statePlot(1, finalNumberOfSteps) + 0.5)) / 11) + 0.5; %scales and plots the value in the centre


        % greedy action selection - no exploration so e = 0
        action = maze.greedyActionSelector(optimisedQTable, state, 0);

        % resulting state
        resultingState = maze.tm(state, action);

        % sort out reward function
        reward = maze.RewardFunction(state, action);

        % termination of learning
        if(state == terminationState)
            isRunning = 0;
            break % terminate
        end

        % update the steps
        finalNumberOfSteps = finalNumberOfSteps + 1;


        %update the current state
        state = resultingState;
    end
end
```

Figure 30: Running Q-Learning with epsilon set to zero

This function is almost identical to the previous run through of the Q-Learning Algorithm except for two key differences. The first difference is that the epsilon value for the $\varepsilon$-greedy action selector is set to zero. This means that only the Q-Values are used to reach the end state (no exploration is done). The other key difference is that the Q-Table and values are not updated at each run through of the algorithm. This means that the Q-learning alogirthm has found the most optimised route to take through the maze. Three examples of optimal roots are shown in figure 31, figure 32, and figure 33.

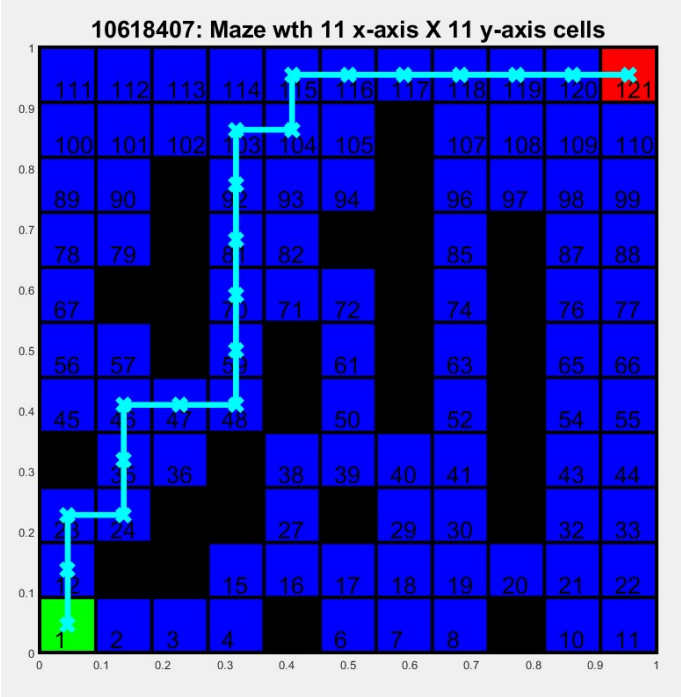



Figure 31: Plot of maze with path from highest Q-Values - solution 1

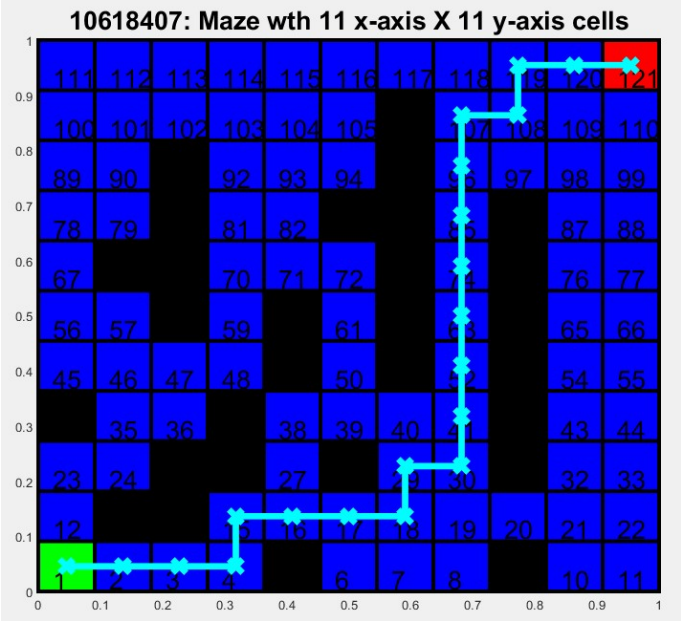



Figure 32: Plot of maze with path from highest Q-Values - solution 2

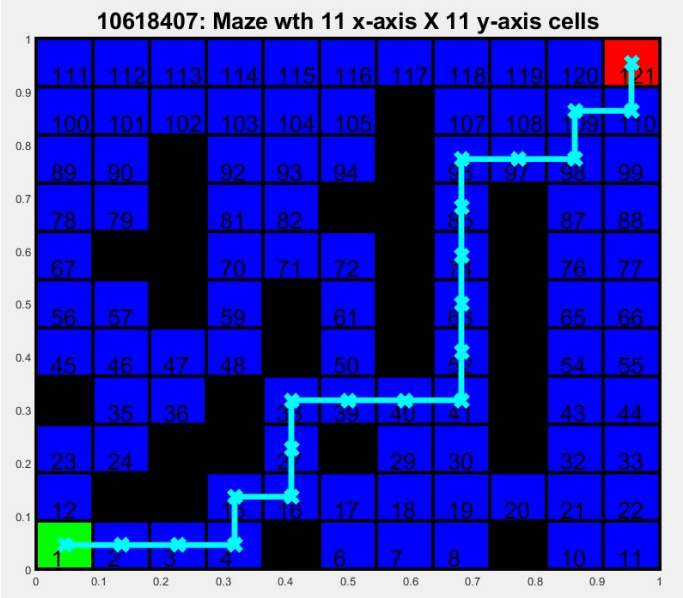



Figure 33: Plot of maze with path from highest Q-Values - solution 3

It can be seen from figure 31, figure 32, and figure 33 that there are multiple solutions to solving this maze in the shortest number of steps. The above figures are merely demonstrations of three of these possible solutions.

# 4 Move arm endpoint through maze

## 4.1 Generate kinematic control to revolute arm

```
% Use maze path locations as inputs for the robot arm inverse kinematic
% model (do this using the inverse kinematic model that we created earlier.
% Use the forwards kinematic function with the angles as inputs to
% calculate the elbow and endpoint positions.
% The MLP inverse model will not be perfect but try to get it to work as
% well as possible.
P1Data = load('Task_1_workspace');
P2Data = load('Task_2_workspace');
P3Data = load('Task_3_workspace');

%define limits of the maze
limits = [-0.7 -0.1; -0.2 0.3;];
%draw maze
maze = CMazeMaze11x11(limits);
%drawing maze on arm workspace to check it is workspace of calulated arm
hold on
maze.DrawMaze();
plot(P1Data.point2(1,:), P1Data.point2(2,:), 'r+');
title('10618407: Maze inside workspace of robot arm');
```

Figure 34: Setting up maze limits and plotting it in the workspace of the arm

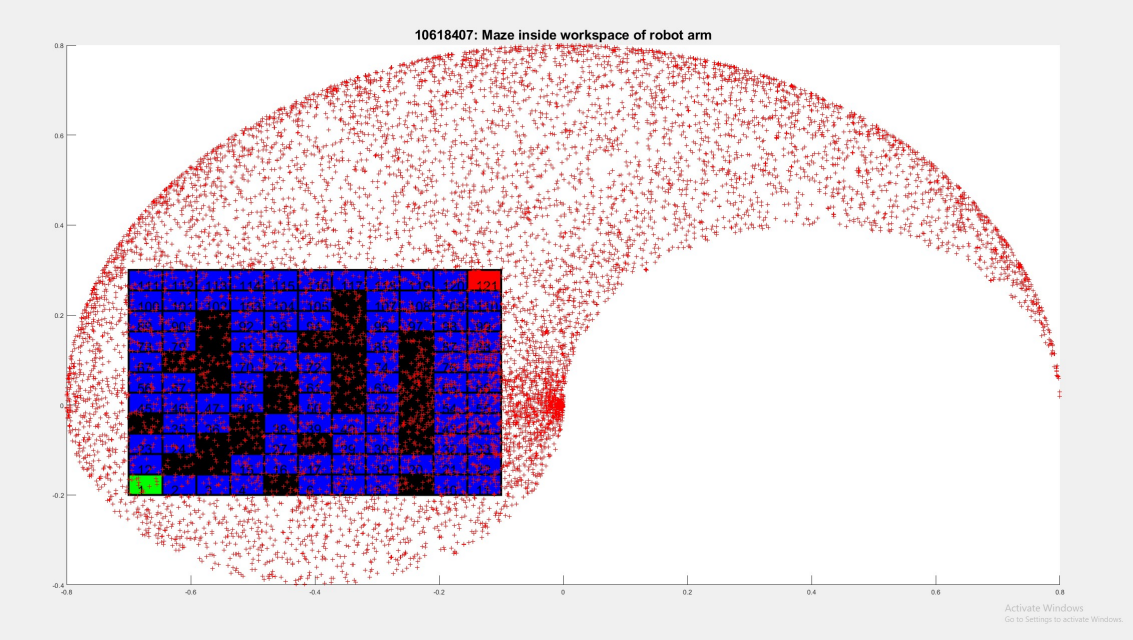



Figure 35: Plot of robot arm workspace encompassing the maze

Figure 34 and figure 35 show the scaling of the maze is correct. This is shown by the maze falling within the workspace of the robot arm. This means that the arm will be able to reach all locations in the maze.

```
% scaling the X and Y coordinates for the mapping
scaledXYCoords(1,:) = (P3Data.statePlot(1,:)*(limits(1,2)-limits(1,1))+limits(1,1));
scaledXYCoords(2,:) = (P3Data.statePlot(2,:)*(limits(2,2)-limits(2,1))+limits(2,1));

xHat = [scaledXYCoords; ones(1, length(scaledXYCoords))];

net2 = P2Data.W1 * xHat;
a2 = 1 ./(1+exp(-net2));
len=length(a2);
a2Hat = [a2; ones(1, len)];
outputVector = P2Data.W2 * a2Hat;

armLen = [0.4, 0.4];
origin = [0, 0];
[elbow, hand] = RevoluteForwardKinematics2D(armLen, outputVector, origin);
```

Figure 36: Calculating the path's joint angles using a forward pass of the trained neural network

Once the locations for the path are scaled, these new coordinates are passed into a feed forward pass of the neural network to calculate the joint angles. After the joint angles are calculated, these are then transformed into the positions of the end effector and joint via the 'RevoluteForwardKinematics2D' function. This generates the points to be plotted onto the graph.

```
hold on
maze.DrawMaze();
h=title('10618407 - Animation of revolute arm moving along path in maze');
h.FontSize=20;
h=xlabel('Horizontal position');
h.FontSize=10;
h=ylabel('Vertical position');
h.FontSize=10;
plot(hand(1,:), hand(2,:), 'y+-', 'Linewidth', 3);
pause(5);

for idx = 1:length(hand)
    xConfig = [origin(1), elbow(1,idx), hand(1,idx)];
    yConfig = [origin(2), elbow(2,idx), hand(2,idx)];
    h1 = plot(xConfig,yConfig, 'g-', 'LineWidth',3);
    h2 = plot(hand(1,idx), hand(2,idx), 'bo','MarkerSize',10, 'MarkerFaceColor', '#E2D3A8');
    h3 = plot(elbow(1,idx), elbow(2,idx), 'wo' , 'MarkerSize',10, 'MarkerFaceColor', 'k');
    h4 = plot(origin(1,1), origin(1,2), 'r+' , 'MarkerSize',10, 'MarkerFaceColor', 'cy');
    pause(0.3);
    delete(h1);
    delete(h2);
    delete(h3);
    delete(h4);
end

save('Task_4_workspace'); % saves workspace for other tasks
```

Figure 37: Calculating the path's joint angles using a forward pass of the trained neural network

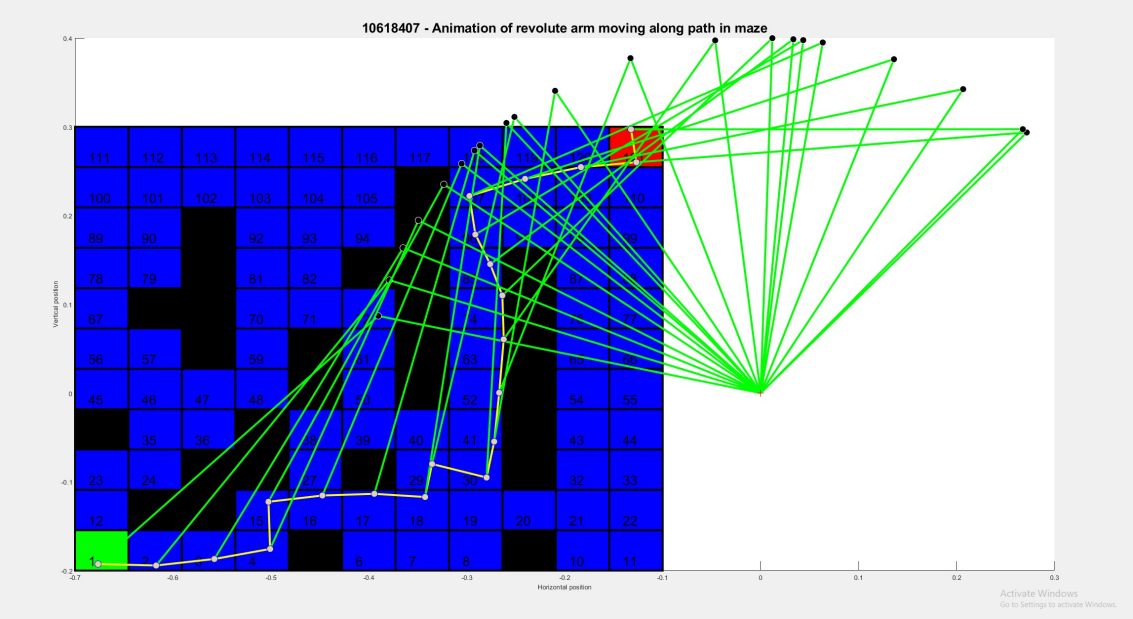



Figure 38: Superimposed Arm locations for each step on the maze

Figure 38 and figure 37 above show the path travelled by the arm. The plot code takes the locations calculated by the code in figure 36 and plots them onto the maze workspace. The code iterates through the various locations and outputs them onto the maze. After this, the arm is

deleted so the next one can be shown. This can be seen in the video linked below. Before the main iterative loop, the yellow line is plotted onto the maze. This line shows the path that the arm is taking through the maze. It can be seen that the plotted line falls within the correct states on the maze. This shows that the Q-Learning and Inverse Kinematic model have both produced their expected outputs, and have worked as intended.

### 4.2 Animate revolute arm movement

The animation of the arm moving through the maze can be seen here: - Robot Arm Traversing Through The Maze