AINT351 – MACHINE LEARNING

10555972 | Computer Science | 12th December 2019

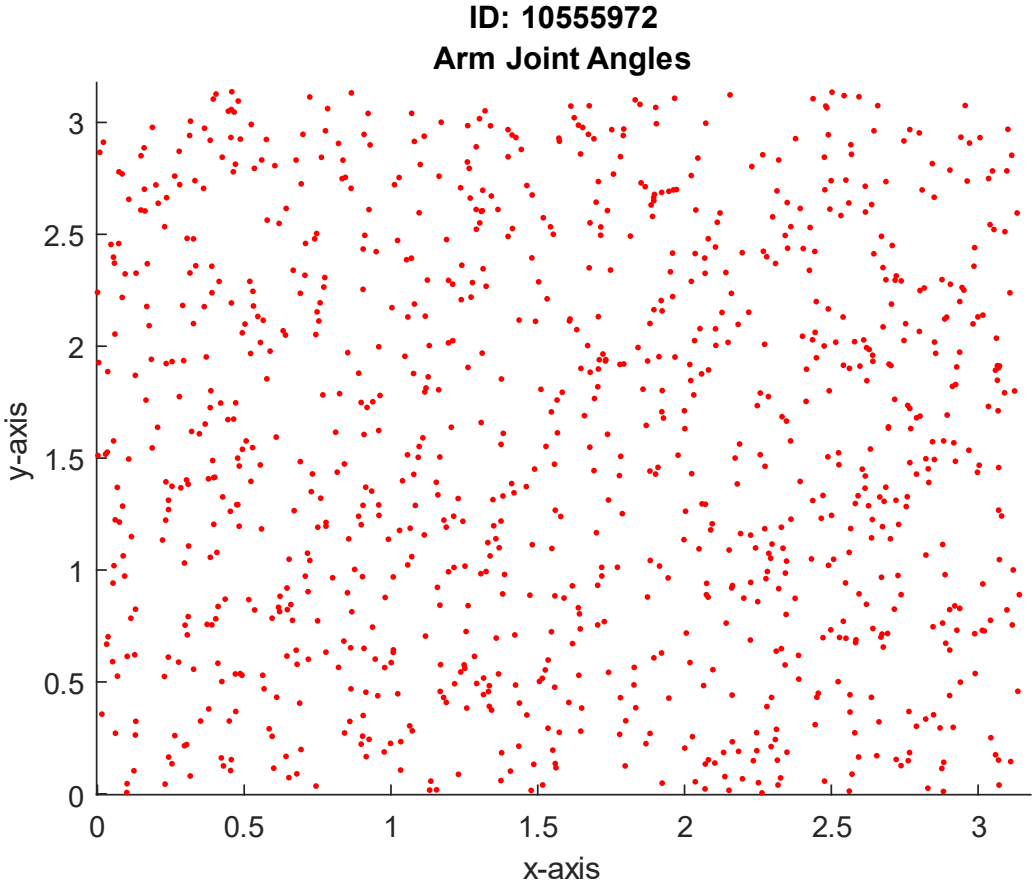
Contents

1. TRAINING DATA GENERATION

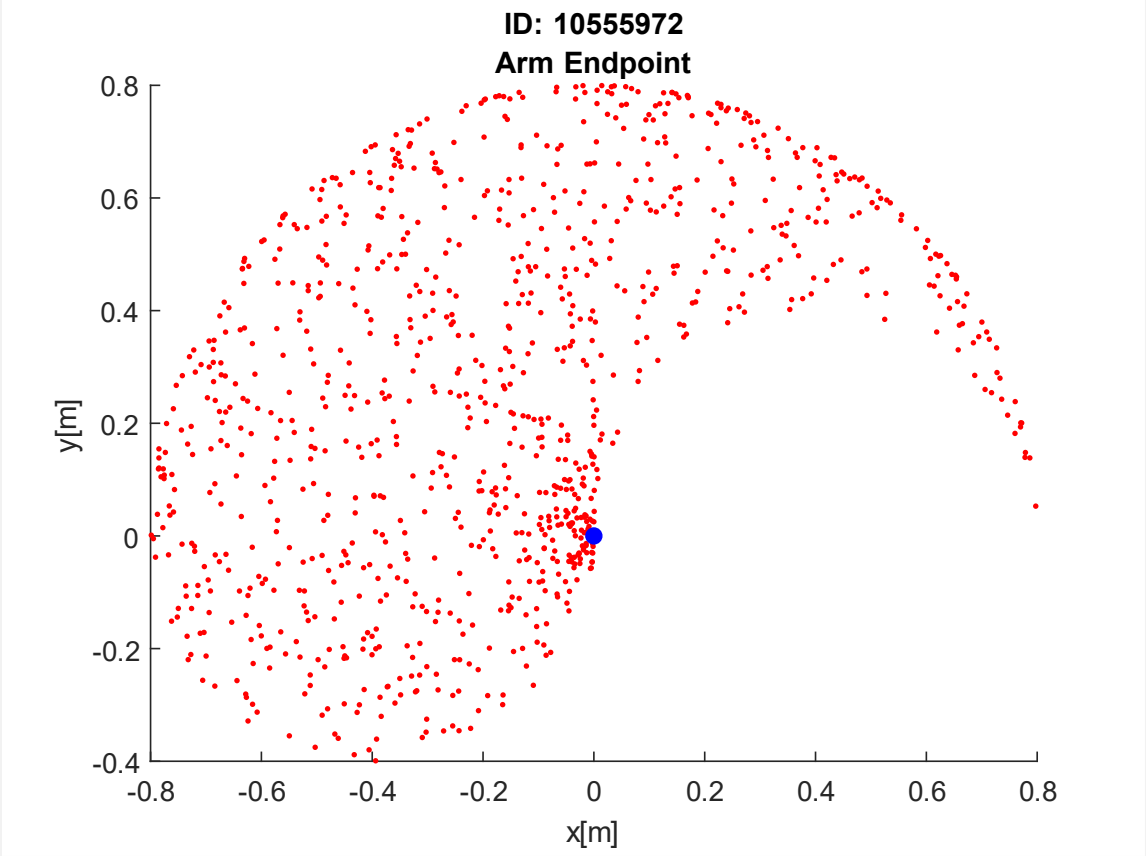
The provided Revolute Forward Kinematics 2D function is used to be able to output the arm end points by using the arms joint angles, the length of the arms and the base origin of the arms. The end points that are generated from this function will then later be used to train a neural network.

1.1. DISPLAY WORKSPACE OF REVOLUTE ARM

To display the workspace of the revolute arm I generated a random dataset between the values of 0 and π. The dataset had uniform distribution and contained 2x1000 samples. This dataset contains the angles that will be passed through the forward kinematics to calculate the end points and show the workspace of the arm.



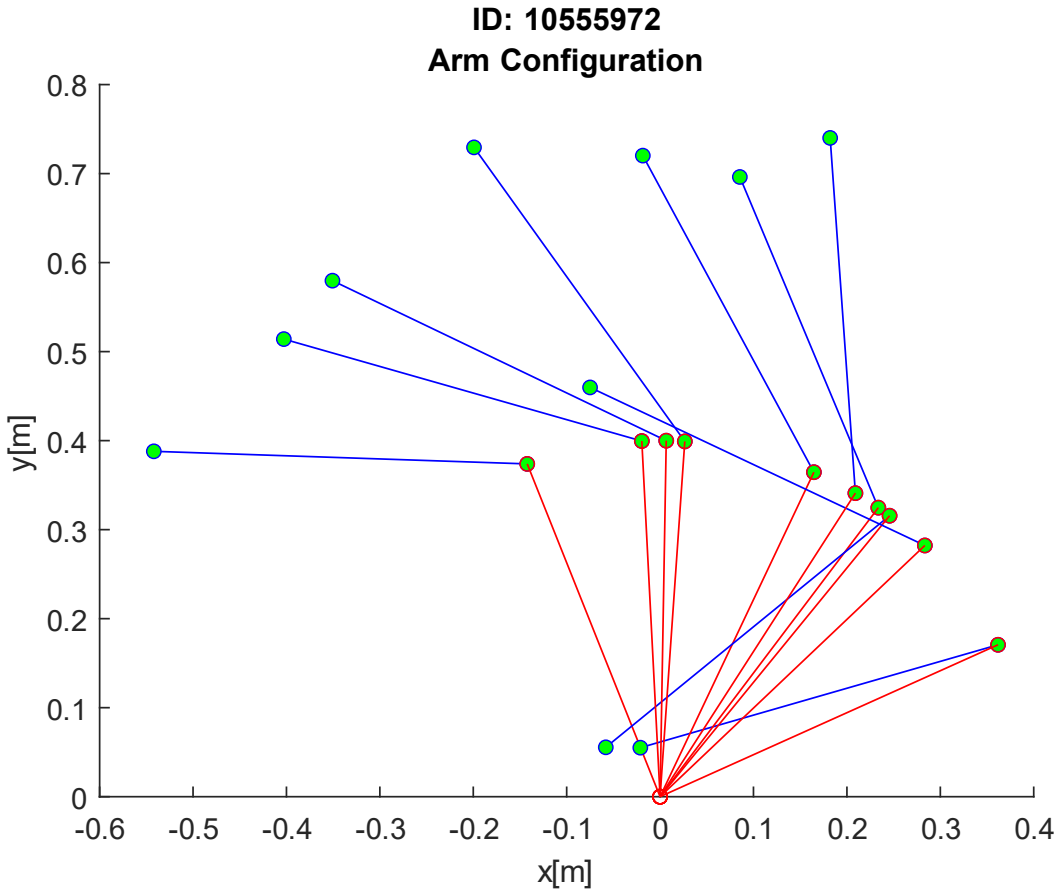
I then set the parameters for the Revolute Forward Kinematics function to use, the arm lengths for before and after the elbow were set to 0.4 and the base origin coordinates were set to (0, 0). Passing in these values and the joint angles previously generated, the function produces the correspdoning end points. Due to the arm only having 2 degrees of freedom the useful range of the end points is rather limited. This could be increase by adding a third joint to the arm, allowing it to move freely throughout the plane.




```
% Defining variables
armLength = [0.4;0.4];
baseOrigin = [0, 0];
samples = 1000;

% Generating 2 x samples between 0 - pi
angles = pi * rand(2,samples);
% Run angles through forward kinematics
[P1, P2] = RevoluteForwardKinematics2D(armLength, angles, baseOrigin);

% Plot randomly generated angles
figure
hold on
title({'ID: 10555972', 'Arm Joint Angles'});
xlabel('x-axis');
ylabel('y-axis');
plot(angles(1,:), angles(2,:), 'r.');

% Plot end points
figure
hold on
title({'ID: 10555972', 'Arm Endpoint'});
xlabel('x[m]');
ylabel('y[m]');
plot(P2(1,:), P2(2,:), 'r.')
plot(baseOrigin(1), baseOrigin(2), 'b.', 'MarkerSize', 20);
```

1.2. CONFIGURATIONS OF A REVOLUTE ARM

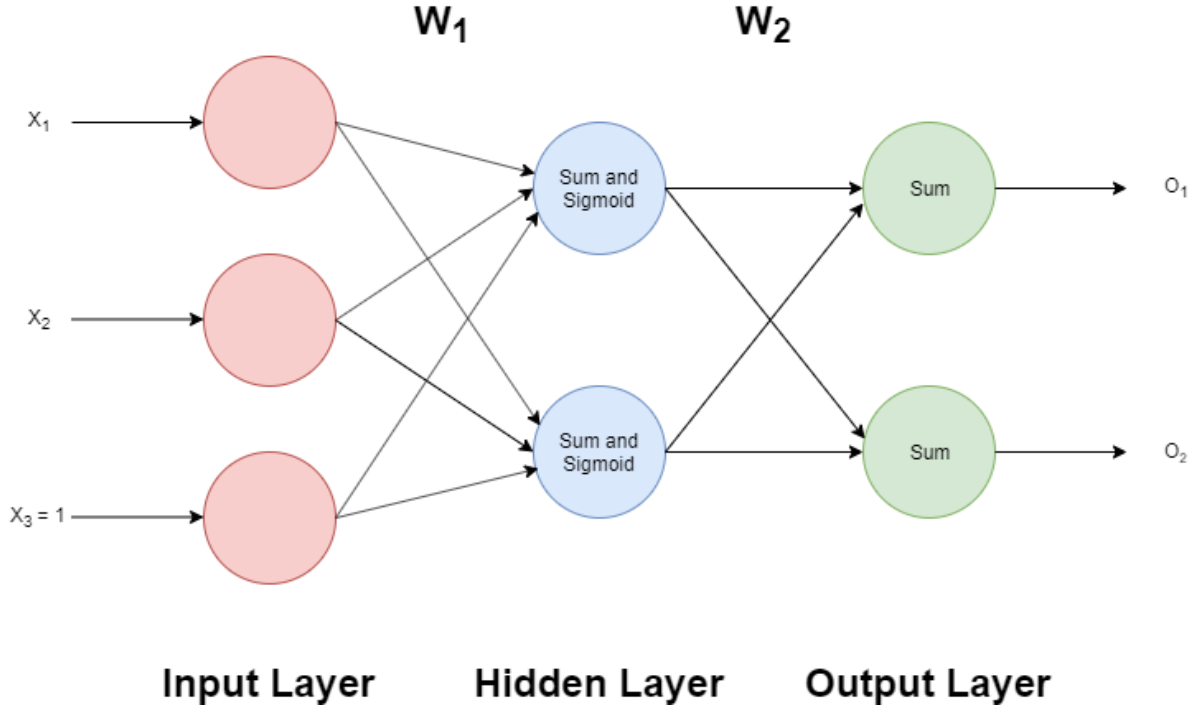
To help illustrate the arm configurations I have plotted 10 elbow and end points locations and the arm between them. This has been done by using 10 of the randomly generated set of angles previously and running it through the forward kinematics function. This plot gives a greater understanding about the movement of the arm and the range of motion it can have.




```
% Plot 10 arm configurations
figure
title({'ID: 10555972', 'Arm Configuration'});
xlabel('x[m]');
ylabel('y[m]');
for i = 1:10
    hold on
    % Plotting from elbow to end of arm
    plot([P1(1,i)  P2(1,i)],[P1(2,i)  P2(2,i)],  'b-o',  'MarkerSize',  5,
'MarkerFaceColor', 'green');
    % Plotting from origin to elbow
    plot([P1(1,i)   baseOrigin(1)],   [P1(2,i)   baseOrigin(2)],   'r-o',
'MarkerSize', 5);
end
```

2. IMPLEMENT A 2-LAYER NETWORK

The next step is to build a multi-layer neural network which will be used to learn and calculate the robot arm's inverse kinematics. To do this I will build a network that has 2 inputs (plus a third for the bias), a layer of hidden nodes (the diagram shows 2 but this can be n number of nodes) and two outputs. The network is fully connected by weight matrices on both the first and the second layer and will be passed through a sigmoid function in the hidden layer.



The input data will be the arm endpoints that have been calculated from the forward kinematics function and the output will be the inverse of the kinematics, so in this case it will be the randomly generated dataset between 0 and π. I have chosen to have two outputs for this network instead of one as it will allow me to output both the x and the y coordinates at the same time instead of having to have two separate networks and pass through each one.

2.1. IMPLEMENT THE NETWORK FEEDFORWARD PASS

To start I created a feed forward function which takes as parameters the input data and the weight matrices for the network. This function completes a one whole pass of the network to calculate the output which is then returned by the function. I have also created a function which calculates and returns the sigmoid activation of any given input. A sigmoid function is useful because it is non-linear and reduces the range between 0 and 1 whilst keeping continuous values.

```
% This function is used to carry out a feedforward pass of the network
```

```
% given its input data and both weight matrices.
function output = FeedForward(input, W1, W2)
    % Add bias to input matrix
    input = [input; 1];
    % Calculate output from hidden layer
    net = W1*input;
    % Sigmoid activation function
    a2 = SigmoidFunction(net);
    % Adding bias to activation from hidden layer
    a2hat = [a2; 1];
    % Calculating output from output layer
    output = W2*a2hat;
end

% Function to carry out the sigmoid acivation calculation
function result = SigmoidFunction(net)
    result = 1 ./ (1+(exp(-net)));
end
```

2.2. IMPLEMENT 2-LAYER NETWORK TRAINING

To train a network we must first calculate the error of the output, this is done by finding the difference between the target output and the actual output. Neural networks are a type of supervised learning, so the system requires the target data to be able to calculate the error. To do this for a 2-layer network backpropagation should be used to ensure that the entire network is updated as the second layer takes the first layer's weights as inputs, so adjusting just the first layer would have a knock-on effect to the second layer. To adjust the weights from the first layer we can then use the delta term calculated from the second layer to be the error term for the first layer.

The function below runs through a feedforward pass and then backpropagates to adjust the weights accordingly. The function takes the input data, target data and both weight matrices, and then returns the updated weight matrices. Delta 3 is equal to the error between the input data and the target data. Delta 2 is equal to the error back propagated from the higher level (the weight matrix has its bias removed before this calculation), multiplied by a scaler due to the sigmoid function in the hidden layer.

The error gradient for both weight matrices is then calculated by using the delta for that layer multiplied by the input of that layer. The weights are then updated and returned by taking away the gradient multiplied by the learning rate.

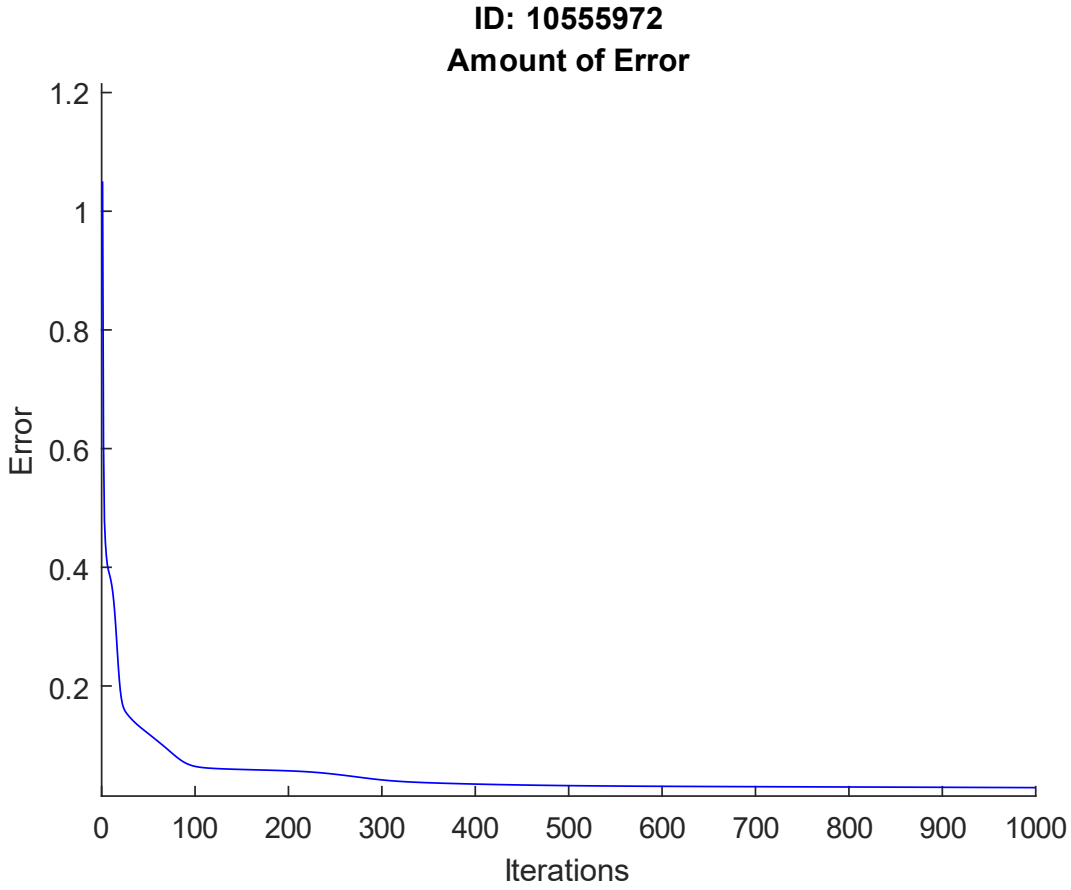
```
% Function to train the network given input data, target data and the
weight matrix. By calculating the error gradient and updating the weight
values.
function [W1, W2] = Train(input, target, W1, W2)
    % Setting learning rate
    learningRate = 0.01;
```

```
    % FEEDFORWARD PASS
    % Calculate output from hidden layer and add bias
    input = [input; 1];
    net = W1*input;
    % Sigmoid activation function
    a2 = SigmoidFunction(net);
    % Adding bias to activation from hidden layer
    a2hat = [a2; 1];
    % Calculating output from output layer
    o = W2*a2hat;

    % BACKPROPAGATION
    % Delta 3 is equal to the output error
    delta3 = -(target-o);

    % Removing bias from weights
    for i = 1:size(W2,2)-1
        W2Hat(1,i) = W2(1,i);
        W2Hat(2,i) = W2(2,i);
    end

    % Delta 2 is equal to the error from the second layer multiplied by a
    scaling factor due to the sigmoid function
    delta2 = (W2Hat'*delta3).*a2.*(1-a2);

    % Calculating the error gradient
    errGradientW1 = delta2*input';
    errGradientW2 = delta3*a2hat';

    % Updating weights using the learning rate and error gradient
    W1 = W1 - learningRate*errGradientW1;
    W2 = W2 - learningRate*errGradientW2;
end
```

2.3. TRAIN NETWORK INVERSE KINEMATICS

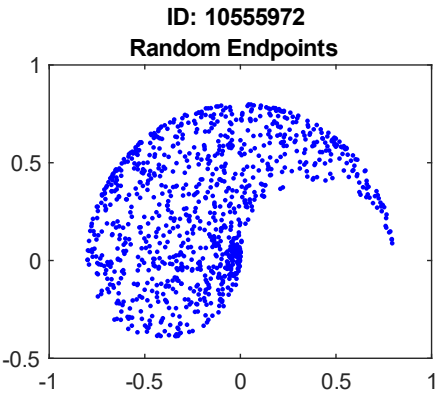
It is possible to calculate the inverse kinematics for a robot arm with only 2 degrees of freedom. However, if this number was greater, then the calculation would become much more difficult. So, in this case we want the network to learn inverse kinematics for us. To do this, we will feed in the arm end points as the input data and some randomly generated angles as the target data. Due to my network having 2 outputs we will only have to train one network.

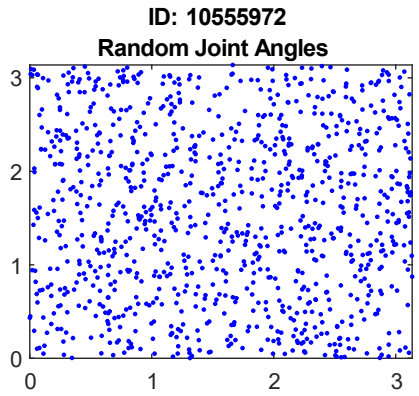
Below I have plotted the error of the neural network as the training happens. To calculate the error, I used the following formula: $(t-o)^2$. I calculated the error for each data point and then took the mean for each iteration. I ran the training for 1000 iterations and the error tends towards 0.

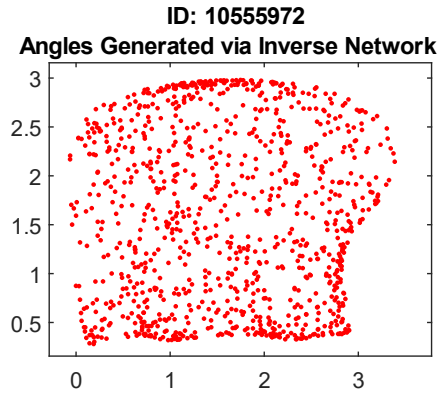


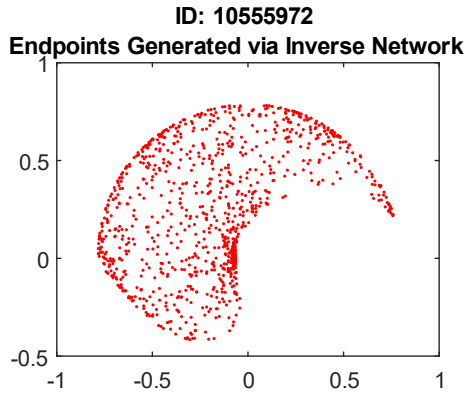
2.4. TEST AND INTERPRET INVERSE MODEL

After the network has been trained, I can carry out feedforward passes on arm end points to get the angles. I have generated a new random dataset and used the Forward Kinematics function to get the endpoints. I have then completed the feed forward pass on this data because it is different data from the data that the network was trained on. As you can see below the results are within the correct range and the output is fairly accurate. For these results I have used 1000 iterations, 10 hidden nodes and a learning rate of 0.01 for training. These values have been chosen to ensure that overfitting does not occur when training my network. Overfitting is when the training error becomes so low on the data that it is being trained on that the error on new data being passed through the network is very large. I have found that these values provide the best results while in turn not taking a considerably long time to train the network.









To better improve the accuracy of a neural network it is sometimes feasible to randomize the order of data when training the network this is so that there is no correlation between data and that the order of the data has no effect on the output of the network. Therefore, I tried randomizing the training data as shown below and this had little to no effect on the training of the network. This could be because the data set it relatively small and because it is being trained on a random set of data.

```
% Matrix from 1 to samples in random order
r1 = randperm(samples);
% Training the data for the number of iterations for each data point
for i = 1:iterations
     for j = 1:samples
          [W1, W2, err(j)] = Train(P2(:,r1(j)), randAngles(:,r1(j)),W1,W2);
     end
end
```

It is possible to normalize the both the training and input data before running it through the network. This is achieved by taking away the mean of the data and dividing it by the standard deviation. This allows for a much more fixed range of numbers and could potentially lead to faster training times. However, I have

chosen not to do this for my data because I am getting good results without it and I am already using a sigmoid activation function within my network.

Due to this network being used to produce a revolute arm to guide its way through a maze, a much more appropriate dataset to train the network on would be to train it specifically between the boundaries of the maze. This is because that the rest of the training data is wasted as the data outside of these boundaries would never be used. In addition, this could potentially allow the network to reduce its training error a lot quicker, in turn reducing training times. Data points in extrinsic space are clustering together and are not uniformly distributed. This is due to the arm only having two degrees of freedom which means that when the arm is either at its maximum or minimum reach movement is limited. To make the data more uniformly distributed an additional degree of freedom could be added to the arm.

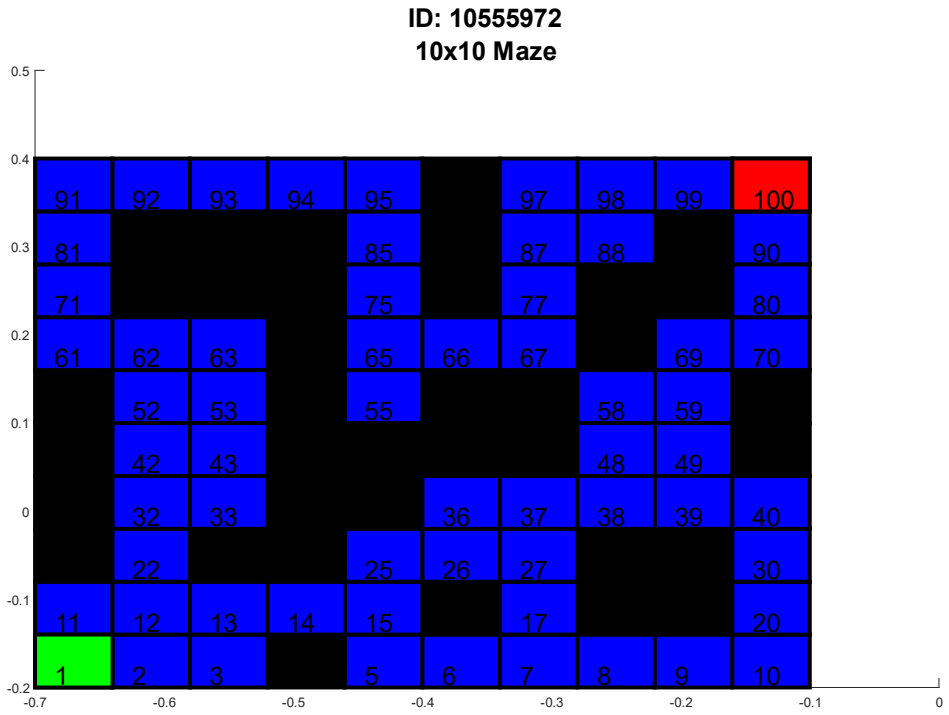
Below is the code used to train the network, complete a feedforward pass on a set of data and then plot the output as well as the original data.

```
function [W1, W2] = Network()
    % Defining variables
    armLength = [0.4;0.4]; baseOrigin = [0, 0];
    samples = 1000; iterations = 1000;
    noOfInputs = 2; noOfHiddenNodes = 10; noOfOutputNodes = 2;

    % Generating 2 x samples data between 0 and pi
    randAngles = pi * rand(2,samples);
    % Calculating arm end points given angles
    [P1,    P2]    =    RevoluteForwardKinematics2D(armLength,    randAngles,
baseOrigin);

    % Initialising random weights, plus 1 used for the bias
    W1 = rand(noOfHiddenNodes, noOfInputs + 1);
    W2 = rand(noOfOutputNodes, noOfHiddenNodes + 1);

    % Training the data for the number of iterations for each data point
    for i = 1:iterations
        for j = 1:samples
            [W1, W2, err(j)] = Train(P2(:,j), randAngles(:,j), W1, W2);
        end
    end

    % Generating new data to feed forward pass through the network
    randAngles2 = pi * rand(2,samples);
    [P1,  endPoints]  =  RevoluteForwardKinematics2D(armLength,  randAngles2,
baseOrigin);

    % Passing data through network
    for i = 1:samples
        outputtedAngles(:,i) = FeedForward(endPoints(:,i), W1, W2);
    end

    % Using the output angles and getting the arm end points
```

```
    [P3,  P4]  =  RevoluteForwardKinematics2D(armLength,  outputtedAngles,
baseOrigin);

    % Plot the random angles and endpoints. Then plot the generated
    % inverse angles and end points from the network.
    figure
    hold on
    tiledlayout(2,2)
    nexttile
    plot(endPoints(1,:), endPoints(2,:), 'b.');
    title({'ID: 10555972', 'Random Endpoints'});
    nexttile
    plot(randAngles2(1,:), randAngles2(2,:), 'b.');
    title({'ID: 10555972', 'Random Joint Angles'});
    nexttile
    plot(outputtedAngles(1,:), outputtedAngles(2,:), 'r.');
    title({'ID: 10555972', 'Angles Generated via Inverse Network'});
    nexttile
    plot(P4(1,:), P4(2,:), 'r.', 'markersize',4);
    title({'ID: 10555972', 'Endpoints Generated via Inverse Network'});
end
```

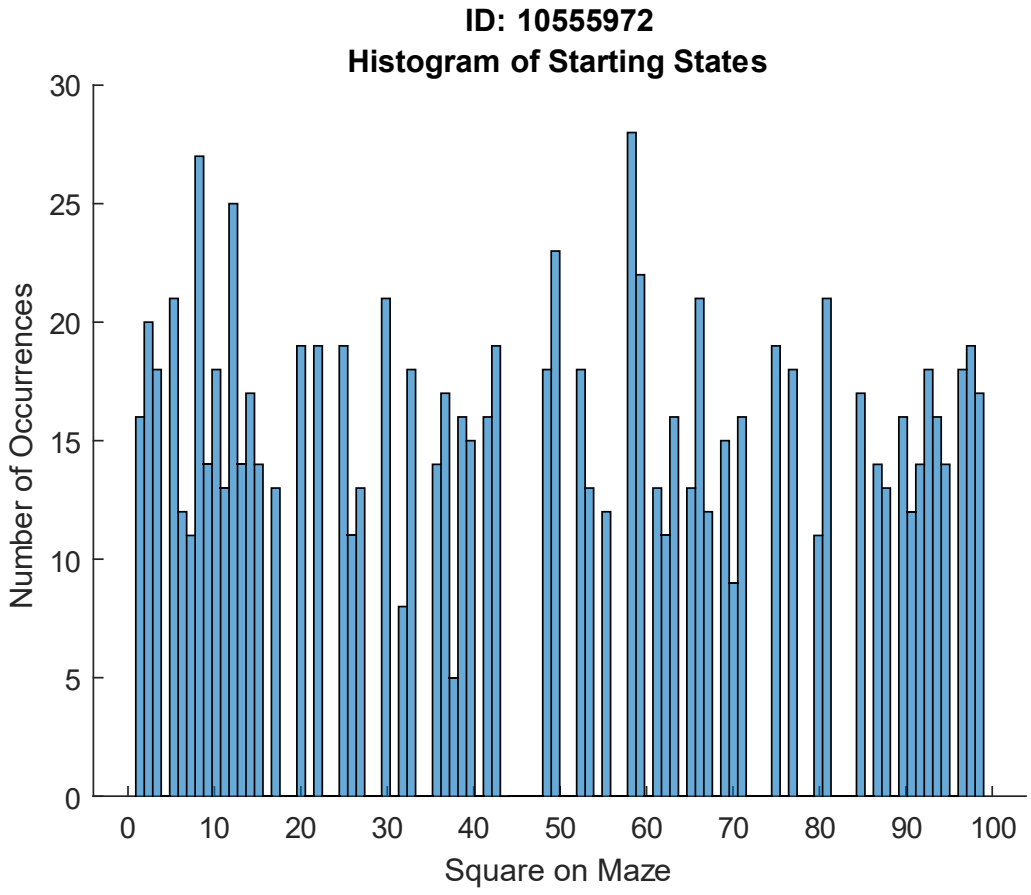
3. PATH THROUGH A MAZE USING Q-LEARNING

The next part is to implements a Q-Learning algorithm to generate a path through a given maze. This path will then be used to generate the angles for the revolute robot arm to move it along the path. Below is a 10x10 maze with various squares that are blocked, and 1 being the start state and 100 being the goal state.



3.1. GENERATE RANDOM START STATE

To begin with I have implemented a function that randomly generates a starting state within the maze. This starting state must not be a blocked or the goal state of the maze. This is shown as a histogram below when generating 1000 random starting states. It is clear here that there are 0 occurrences for the blocked and goal states.




```
% Function to compute a random starting state between 0 and 100 not
% including blocked states
function startingState = RandomStartingState(f)
     % Initial values
     allowed = false;
     b = 100;

     while (allowed == false)
        % Getting a random starting state
        startingState = ceil(b*rand);
        % Checking if starting state is in the blockedLocations array
        for i = 1:size(f.blockedLocations)
        % Getting tile number
        sidx=f.stateNumber(f.blockedLocations(i,    1),f.blockedLocations(i,
2));
        % Changing bool value depending on if state is allowed or not
```

```
        if (sidx == startingState)
          allowed = false;
          break;
        else
          allowed = true;
        end
      end
    end
end
```

3.2. BUILD A REWARD FUNCTION

Below is a reward function that rewards the algorithm if the correct action is taken when in a certain state to then reach the goal state.

```
% Reward function that takes a stateID and an action
function reward = RewardFunction(f, stateID, action)
      if ((stateID == 90 && action == 1) || (stateID == 99 && action == 4))
        reward = 10;
      else
        reward = 0;
      end
end
```

3.3. GENERATE A TRANSITION MATRIX

A transition matrix is a matrix that defines a new state given a current state and an action. Below is a function I have implements to generate a transition matrix for me given the 4 possible actions that can be taken.

```
% Function to build the transition matrix
function f = BuildTransitionMatrix(f)
      % Defining actions
      north = 1; east = 2; south = 3; west = 4;

      % Loop through each state
      for i = 1:f.xStateCnt
        for j = 1:f.yStateCnt
          % Check if state is open
          if (f.stateOpen(i, j))
            sidx=f.stateNumber(i,j);
            % Loop through each possible action
              for k = 1:4
                if (k == north)
                % Increase or decrease coordinates depending on action
                  if (i > 0 && i <= 10 && j > 0 && j <= 9)
                    if (f.stateOpen(i, j + 1))
                      f.tm(sidx, k) = f.stateNumber(i,j + 1);
                    else
                      f.tm(sidx, k) = f.stateNumber(i,j);
                    end
                  else
                    f.tm(sidx, k) = f.stateNumber(i,j);
```

```
                    end
                elseif(k == east)
                    if (i > 1 && i <= 10 && j > 0 && j <= 10)
                      if (f.stateOpen(i - 1, j))
                        f.tm(sidx, k) = f.stateNumber(i - 1,j);
                      else
                        f.tm(sidx, k) = f.stateNumber(i,j);
                      end
                    else
                      f.tm(sidx, k) = f.stateNumber(i,j);
                    end
                  elseif(k == south)
                    if (i > 0 && i <= 10 && j > 1 && j <= 10)
                      if (f.stateOpen(i, j - 1))
                        f.tm(sidx, k) = f.stateNumber(i,j - 1);
                       else
                        f.tm(sidx, k) = f.stateNumber(i,j);
                      end
                     else
                        f.tm(sidx, k) = f.stateNumber(i,j);
                     end
                   elseif(k == west)
                      if (i > 0 && i <= 9 && j > 0 && j <= 10)
                        if (f.stateOpen(i + 1, j))
                          f.tm(sidx, k) = f.stateNumber(i + 1, j);
                         else
                          f.tm(sidx, k) = f.stateNumber(i,j);
                         end
                       else
                         f.tm(sidx, k) = f.stateNumber(i,j);
                        end
                    end
                end
            end
        end
    end
```

3.4. INITIALIZE Q-VALUES

The next step is to generate a table for the Q-Values to be stored. This is a 100x4 size matrix as there is 100 states all with a possible 4 actions, all the values are randomised between 0.01 and 0.1. These Q-Values will be modified throughout the experiment depending on the discount and learning rate. These values will determine what route the algorithm will take through the maze.

```
% Intialise the Q-Table
function f = InitQTable(f, minVal, maxVal)
   f.QValues=(maxVal-minVal).*rand(f.totalStateCnt, f.actionCnt) + minVal;
end
```

3.5. IMPLEMENT Q-LEARNING ALGORITHM

To implement the Q-Learning algorithm I have set up the experiment to run for 100 trials, each with 1000 episodes. Each episode loops until the termination state is reached, the number of steps taken in an episode is recorded as this shows how well the algorithm is performing.

During each episode the next action is chosen by using a greedy action selection. This happens by getting the maximum value from the Q-Values for that state which corresponds to an action 90% of the time, the remaining 10% of the time it will randomly pick an action. This is known as the exploration part of the algorithm. After an action is determined, the next state can be found using the transition matrix as well as the reward by using the reward function previously defined.

Now by using the Q-Learning equation we can update a value in the Q-Table:

$$Q(S,A) = Q(S,A) + \alpha\,[R + \gamma maxQ(S',a) - Q(S,A)]$$

Where:
α = Learning Rate
R = Reward
γ = Discount Rate
$Q(S',a)$ = Next state and action
$Q(S,A)$ = Current state and action

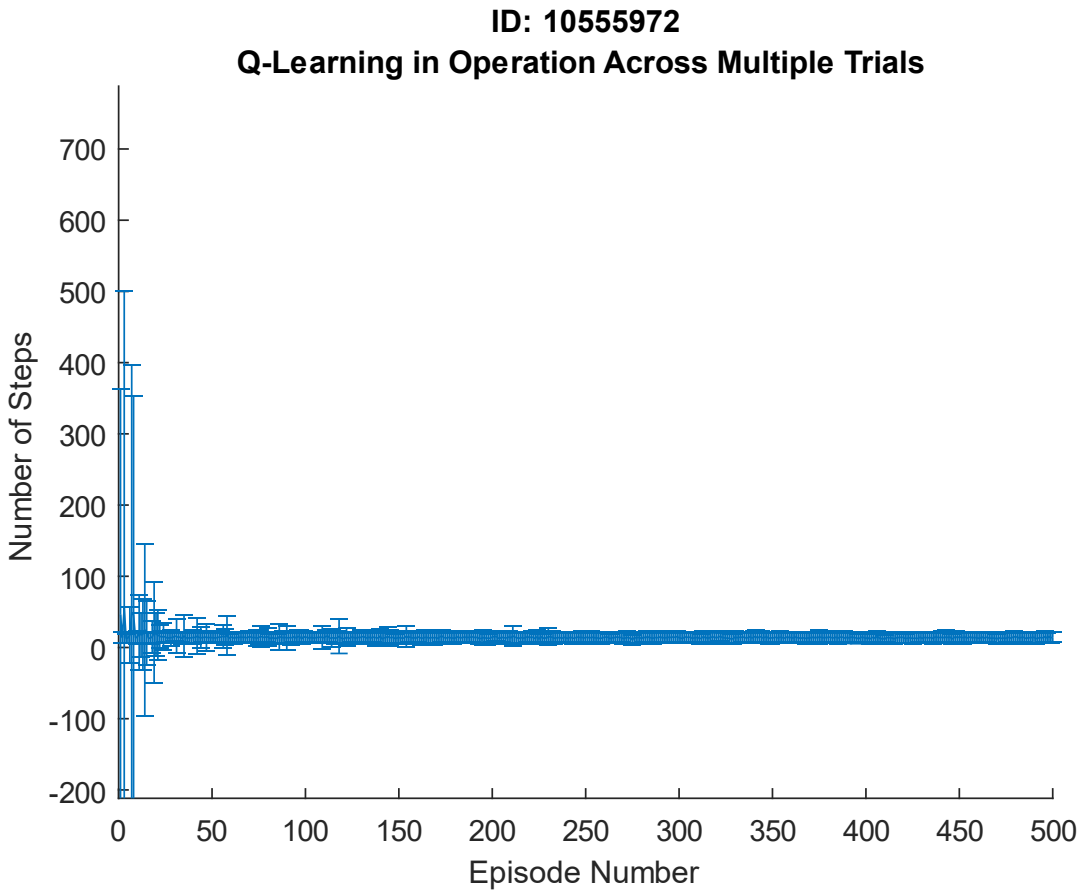
```
% Calculates mean and std and runs the number of trials
function     [meanVal,     stdVal,     stepsAcrossTrials,     coordinates]
Experiment(maze, episodes, trials)
    % Loops through number of trials
    for i = 1:trials
        [maze, stepsAcrossTrials(i, :)] = Trial(maze, episodes);
    end
    meanVal = mean(stepsAcrossTrials);
    stdVal = std(stepsAcrossTrials);
    % Gets coordinates for the final optimal route through maze
    coordinates = GetCoordinates(maze);
end

% Defines termination state and runs through the number of episodes
function [maze, steps] = Trial(maze, episodes)
    terminationState = 100;
    for i = 1:episodes
        [maze, steps(i)] = Episode(maze, terminationState);
    end
end

% Implementation of a Q-Learning episode
function [maze, steps] = Episode(maze, terminationState)
    % Defining initial values
    running = 1;
    steps = 0;
```

```
    state = maze.RandomStartingState();

    % Loops until termination state is reached
    while (running == 1)
        % Getting action, the next state, and the reward
        action = GreedyActionSelection(maze, state);
        nextState = maze.tm(state, action);
        reward = maze.RewardFunction(state, action);

        % Gets updated maze object, including the updates QValues
        maze = UpdateQ(maze, state, action, nextState, reward);

        % Termination condition
        if(nextState == terminationState)
            running = 0;
        end

        % Updating no. of steps and the current state
        steps = steps + 1;
        state = nextState;
    end
end

% Using Q-Algorithm to update a value in the QValues using the learning
and discount rate
function maze = UpdateQ(maze, state, action, resultingState, reward)
    a = 0.2;
    y = 0.9;
    maze.QValues(state, action) = maze.QValues(state, action) + a *
(reward + y * max(maze.QValues(resultingState, :)) - maze.QValues(state,
action));
end

% Selects the maximum value at any given state from QValues and returns
the action 90% of the time. 10% of the time it will "explore" and return a
random value
function action = GreedyActionSelection(maze, state)
    % Calculating random probability
    p = rand(1);

    if (p > 0.9)
        % Return random action if greater than 0.9 (10%)
        a = 0;
        b = 4;
        action = ceil((b-a) * rand + a);
    else
        % Return the index of the maximum value of the current state
        [temp, action] = max(maze.QValues(state, :));
    end
end
```

3.6. RUN Q-LEARNING

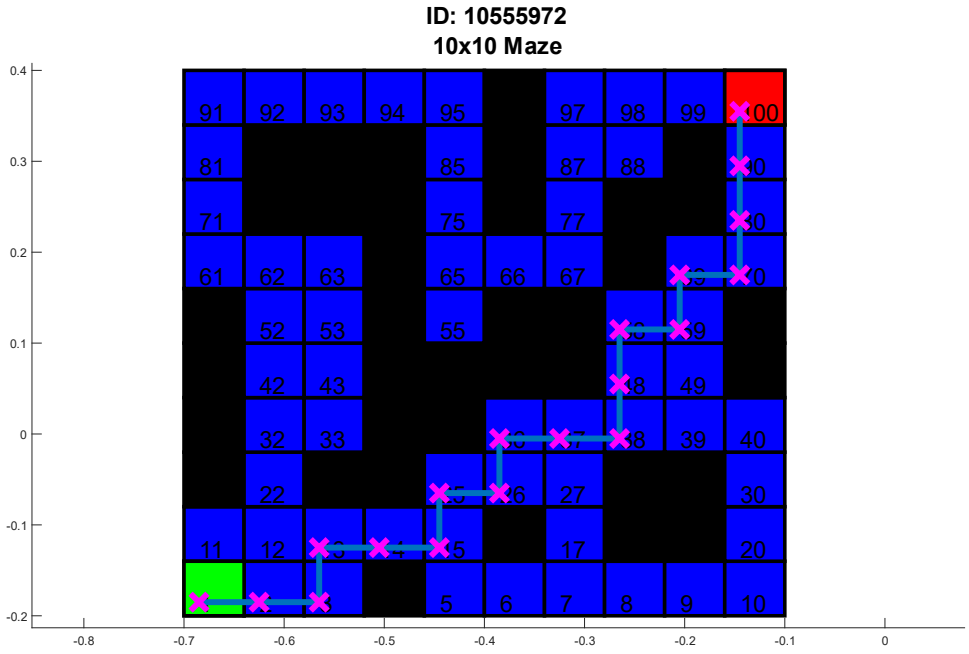
When running the Q-Learning algorithm I have used the values 0.9 for my discount rate and 0.2 as my learning rate. For each episode I have stored the number of

steps taken to reach the goal state and plotted an error bar graph using the mean and standard deviation for each episode.



3.7. EXPLOITATION OF Q-VALUES

After the entire experiment has completed, I can use the Q-Values and make a greedy action selection without exploration to find the most optimal route through the maze. I do this by making the starting state 1 and taking the maximum value in the table for that state to get the next action. This action is then used to obtain the next state. This continues to happen until the termination state is reached. I store all the coordinates throughout this process to be able to print onto the maze as shown below.




```
% Returns the coordinates of the most optimal route through the maze
function coordinates = GetCoordinates(maze)
     % Setting intial values
     termination = false;
     i = 1;
     states(i) = 1;

     while (termination == false)
         % Calculate action from QValues depending what state
        [temp, action] = max(maze.QValues(states(i), :));
        % Update the state given the action
        states(i + 1) = maze.tm(states(i), action);

        % Termination condition
        if(states(i + 1) == 100)
            termination = true;
        end
        % Get the coordinates of the current state
        coordinates(1, i) = maze.stateX(states(i));
        coordinates(2, i) = maze.stateY(states(i));
        i = i + 1;
     end
     % Get final coordinates
     coordinates(1, i) = maze.stateX(states(i));
     coordinates(2, i) = maze.stateY(states(i));
end
```

4. MOVE ARM ENDPOINT THROUGH MAZE

The final part is to use the maze path and plot the endpoints of the arm to follow through the maze. This will be achieved by taking the coordinates that have been stored by calculating the most optimal route through the maze and passing them

through the trained neural network to get the revolute arm angles to be able to plot the arm onto the maze.

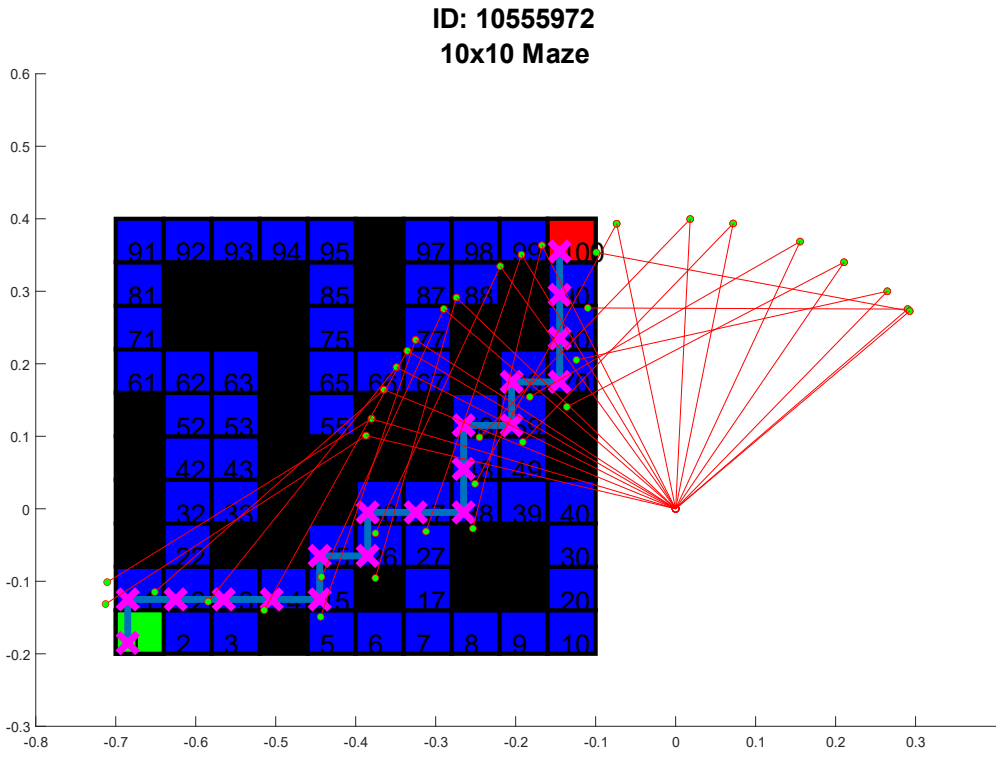
4.1. GENERATE KINEMATIC CONTROL TO REVOLUTE ARM

To plot the robot arm onto the maze, first I must take the scaled coordinates and completed a feedforward pass on them to get the angles of the arm. I then take those angles and run them through the forward kinematics function to get the arm endpoints. I then plot the maze with the calculated route from the Q-Learning algorithm and then plot the arm endpoints along with the connecting lines on top of the maze. I have scaled the maze to ensure that the entire arm will fit into the workspace without distorting the maze.

```
% Passing scaled coordinated through network to get arm angles
for i = 1:size(scaledCoordinates,2)
    outputtedAngles(:,i)     =     FeedForward([scaledCoordinates(1,i)     ;
scaledCoordinates(2,i)], W1, W2);
end

% Defining variables and passing outputted arm angles to get endpoints
armLength = [0.4;0.4]; baseOrigin = [0, 0];
[P1,    P2]    =    RevoluteForwardKinematics2D(armLength,    outputtedAngles,
baseOrigin);

% Drawing the maze and plotting the route to termination state
maze.DrawMaze();
title({'ID: 10555972', '10x10 Maze'});
xlim([-0.8 0.4]);
ylim([-0.3 0.6]);
line(scaledCoordinates(1,:),    scaledCoordinates(2,:),    'Marker',    'x',
'MarkerEdgeColor',  'm','MarkerFaceColor',  [1,  0,  1],  'MarkerSize',  20,
'LineWidth', 5);

% Plotting points with arm, connecting to origin
for i = 1:19
  plot([P1(1,i)   P2(1,i)],[P1(2,i)   P2(2,i)],   'r-o',   'MarkerSize',   5,
'MarkerFaceColor', 'green');
  plot([P1(1,i)    baseOrigin(1)],    [P1(2,i)    baseOrigin(2)],    'r-o',
'MarkerSize', 5);
end
```

4.2. ANIMATED REVOLUTE ARM MOVEMENT

Below is a screenshot of the arm endpoints from the network plotted on top of the maze. I have also included a link to an animation of the arm moving through the maze.



https://www.youtube.com/watch?v=bdKctw_1Dq0