1. TRAINING DATA GENERATION

In this section we are going to generate sets of data to test and train our Neural Network in Part 2. We are also going to create / plot a revolute arm and test it with random values.

To generate the data sets we are going to first generate 1000 uniformly distributed random value samples, between 0 and Pi, for 2 joints. Then put them through the two forward kinematics equations provided to get the end-point positions of both links. This allows us to plot and observe all the reachable positions of the revolute arm within a given workspace, which then helps us to decide where in that workspace we can fit our maze (generated in Part 3) so that the arm is able to reach and move around the maze.

1.1. Display workspace of revolute arm

- Generate 1000 uniformly distributed set of training angle data points in the array theta over the range 0 to Pi radians using the Matlab rand command.
- Also generate a testing dataset of 1000 samples.

Call to functions

```
% Number of samples
numOfSamples = 1000;
% Training data set
trainingThetaSet = fk.GenThetaSet(numOfSamples);
% Testing data set
testingThetaSet = fk.GenThetaSet(numOfSamples);
```

Function to generate uniformly distributed data sets

```
function theta = GenThetaSet(obj,samples)
    %  Matlab's rand function guide page
    %  Generate N random numbers in the interval (a,b)
    %  r = a + (b-a).*rand(N,1)
    theta = pi .* rand(2,samples);
end
```

- Run the RevoluteForwardKinematics2D function with the specified parameters to generate the corresponding endpoint locations.

```
% Training data set
trainingThetaSet = fk.GenThetaSet(numOfSamples);
% Declare lengths of the two links
armLen = [0.4,0.4];
% Declare origin of the arm
origin = [0,0];
% Perform forward kinematics to get end-point positions of both links
[Pos1,Pos2] = RevoluteForwardKinematics2D(armLen, trainingThetaSet, origin);
```

- Plot the endpoint positions and the arm origin position too.

Call to function

```
% Plot all end-point positions for link 2 (end-effector positions)
fk.plotWorkspace("Workspace of Revolute Arm",Pos2(1,:),Pos2(2,:),origin);
```

Function to plot the workspace of the revolute arm

```
function plotWorkspace(obj,name,ENDx,ENDy,origin)
    figure
    hold on
    plot(ENDx,ENDy,".r");
    plot(origin(1),origin(2),"Ok", 'MarkerFaceColor',[0,0,0],...
        'MarkerSize',10);

    title(strcat("10538828: ",name));
    xlabel("X-axis");
    ylabel("Y-axis");
    legend("End-Effector Pos", "Origin");
    hold off
end
```

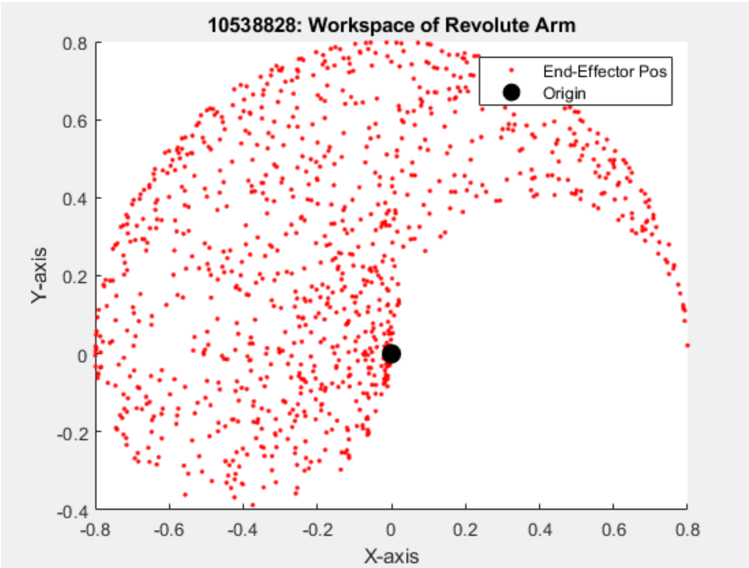


Figure 1 - Two Link Revolute Arm Workspace

- What can you say about the useful range of this arm?

From Figure 1 we can observe that a reasonable place to fit the maze in Part 3 could be the rectangular area defined by X = [-0.6,-0.2] and Y = [-0.2,0.2].

- Include your plot and commented Matlab solution code embedded in the report document.

Main

```
% Create object of class ForwardKinematics
fk = ForwardKinematics;
% Number of samples
numOfSamples = 1000;
% Testing data set
testingThetaSet = fk.GenThetaSet(numOfSamples);
% Training data set
trainingThetaSet = fk.GenThetaSet(numOfSamples);
% Declare lengths of the two links
armLen = [0.4,0.4];
% Declare origin of the arm
origin = [0,0];
% Perform forward kinematics to get end-point positions of both links
[Pos1,Pos2] = RevoluteForwardKinematics2D(armLen, trainingThetaSet, origin);
% Plot all end-point positions for link 2 (end-effector positions)
fk.plotWorkspace("Workspace of Revolute Arm",Pos2(1,:),Pos2(2,:),origin);
```

Function to generate uniformly distributed data sets

```
function theta = GenThetaSet(obj,samples)
    %  Matlab's rand function guide page
    %  Generate N random numbers in the interval (a,b)
    %  r = a + (b-a).*rand(N,1)
    theta = pi .* rand(2,samples);
end
```

Function to plot the workspace of the revolute arm

```
function plotWorkspace(obj,name,ENDx,ENDy,origin)
    figure
    hold on
    plot(ENDx,ENDy,".r");
    plot(origin(1),origin(2),"Ok", 'MarkerFaceColor',[0,0,0],...
        'MarkerSize',10);

    title(strcat("10538828: ",name));
    xlabel("X-axis");
    ylabel("Y-axis");
    legend("End-Effector Pos", "Origin");
    hold off
end
```

Forward Kinematics Function Provided

```
function [P1,P2] = RevoluteForwardKinematics2D(armLen, theta, origin)
% calculate revolute arm forward kinematics
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% all rights reserved
% Author: Dr. Ian Howard
% Associate Professor (Senior Lecturer) in Computational Neuroscience
% Centre for Robotics and Neural Systems
% Plymouth University
% A324 Portland Square
% PL4 8AA
% Plymouth, Devon, UK
% howardlab.com
% 22/09/2018
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% unpack joint angles to make clear what we are dong here
theta1 = theta(1,:);
theta2 = theta(2,:);

% unpack segment length
l1 = armLen(1);
l2 = armLen(2);

% relative forward kinematics
x = l1 * cos(theta1) + ones(size(theta1)) * origin(1);
y = l1 * sin(theta1) + ones(size(theta1)) * origin(2);

% pack results
P1 = [x; y;];

% relative forward kinematics
x = l1 * cos(theta1) + l2 * cos(theta1 + theta2) + ones(size(theta2)) * origin(1);
y = l1 * sin(theta1) + l2 * sin(theta1 + theta2) + ones(size(theta2)) * origin(2);

% pack results
P2 = [x; y;];
end
```

1.2. Configurations of a revolute arm

- To illustrate arm configurations, keeping other parameters as before, now generate just 10 uniformly distributed set of angle data points between 0 – Pi radians using the Matlab rand command.
- Again, run the RevoluteForwardKinematics2D function with the specified parameters to generate the corresponding elbow and endpoint locations.
- Plot the elbow and endpoint locations and the arm sections between them.
- Include your plot and commented Matlab solution code embedded in the report document.

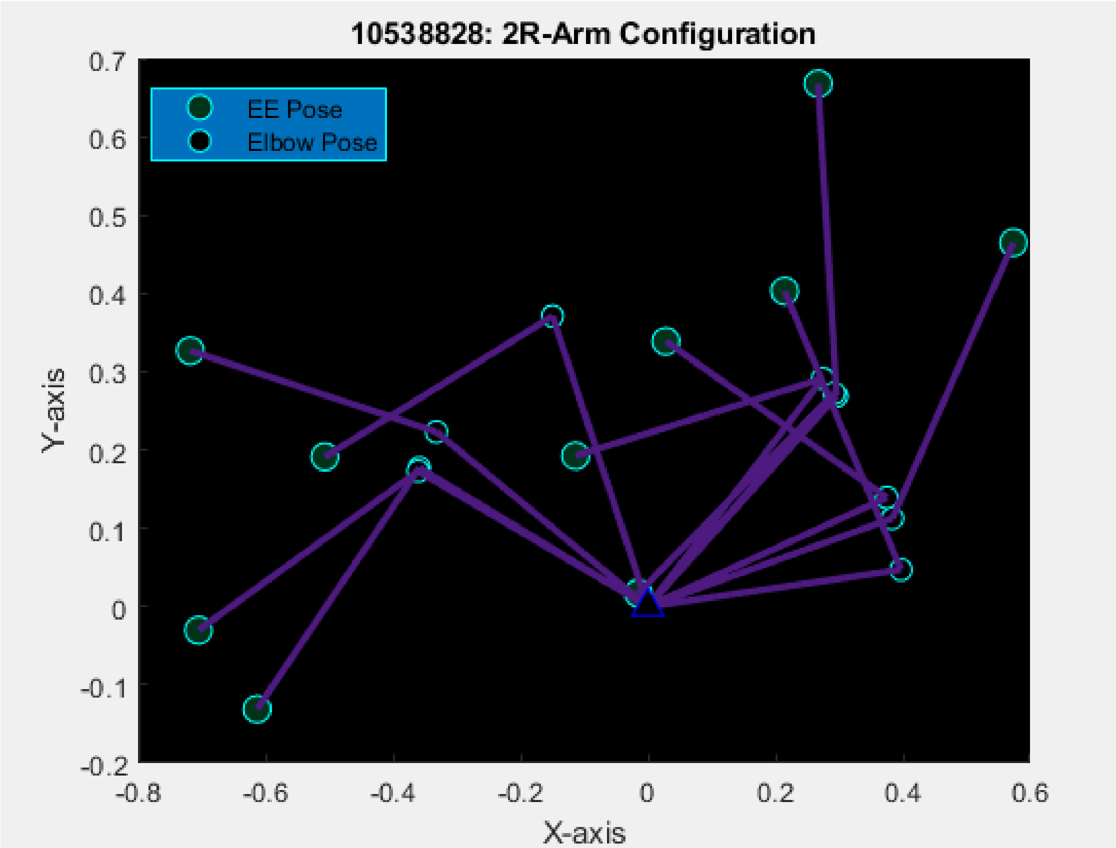


Figure 2 - Revolute Arm Random Joint Angles Test

Main

```
% Constants
numOfSamples = 10;
armLen = [0.4,0.4];
origin = [0,0];

% Generate Dataset
ThetaSet = fk.GenThetaSet(numOfSamples);
% Perform forward kinematics to get end-point positions of both links
[Pos1,Pos2] = RevoluteForwardKinematics2D(armLen, ThetaSet, origin);
% Plot all end-points and links
fk.plotArm("2R-Arm Configuration",...
           Pos1(1,:),Pos1(2,:),...
           Pos2(1,:),Pos2(2,:),...
           origin);
```

Function to plot end-effector and elbow joints and links

```
function plotArm(obj,name,x1,y1,x2,y2,origin)

    % plot origin point
    plot(origin(1),origin(2),"^c","MarkerFaceColor","k","MarkerSize",10);
    hold on
    % loop through all vector points
    for i = 1:length(x1)

        % plot end-effector points
        plot(x2(i),y2(i),"O","MarkerSize",10,'MarkerEdgeColor','c','MarkerFaceColor',[0.0, 0.2, 0.1]);
        % plot links between elbow points and end-effector points
        plot([x1(i),x2(i)],[y1(i),y2(i)],"-","color",[0.3, 0.1, 0.5],"LineWidth",2.5);

        % plot elbow points
        plot(x1(i),y1(i),"O","MarkerSize",8,'MarkerEdgeColor','c','MarkerFaceColor',"k");
        % plot links between origin and elbow points
        plot([origin(1),x1(i)],[origin(2),y1(i)],"-","color",[0.3, 0.1, 0.5],"LineWidth",2.5);

    end
    % set plot background colour to black
    set(gca,'Color','k')
    % set plot title by concatinating my student ID with title required
    title(strcat("10538828: ",name));
    % set x-axis and y-axis labels
    xlabel("X-axis");
    ylabel("Y-axis");
    % set legends
    legend("EE Pose","Elbow Pose");
    hold off
end
```

2. IMPLEMENT 2-LAYER NETWORK

In this section we are going to setup, train and then use a Multilayer Neural Network to perform inverse kinematics on a 2 link 2 revolute joint arm that will follow a path through the maze.

We will be setting up a 2 layer Neural Network with a sigmoid activation in the first layer and no activation in the second layer. The sigmoid activation squeezes all data between 0 and 1. Outputs (from sigmoid) closer to 1, tend to move towards 1 and outputs closer to 0 tend to move closer to 0.

During the training process the network is put through a loop and for every iteration the following sequence is performed:

- FeedForward Pass
- Error Gradient and Back Propagation
- Update Weights

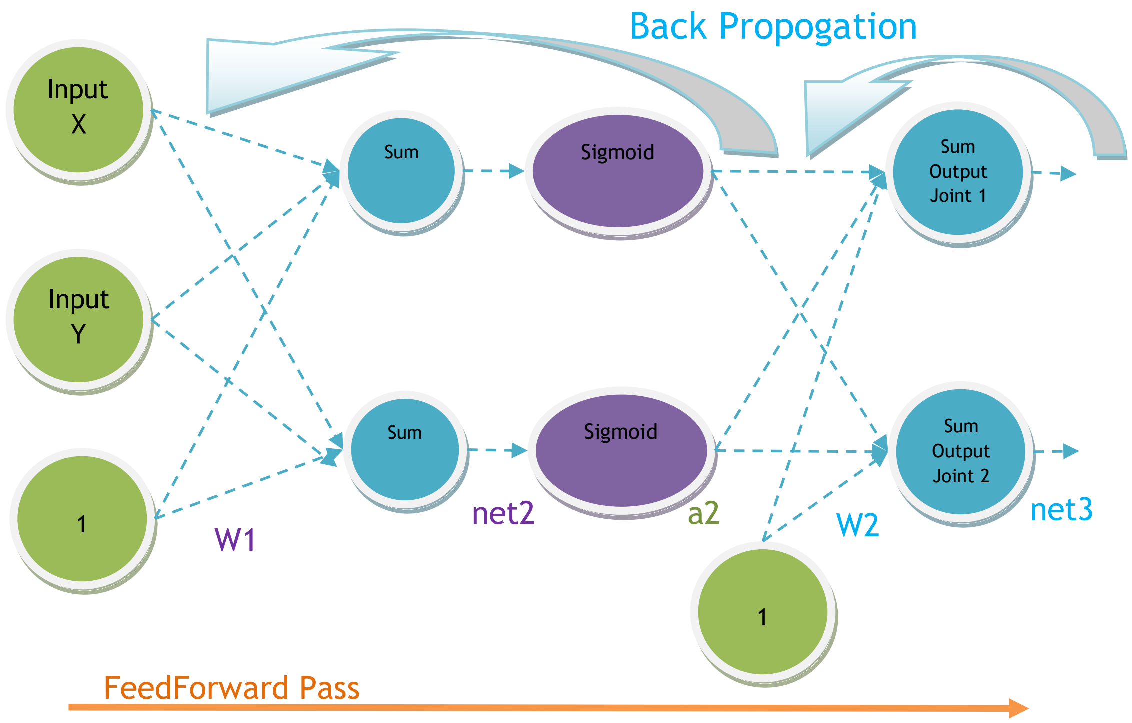


Figure 3 - 2 Layer Multilayer Neural Network

A. **FeedForward Pass**

The aim here is that after training we should have a value for weights 1 (**W1**) that we can multiply with the inputs (this gives us **net2**) then put it through the sigmoid activation (this gives us **a2**) then multiply with weights 2 (**W2**) to get target values (to get **output** / **net3**). This process is called FeedForward Pass.

The equations are as follows:

- **net2** = **W1 * X** (where X is input matrix)
- **a2** = $\frac{1}{1+e^{-\boldsymbol{net2}}}$ (sigmoid, lower layer output activation)
- **â2** = [**a2** ; 1] (augment a2 by adding a row of all 1s, due to bias)

- **net3** = **W2 * â2** (Top layer output)

B. Error Gradient and Back Propagation

This technique is used to adjust weights and biases in all the layers of the neural network. This strengthens the effects by the relevant weights and biases on the error gradient.

As we have 1 hidden layer in our neural network, we have 2 steps of back propagation. Let the back propagation from **net3** back to **â2** = **delta3** (top layer back propogation) and from **â2** back to **inputs** (**X**) = **delta2** (lower layer back propogation). As we don't have a sigmoid activation in the top layer, calculating **delta3** is a little easier.

Error gradient looks at the change in error with respect to W1 and W2 respectively, this can be written as follows:

- $\frac{\partial e}{\partial W_2}$
- $\frac{\partial e}{\partial W_1}$

Where e = target (t) – output (o). However for normal distributed data we use the Sum of Squred Error instead (SSE), which is = $(t - o)^2$

Calculate $\frac{\partial e}{\partial W_2}$:

For convenience we can add a factor of a half in front of the expression for SSE, which cancels out after subsequent differentiation. This gives us:

e = $\frac{1}{2}(t-0)^2$ (Let u = $t - 0$)

⇨ e = $\frac{1}{2}u^2$

⇨ $\frac{\partial e}{\partial W_2}$ = $\frac{\partial}{\partial W_2}\left[\frac{1}{2}u^2\right]$ (function of a function)

⇨ Using chain rule, we get: $\frac{\partial e}{\partial W_2}$ = $\frac{\partial e}{\partial u}\frac{\partial u}{\partial W_2}$

⇨ $\frac{\partial e}{\partial W_2}$ = $-(t-o)\frac{\partial o}{\partial W_2}$

⇨ Since (for a linear network) output (O) = WX (Where X is the inputs)

⇨ And we know: $\frac{\partial}{\partial W'}(W' * X) = X'$ (Where **'** represents transpose of a matrix)

⇨ $\frac{\partial o}{\partial W_2}$ = $\frac{\partial}{\partial W'_2}(W'_2 * a2) = a2'$ (Where **a2** are inputs to top layer)

⇨ Finally we get:

$$\boldsymbol{\frac{\partial e}{\partial W_2} = -(t-o)a2'}$$

Where delta3 = $\boldsymbol{-(t-o)}$

Calculate $\frac{\partial e}{\partial W_1}$:

e = $\frac{1}{2}(t-0)^2$ (Let u = $t-0$)

- ⇨ e = $\frac{1}{2}u^2$
- ⇨ $\frac{\partial e}{\partial W_1} = \frac{\partial}{\partial W_1}\left[\frac{1}{2}u^2\right]$ (function of a function)
- ⇨ Using chain rule, we get: $\frac{\partial e}{\partial W_1} = \frac{\partial e}{\partial u}\frac{\partial u}{\partial W_1}$
- ⇨ $\frac{\partial e}{\partial W_1} = \frac{\partial e}{\partial u}.-\frac{\partial O}{\partial W_1}$ (where $\frac{\partial u}{\partial W_1} = -\frac{\partial o}{\partial W_1}$)
- ⇨ $\frac{\partial e}{\partial W_1} = -\frac{\partial e}{\partial u}.\frac{\partial O}{\partial net_3}.\frac{\partial net_3}{\partial W_1}$ (net3 lyes between O and W1 $\frac{\partial O}{\partial W_1} = \frac{\partial O}{\partial net_3}.\frac{\partial net_3}{\partial W_1}$)
- ⇨ Since the output is dependent on net$_3$ or in other words O = $f(net_3)$ then the derivative of the output (O) w.r.t net$_3$ = $\frac{\partial O}{\partial net_3}$ = $f'(net_3)$
- ⇨ $\frac{\partial e}{\partial W_1} = -\frac{\partial e}{\partial u}.f'(net_3).\frac{\partial net_3}{\partial W_1}$
- ⇨ Solve $\frac{\partial net_3}{\partial W_1}$ by chain rule:
  - $\frac{\partial net_3}{\partial W_1} = \frac{\partial net_3}{\partial a_2}.\frac{\partial a_2}{\partial W_1}$
  - $\frac{\partial net_3}{\partial W_1} = \frac{\partial net_3}{\partial a_2}.\frac{\partial a_2}{\partial net_2}.\frac{\partial net_2}{\partial W_1}$ (where $\frac{\partial a_2}{\partial W_1} = \frac{\partial a_2}{\partial net_2}.\frac{\partial net_2}{\partial W_1}$)
  - $\frac{\partial net_3}{\partial W_1} = \frac{\partial net_3}{\partial a_2}.f'(net_2).\frac{\partial net_2}{\partial W_1}$ (where $\frac{\partial a_2}{\partial net_2} = f'(net_2)$)

- ⇨ Put $\frac{\partial net_3}{\partial W_1} = \frac{\partial net_3}{\partial a_2}\boldsymbol{.}\ f'(net_2).\frac{\partial net_2}{\partial W_1}$ back into the equation
- ⇨ $\frac{\partial e}{\partial W_1} = -\frac{\partial e}{\partial u}\boldsymbol{.}\ f'(net_3).\frac{\partial net_3}{\partial a_2}\boldsymbol{.}\ f'(net_2).\frac{\partial net_2}{\partial W_1}$
- ⇨ Since e = $\frac{1}{2}u^2$ where u = (t - O)
- ⇨ $\frac{\partial e}{\partial W_1} = -(t - O).f'(net_3)\ \boldsymbol{.}\frac{\partial net_3}{\partial a_2}\boldsymbol{.}\ f'(net_2).\frac{\partial net_2}{\partial W_1}$
- ⇨ Since $net_3 = W_2a_2$
- ⇨ $\frac{\partial e}{\partial W_1} = -(t - O).f'(net_3)\boldsymbol{.}W_2^T\boldsymbol{.}\ f'(net_2).\frac{\partial net_2}{\partial W_1}$
- ⇨ $net_2 = W_1X$
- ⇨ $\frac{\partial e}{\partial W_1} = -(t - O).f'(net_3)\boldsymbol{.}W_2^T\boldsymbol{.}\ f'(net_2)\boldsymbol{.}X^T$ (where X = inputs)
- ⇨ Since $f'(net_3) = O(1 - O)$ and $f'(net_2) = a_2(1 - a_2)$
- ⇨ Finally, we get:

$$\boldsymbol{\frac{\partial e}{\partial W_1} = -(t - O).O(1 - O).W_2^T.\ a_2(1 - a_2).X^T}$$

Where delta3 = $-(t - O).O(1 - O)$ (With sigmoid activation in the top layer)
or delta3 = $-(t - O)$ (Without sigmoid activation in the top layer)

Therefore **delta2 = (W_2^T . $delta_3$) . $\boldsymbol{a_2(1 - a_2)}$**

C. Update Weights

In this step, we take away the error calculated in the last part (Part B), from the weights. By subtracting the error from the weights we gradually move towards out target values. How gradually we want to move towards the target values, is defined by the Learning Rate (α).

Learning rate is generally kept very small and is multiplied by the error gradient before subtracting it from the weights. If the learning rate is too small the network takes more iterations to converge. If the learning rate is too large then the network can miss the local minima or can be overfitted and hence never converge.

The equations are as follows:

$$W_1 = W_1 - \alpha\left(\frac{\partial e}{\partial W_1}\right) \quad \text{and} \quad W_2 = W_2 - \alpha\left(\frac{\partial e}{\partial W_2}\right)$$

2.1. Implement the network feedforward pass

- Implement the 2-layer network feedforward pass (with n_h hidden units and a linear output) in Matlab.
- Given the weight matrix, this will generate the network output activation for a given input vector.

FeedForward Pass

```
%%  FeedForward Pass
net2 = W1*X;                        % Calculate net2
a2 = 1./(1+exp(-net2));             % Sigmoid Activation
a2Hat = [1./(1+exp(-net2));1;];     % Augment input for top layer
O = W2*a2Hat;                       % net3 / output
```

2.2. Implement 2-layer network training

- Implement the generalized delta rule in Matlab to train a network.

Delta Terms

```
Delta3 = -(t-O);                                 % Calculate delta3 (Top / outer layer)
W2_noBias = W2(:,1:hiddenUnits);                 % Create copy of W2 without bias terms
Delta2 = (W2_noBias'*Delta3).*(a2.*(1-a2));      % Calculate delta2 (Lower / hidden layer)
```

Error Gradients

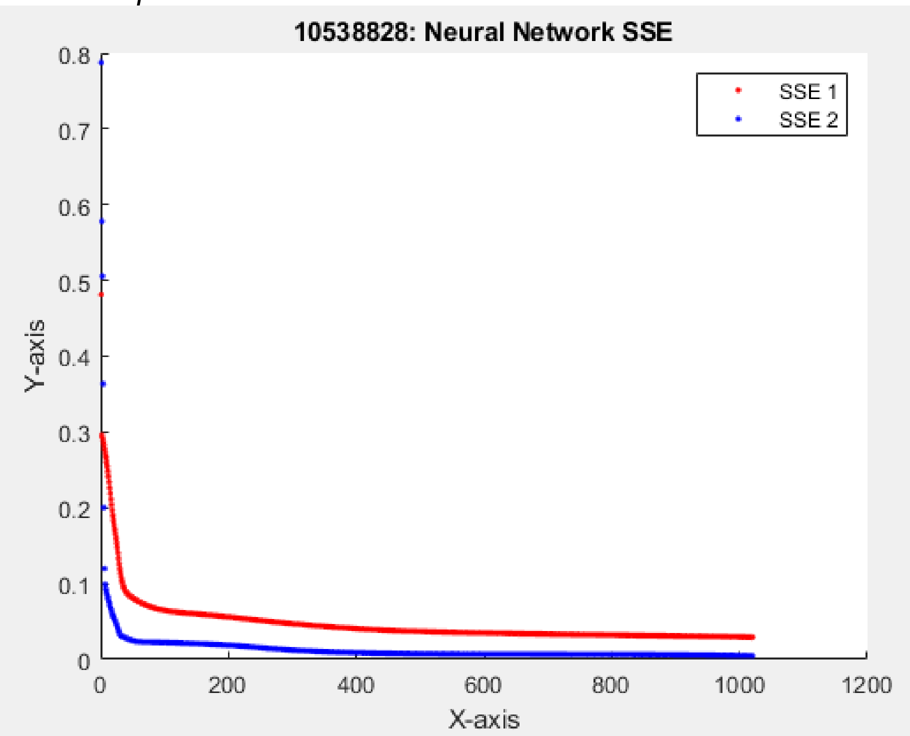
```
de_dw1 = Delta2 * X';         % Error Gradient w.r.t W1 (Lower / hidden layer)
de_dw2 = Delta3 * a2Hat';     % Error Gradient w.r.t W2 (Top / outer layer)
```

2.3. Train network inverse kinematics

- Train your linear output 2-layer network with the ***augmented*** end effect positions as input, and joint angles as outputs.
- To do so you will need two networks with a single linear output each.
- Alternatively, you can use a single network with two outputs.
- Plot the error as training proceeds.

At first I decided to use two Neural Networks with single outputs and in matrix form. However, I couldn't get the performance I was hoping for in this setup and my laptop was struggling to cope with large matricies over high number of iterations. I then decided to re-write the whole thing as a single Neural Network with two outputs and in a element by element form. This setup takes much longer to train but my laptop is able to cope with this and my results are much better.

Sum of squared errors



Train Neural Network

```
close all
clear all

% set true it Testing is required
testReqFlag = false;

% Get FK and NN classes
fk = ForwardKinematics;
NN = NeuralNetworks;

% Revolute Arm Variables
armLen = [0.4,0.4];
origin = [0,0];

% NN Training Variables
learningRate = 0.025;
hiddenUnits = 40;
nOfInputs = 3;
nOfSamples = 2000;
nOfEpisodes = 1000;

% Variables to record data during training
targetBank = [];
outputBank = [];
SSE1 = [];
SSE2 = [];

%% Generate Training Dataset
Theta1 = NN.GenThetaSet(nOfSamples); % Generate Theta1
Theta2 = NN.GenThetaSet(nOfSamples); % Generate Theta2
target = [Theta1;Theta2];            % Concatinate

%%
%___Normalize inputs and target = input - mean / standard deviation
% [Theta1,Mean.X,stDev.X] = NormalizeInputs(Theta1);
% [Theta2,Mean.Y,stDev.Y] = NormalizeInputs(Theta2);


%___Augment inputs
Theta = [Theta1;Theta2;ones(1,length(Theta1))];

% Normalize targets
% [target1,Mean.t1,stDev.t1] = NormalizeInputs(target(1,:));
% [target2,Mean.t2,stDev.t2] = NormalizeInputs(target(2,:));
% target = [target1;target2];

%___Xavier Initialisation of weights
XavierInit = sqrt(1 / (size(Theta,1) + size(target,1)));       % Xavier Initialisation
W1 = abs(randn(hiddenUnits , size(Theta,1)) .* XavierInit);     % Initiate W1
W2 = abs(randn(size(target,1) , hiddenUnits+1) .* XavierInit);  % Initiate W2

%___Initiate random weights with bias
% W1 = randn(hiddenUnits,nOfInputs);
% W2 = randn(size(target,1),hiddenUnits + 1);

[P1,P2] = RevoluteForwardKinematics2D(armLen, Theta, origin);   % Get end point coordinates from joint angles
```

```
for iterations = 1:nOfEpisodes
    for s_i = 1:nOfSamples
        X = P2(:,s_i);          % Get both inputs (all rows) and iterate through columns
        X = [X;1];              % Augment inputs

        t = target(:,s_i);      % grab each target element iteratively

        %% FeedForward Pass
        net2 = W1*X;                    % Calculate net2
        a2 = 1./(1+exp(-net2));         % Sigmoid Activation
        a2Hat = [1./(1+exp(-net2));1;]; % Augment input for top layer
        O = W2*a2Hat;                   % net3 / output

        Delta3 = -(t-O);                            % Calculate delta3 (Top / outer layer)
        W2_noBias = W2(:,1:hiddenUnits);            % Create copy of W2 without bias terms
        Delta2 = (W2_noBias'*Delta3).*(a2.*(1-a2)); % Calculate delta2 (Lower / hidden layer)

        de_dw1 = Delta2 * X';       % Error Gradient w.r.t W1 (Lower / hidden layer)
        de_dw2 = Delta3 * a2Hat';   % Error Gradient w.r.t W2 (Top / outer layer)

        W1 = W1-(learningRate .* de_dw1);   % Update W1
        W2 = W2-(learningRate .* de_dw2);   % Update W2

%         outputBankBuff(:,s_i) = O;
        error = target(:,s_i) - O;              % Calculate error

        SSE1buff(s_i) = error(1) * error(1);    % Calculate SSE for output 1
        SSE2buff(s_i) = error(2) * error(2);    % Calculate SSE for output 2

    end
%     outputBankBuff1 = DeNormalizeOutputs(outputBankBuff(1,:),Mean.t1,stDev.t1);
%     outputBankBuff2 = DeNormalizeOutputs(outputBankBuff(2,:),Mean.t2,stDev.t2);
%     outputBankBuff = [outputBankBuff1;outputBankBuff2];
%     outputBank = [outputBank,outputBankBuff];
    SSE1 = [SSE1,sum(SSE1buff)/size(SSE1buff,2)];   % Store SSE 1 for one trail
    SSE2 = [SSE2,sum(SSE2buff)/size(SSE2buff,2)];   % Store SSE 2 for one trial
    disp(iterations)
end


% [Pos1,Pos2] = RevoluteForwardKinematics2D(armLen, outputBank, origin);
% fk.plotWorkspace("Output Data", Pos2(1,:), Pos2(2,:), origin);
% NN.plotSingle("Neural Network SSE Joint1", SSE1)
% NN.plotSingle("Neural Network SSE Joint2", SSE2)
NN.plotError("Neural Network SSE",SSE1,SSE2)        % plot SSE1 and SSE2
```

Test Neural Network

```
%% Test Trained Neural Network

if (testReqFlag == true)
    %___Generate Test Data
    TestTheta1 = NN.GenThetaSet(nOfSamples);
    TestTheta2 = NN.GenThetaSet(nOfSamples);
    %___Augment Test Data
    TestTheta = [Theta1;Theta2;ones(1,length(Theta1))];

    %___Perform Forward Kinematics
    [TestP1,TestP2] = RevoluteForwardKinematics2D(armLen, TestTheta, origin);
    %___Augment Test Inputs
    TestP2 = [TestP2;ones(1,size(TestP2,2))];

    %___FeedForward Pass
    net2 = W1*TestP2;                 % Calculate net2
    a2 = 1./(1+exp(-net2));           % Sigmoid Activation
    a2Hat = [a2;ones(1,size(a2,2))]; % Augment input for top layer
    o = W2*a2Hat;                     % net3 / output

    %___Perform Forward Kinematics
    [TestPos1,TestPos2] = RevoluteForwardKinematics2D(armLen, o, origin);

    %___Plot raw inputs, targets, FeedForward Pass outputs
    %___and FeedForward Pass through Forward Kinematics
    NN_plotName.P1 = "Joint angles before NN";
    NN_plotName.P2 = "Workspace before NN";
    NN_plotName.P3 = "Joint angles after NN";
    NN_plotName.P4 = "Workspace after NN";
    plotfour(NN_plotName   ,origin,...
        TestTheta(1,:),TestTheta(2,:),...
        TestP2(1,:)   ,TestP2(2,:),...
        o(1,:),o(2,:) ,...
        TestPos2(1,:) ,TestPos2(2,:))
end
```

2.4. Test and interpret inverse mode

- Test the network by running a forward pass on position data to generate predicted arm angles.
- This could be the training data but ideally should be another testing set.
- Then run the forward kinematics of predicted angles and get corresponding end points.
- Plot the joint angles and endpoints and compare with those you started with. Discuss the significance of these plots.

…Next Page

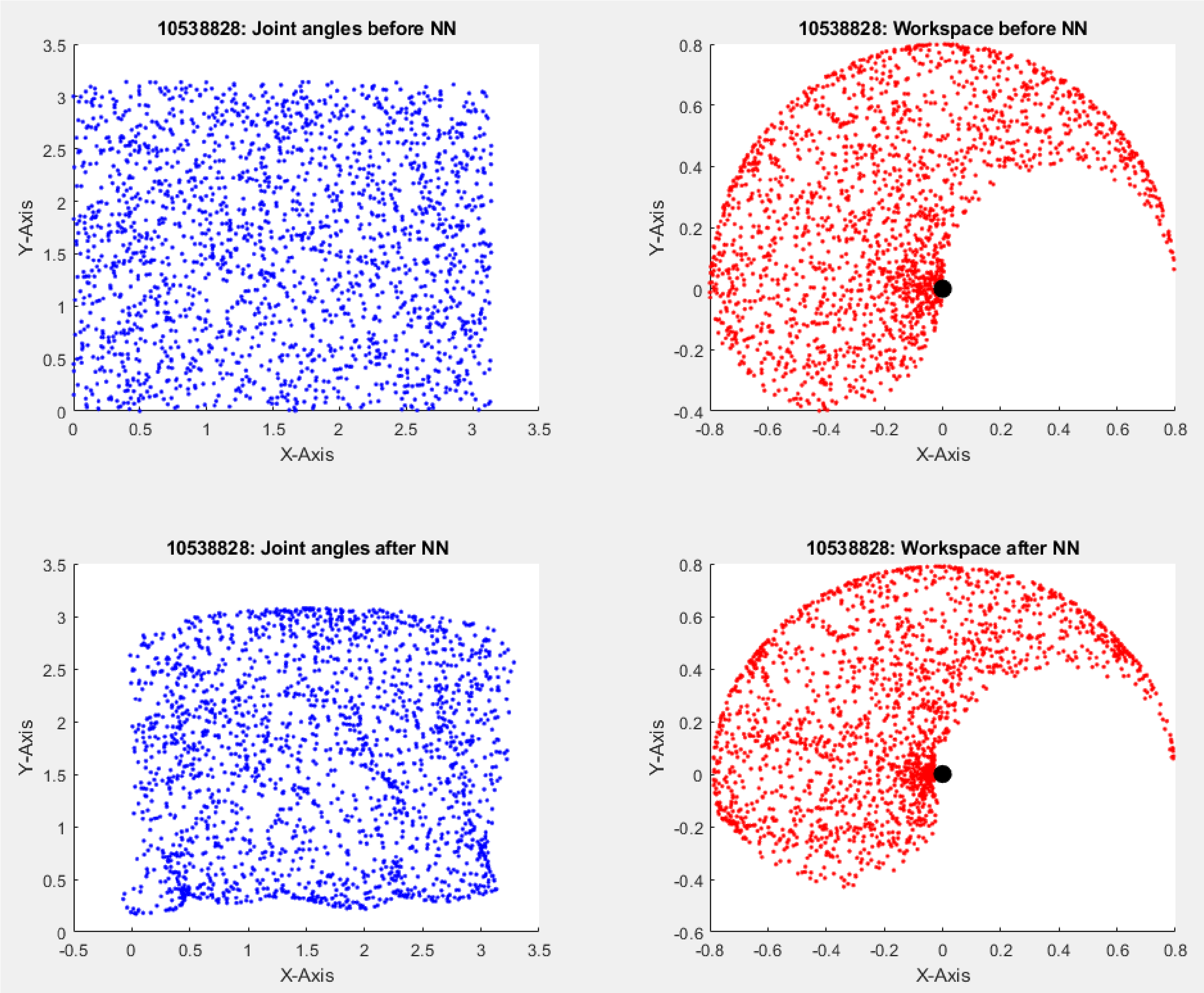


Figure 4 - Neural Network Raw Dataset Vs Neural Network Outputs

The plots above allow us to compare raw input and target data against the ones produced by our trained weights through feedforward pass and forward kinematics.

We can follow the sequence of this process in the four plots as follows:

Top left, test dataset target values (Joint Angles):

```
%___Generate Test Data
TestTheta1 = NN.GenThetaSet(nOfSamples);
TestTheta2 = NN.GenThetaSet(nOfSamples);
%___Augment Test Data
TestTheta = [Theta1;Theta2;ones(1,length(Theta1))];
```

Top Right, test dataset input values (X and Y Coordinates):

```
%___Perform Forward Kinematics
[TestP1,TestP2] = RevoluteForwardKinematics2D(armLen, TestTheta, origin);
%___Augment Test Inputs
TestP2 = [TestP2;ones(1,size(TestP2,2))];
```

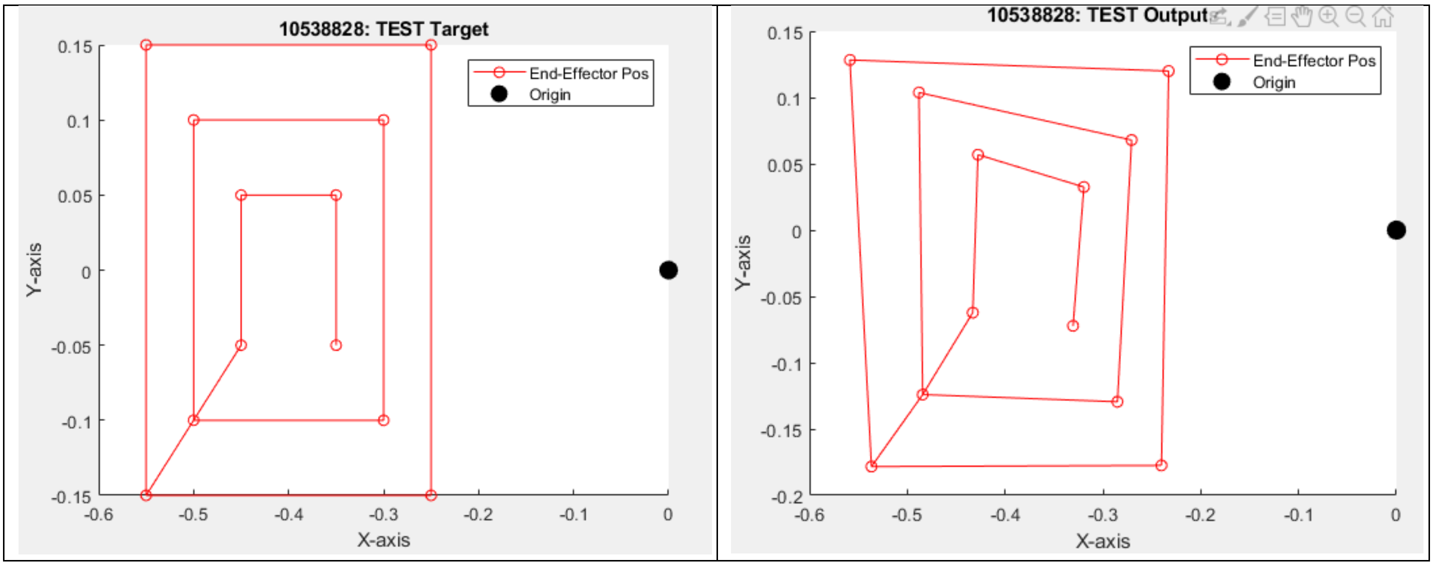
Bottom left, neural network outputs (Joint angles):

```
%___FeedForward Pass
net2 = W1*TestP2;                   % Calculate net2
a2 = 1./(1+exp(-net2));             % Sigmoid Activation
a2Hat = [a2;ones(1,size(a2,2))];    % Augment input for top layer
O = W2*a2Hat;                       % net3 / output
```

Bottom right, neural network outputs after forward kinematics (X and Y Coordinates):

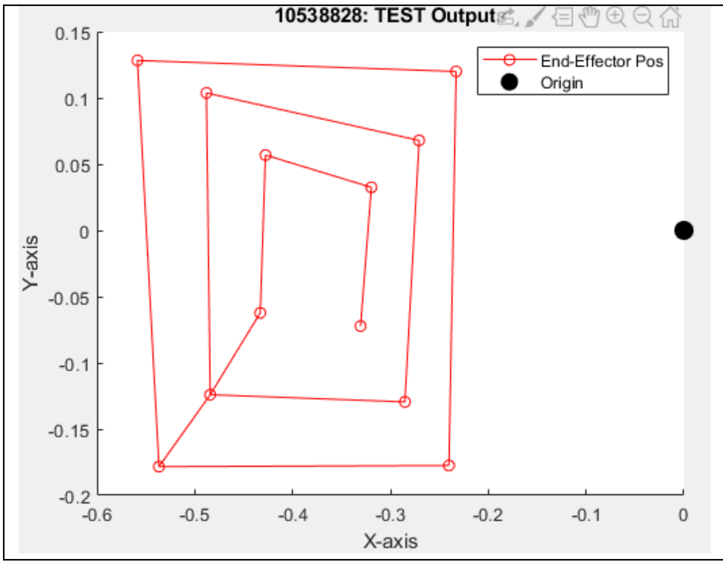
```
%___Perform Forward Kinematics
[TestPos1,TestPos2] = RevoluteForwardKinematics2D(armLen, O, origin);
```

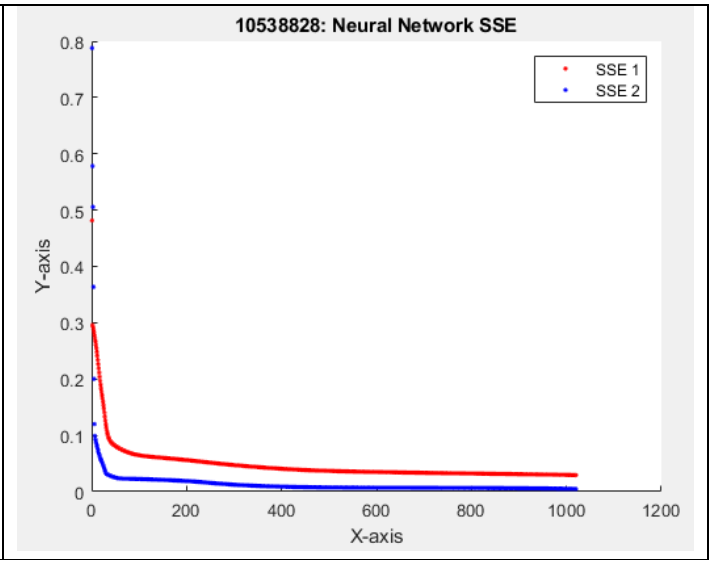
- Is there better datasets to interpret inverse model performance?
  We can use a set of rectangles that lye near the edges, the middle and near the center of the maze. Testing against these, allows us to not only check how accurate the network's outputs are but also where the inaccuracies / distortions lye.



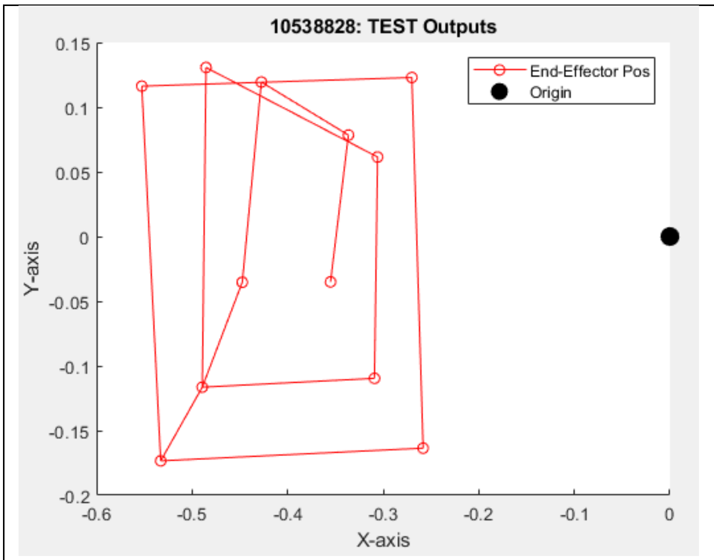
- Experiment with different numbers of hidden units, training times and learning rates.
- Also experiment with different training data sets.

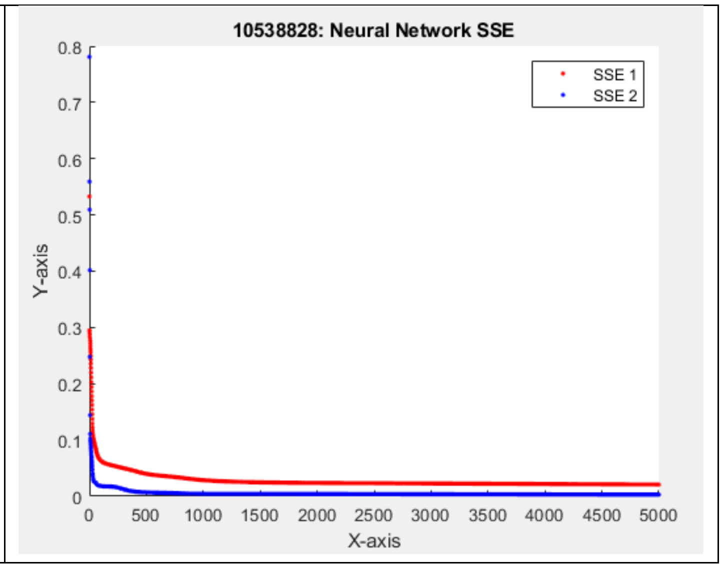
Hidden Units = 40, Iterations = 1020, Samples = 2000, Learning Rate = 0.02



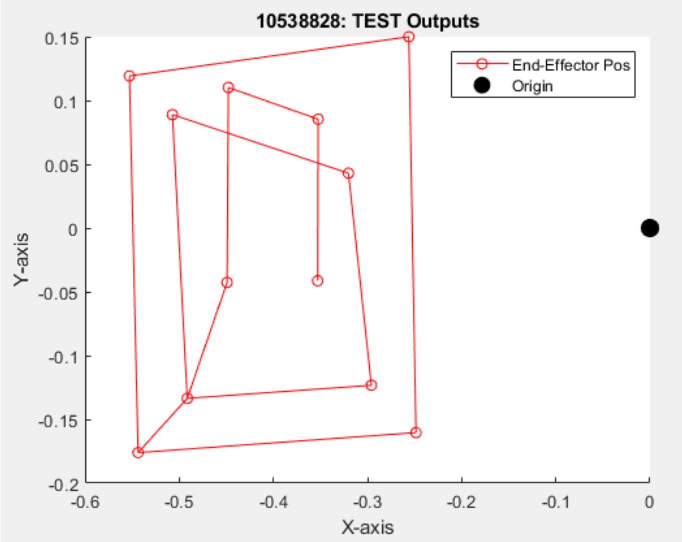


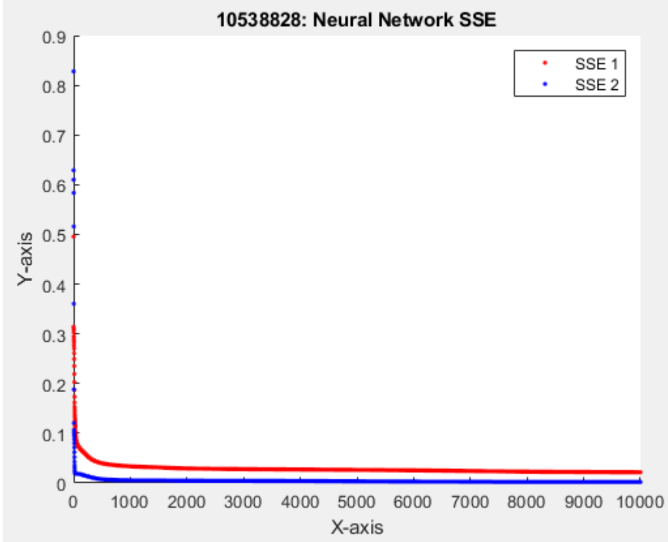
Hidden Units = 40, Iterations = 5000, Samples = 2000, Learning Rate = 0.02



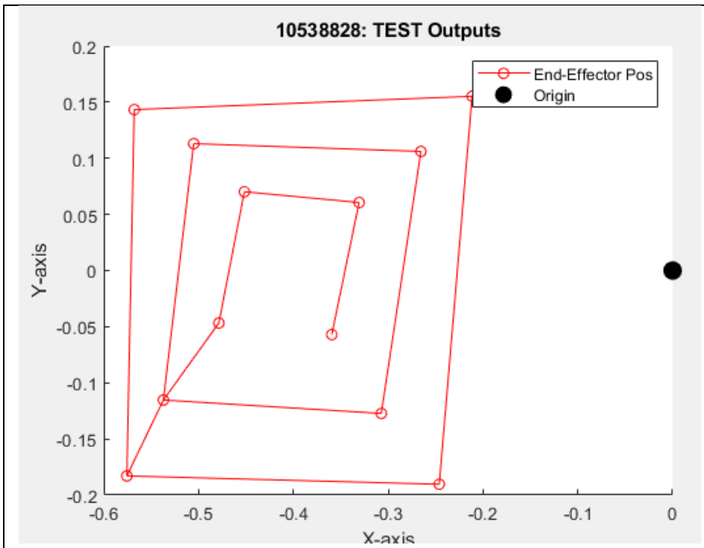


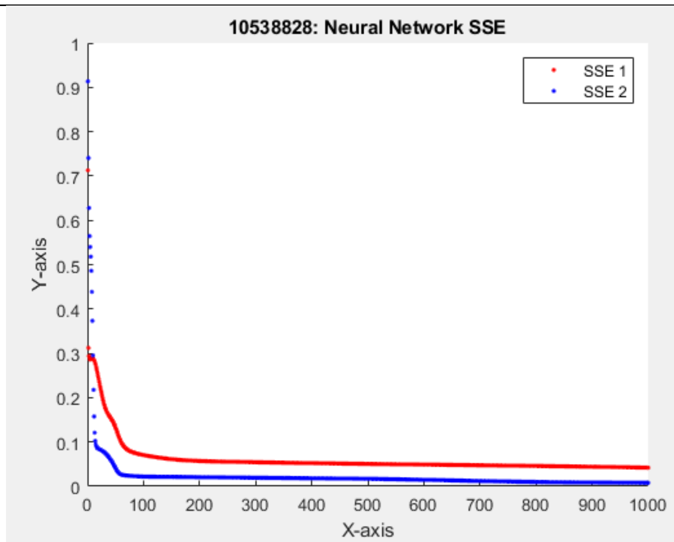
Hidden Units = 40, Iterations = 10000, Samples = 2000, Learning Rate = 0.02



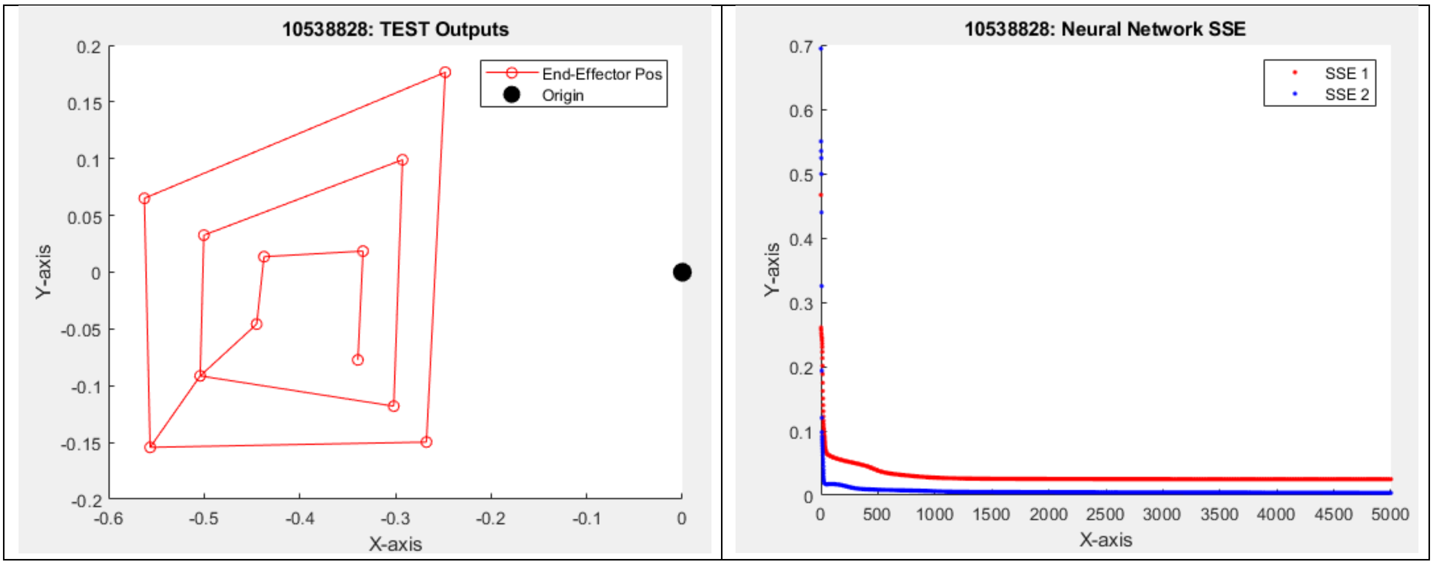


Hidden Units = 10, Iterations = 1000, Samples = 1000, Learning Rate = 0.02

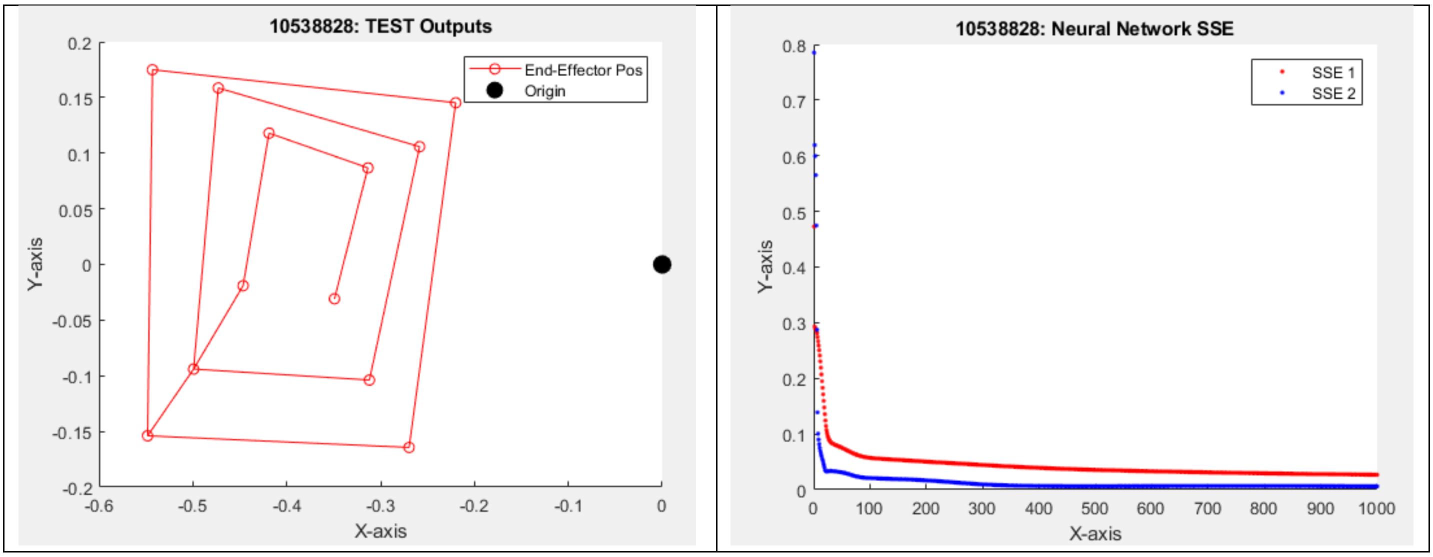




Hidden Units = 12, Iterations = 5000, Samples = 2000, Learning Rate = 0.02



Hidden Units = 40, Iterations = 1000, Samples = 2000, Learning Rate = 0.025



These results were using Xavier Initialisation of the weights. Xavier initialisation sets the weights close to one, i.e. just above and below 1. This stops gradients from vanishing or exploding too quickly.

$$\text{Xaveir Initialisation} = \sqrt{\frac{1}{number\ of\ inputs + number\ of\ targets}}$$

- How can you make the dataset more representative of the maze task?
  Looking at the workspace of the revolute arm in Figure 4 bottom right, we can see that a reasonable place to fit the maze would be the rectangular area defined by X = [-0.6,-0.2] and Y = [-0.2,0.2]. This means that any area outside these coordinates is not relevant to us and therefore we can use the relevant area only to train and test our neural network. To do this we can generate the dataset for training and for testing with joint angles that would produce the end-effector coordinates within these limits.

3. PATH THROUGH A MAZE

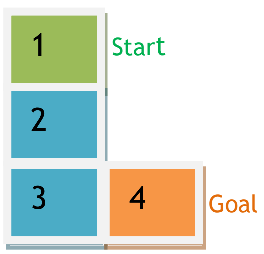
In this section we are going to solve the maze and find the shortest route available from the starting state to the goal state. To do this, we will use Q-Learning which is a Reinforcement Learning algoritm. Reinforcement Learning works by learning a policy and generating an action for a given state.

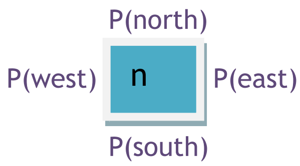
This is an iterative process and it works by gradually tuning our policy (QValues in this instance). Our training process require the following components:

- State
- Action
- Reward
- Next State
- Q-Values / Q-Table
- Learning Rate (α)
- Exploration Rate (ε)
- Gamma (γ) = 1 - ε

To understand the process, we must first understand what Q-Table is. The Q-Table has number of rows = number of states and number of columns = number of possible actions. Values in the columns describe the probability of choosing an action for a given state (see Figure 5). The values in the Q-Table are adjusted during the training process and the aim is to get 1 value that is higher than the others in each row. This value helps us decide which action to take.

MAZE	Action Probability	Q-Table





Let:
north = 1, east = 2,
south = 3, west = 4

S1	P(north)	P(east)	P(south)	P(west)
S2	P(north)	P(east)	P(south)	P(west)
S3	P(north)	P(east)	P(south)	P(west)
S4	P(north)	P(east)	P(south)	P(west)

Figure 5 - Q-Table Explaination

To train the algorithm we start at a random starting point and keep moving around the maze until we hit the goal state and we do this n number of times, where n is defined depending on the performance of the algorithm. One full sequence of moves from starting state to goal state is referred to as an Episode and a number of episodes is referred to as a Trial and finally a number of Trials is referred to as an Experiment.

The Q-Table is updated using the following equation:

Update Q-Table in row for current state and column for selected action

Q-Table value in row for current state and column for selected action

Reward, either based on current state only or based on the selected action for the current state

Highest value in the row for the next state

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[R_{t+1} + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

Learning Rate

Amount of times the policy is used instead of exploring

Q-Table value in row for current state and column for selected action

To understand the equation above I find it easier to look at an example. Using Figure 5 as an example, when the goal state (state 4) is reached a reward is given and a factor of that is added to the value in Q-Table row = 4 (row for the goal state) and column = action selected. Then in the second episode when the goal state is reached not only the Q-Table values for the goal state is increased but because of the part maxQ(S_{t+1},a) in the equation, the maximum value of Q-Table row = goal state is also added to Q(3,2) (which is state 3 column 2 for action east) and then in the third episode Q-Table value for goal state, state 3 and state 2 are increased and so on. This distribution of consistently increasing reward along a path from goal state to starting state allows us to solve the maze by following the maximum values for a state in the Q-Table for a given action. While this gives us lots of solutions with various paths and starting states, by recording the number of steps and the state each episode started on, we can choose the ones with starting state we require, and the minimum steps taken to solve from these episodes. This allows us to find the shortest path from the starting state required.

The paragraph above explains how the maze is solved by using only the best known solution, however in some cases we might think that the solution we have is the best but a better solution might exist. To reduce the possibility of this happening, we introduce exploration in the algorithm. This is achieved by generating a random action instead of using the Q-Table to generate one, at a given rate. This is called an Exploration Greedy ($\varepsilon - Greedy$) function (see Section 3.7 Exploration Greedy Funcion).

3.1. Random start state

- Write a function to generate a random starting state in the maze. NB: The start state may not be a blocked state or the goal state.

Call function to generate either a randomly generated starting state or start at state = 1

```
%% Generate Random Starting location
self.state = self.RandomStartingState(self.fixedStartingState);
```

Function for generating starting state

```
% function computes a random starting state
function startingState = RandomStartingState(self,fixedStartingState)
    % Check if a fixed starting state at state 1 is required
    switch fixedStartingState
        case true
            startingState = 1; % set starting state to 1
        case false
            range = [1,100];
            goal_state = self.goalState;
            startingState = goal_state; % initiate starting state as the goal state

            % loop until starting state is not goalstate or starting state is not
            % one of the blocked states.
            blockedStateFlag = true;
            while ((startingState == goal_state) ||...
                    blockedStateFlag == true)

                % set random starting state between 1 and 100
                startingState = randi(range,1,1);

                % loop through all blocked states and set flag to false
                % if the random starting state was one of the
                % blocked state. Array of blocked states generated
                % in the maze class
                for i = 1:length(self.blockedStates)
                    % Check if next state lyes within the blocked locations
                    if (startingState == self.blockedStates(i))
                        blockedStateFlag = false;
                    end
                end
            end
```

```
            %% These warnings are only displayed if the algorithm failed and the
            %  returned state was set to a blocked state or the goal state
            %______________Warning messages for debugging_____________%
            for i = 1:length(self.blockedStates)
                % Check if next state lyes within the blocked locations
                if (startingState == self.blockedStates(i))
                    warning("STATE WAS INITIATED IN A BLOCKED STATE!!");
                end
            end
            if (startingState == goal_state)
                warning("STATE WAS INITIATED AS THE GOAL STATE!!");
            end
            %_________________________________________________________%
        end
    end
%------------------------------------------------------------%
```

Blocked states array

```
blockedStates = [96,82,83,84,86,89,72,73,74,76,...
                 78,79,64,68,51,54,56,57,60,41,...
                 44,45,46,47,50,31,34,35,21,23,...
                 24,28,29,16,18,19,4];
```

- Generate 1000 starting states and plot a histogram using the Matlab histogram function to check your function is working.

Call function to generate 1000 starting states and plot historgram

```
%___Generate thousand starting states and plot histogram
self.thousandStartingStates(1000);
```

Function to generate starting states

```
%%
function thousandStartingStates(self,nOfStates)
    for i = 1:nOfStates
        sStates(i) = self.RandomStatingState();
    end
    figure
    hold on
    histogram(sStates,100);
    title("10538828: Histogram of Starting States");
    hold off
end
```

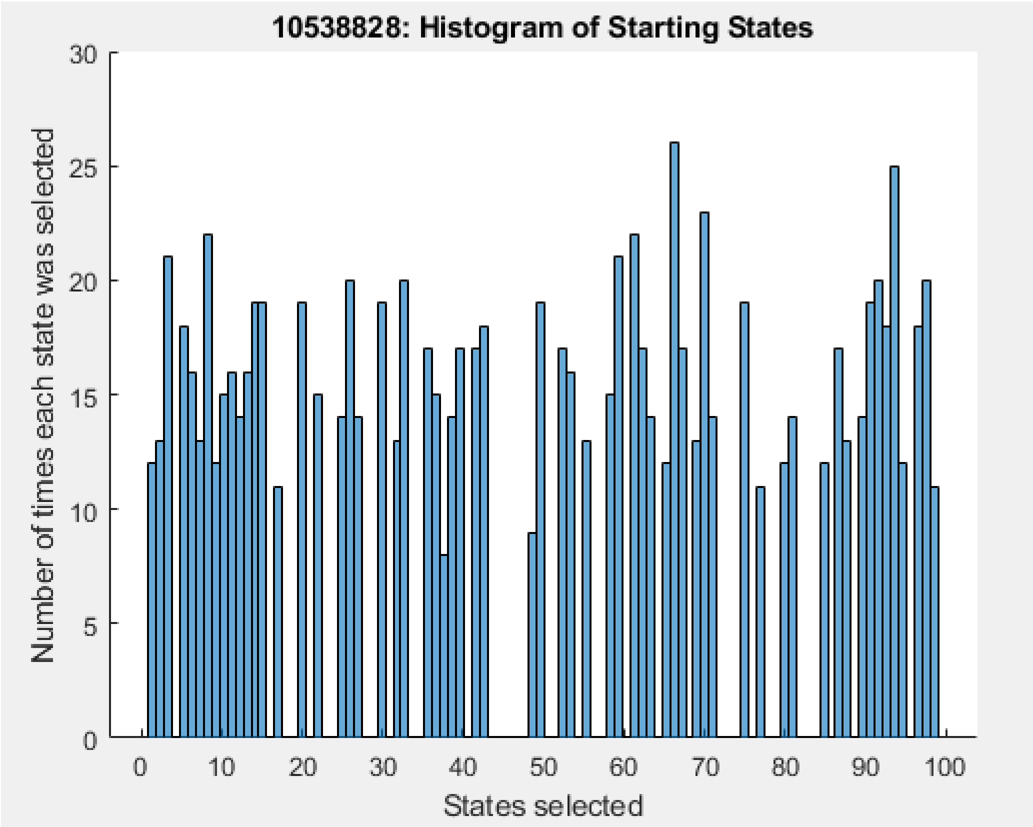


Figure 6 - Thousand Random Starting States

- Comment on how the displayed state occurrences align with the maze.

Out of all the thousand randomly generated starting states (x-axis) we expect no starting state to be the goal state (state 100) or any of the blocked states in the array below. The histogram shows the states generated on the x-axis and the number of times each state was generated on the y-axis. Therefore, all the gaps / missing bars can be observed on the goal state and blocked states.

```
blockedStates = [96,82,83,84,86,89,72,73,74,76,...
                 78,79,64,68,51,54,56,57,60,41,...
                 44,45,46,47,50,31,34,35,21,23,...
                 24,28,29,16,18,19,4];
```

3.2. Build a reward function

- Specify the reward function for the maze

```
% reward function that takes a state and an action
function reward = RewardFunction(self, state, action)
    % Reward = 10 only if the state is the goal state (state 100)
    % else Reward = 0
    if (state == 100)
        reward = 10;
    else
        reward = 0;
    end
end
```

3.3. Generate the transition matrix

- Specify a transition matrix for the maze.

```
function nextState = BuildTransitionMatrix(self,action,state)
    % set boundry limits of the maze in relation to the action
    % required
    boundryNorth = 91:100;
    boundryEast  = linspace(10,100,10);
    boundrySouth = 1:10;
    boundryWest  = linspace(1,91,10);
    boundryHit = false;

    switch action
        %______________
        case self.north
            %_Increase by 10 for next state if going north
            nextState = state + 10;

            % if nextState is out of boundry set boundryHit flag
            for i = 1:length(boundryNorth)
                if (state == boundryNorth(i))
                    boundryHit = true;
                end
            end
        %______________
        case self.east
            %_Increase by 1 for next state if going east
            nextState = state + 1;

            % if nextState is out of boundry set boundryHit flag
            for i = 1:length(boundryEast)
                if (state == boundryEast(i))
                    boundryHit = true;
                end
            end
```

```
    %______________
    case self.south
        %_Decrease by 10 for next state if going south
        nextState = state - 10;

        % if nextState is out of boundry set boundryHit flag
        for i = 1:length(boundrySouth)
            if (state == boundrySouth(i))
                boundryHit = true;
            end
        end
    %______________
    case self.west
        %_Decrease 1 for next state if going west
        nextState = state - 1;

        % if nextState is out of boundry set boundryHit flag
        for i = 1:length(boundryWest)
            if (state == boundryWest(i))
                boundryHit = true;
            end
        end
    %______________
    otherwise
        error("invalid action")
end
```

```
    % if boundryHit flag was set, reset nextState to current state
    if (boundryHit == true)
        nextState = state;
    end

    for i = 1:length(self.blockedStates)
        % Check if next state lyes within the blocked states,
        % and if so then reset next state to current state.
        if (nextState == self.blockedStates(i))
            nextState = state;
        end
    end
end
```

3.4. Initialize Q-values

- Initialize the Q-values matrix to sensible numbers.

```
% init the q-table
function QValues = InitQTable(self)
    % Set Q-Value limits
    yLower = 0.01;
    yUpper = 0.1;

    % Get number of states and actions
    R = self.totalNumberOfStates;
    C = self.totalNumberOfActions;

    % Calculate range and mean
    range = yUpper - yLower;
    Mean = (yUpper + yLower)/2;

    % generate N random numbers in the interval (a,b)
    % with the formula r = a + (b-a).*rand(N,1)
    y = (rand(R,C))*range + Mean;
    QValues = y;
end
```

3.5. Implement Q-learning algorithm

- In this assignment your algorithm will need an outer loop to run 100 trials.
- In this assignment within a trial, you will need a loop to run 1000 episodes.
- Within each episode you will need a loop to run for a given number of states until the goal state is reached.
- The number of steps needed in an episode is indicative of how good the Qlearning policy is becoming, so it can be used as an indication of algorithm performance on the training data.

```
% Run Experiments
[self,self.ExperimentRecord] = self.Experiments();
```

```
%% Runs Q-Learning Experiments
function [self,ExperimentRecord] = Experiments(self)

    ExperimentRecord = [];
    for i = 1:self.reqTrials % Loop for the required amount of Trials
        fprintf("trial %d/%d\n", i, self.reqTrials);
        [self,ExperimentRecordBuff] = self.Trials(); % Run a single Trial

        % Store all states visited and steps taken to solve the maze for
        % each episode and for all Trials
        ExperimentRecord(i).Trials = ExperimentRecordBuff;
    end
end
```

```
%% Runs Q-Learning Trials
function [self,TrialRecord] = Trials(self)
    %___Initiate QTable
    self.QValues = self.InitQTable();

    % Run Trials
    TrialRecord = [];
    for i = 1:self.reqEpisodes % Loop for the required amount of Episodes
        [self,TrialRecordBuff] = self.Episodes(); % Run a single episode

        % Store all states visited and steps taken to solve the maze for
        % all Episodes
        TrialRecord(i).Episodes = TrialRecordBuff; %
    end
end
```

```
%% Run an Episode
function [self,EpRecord] = Episodes(self)

    %% Variables and Constants
    goalFlag = false;
    steps = [];
    epSteps = 1;
    epStateRecord = [];

    %% Generate Random Starting States
    self.state = self.RandomStartingState(self.fixedStartingState);

    %% loop until goal state has reached
    while (goalFlag == false)

        %% Generate action from exploration greedy function
        self.action = self.E_Greedy(self.QValues,self.state);

        %% Generate next state for new action
        self.nextState = self.BuildTransitionMatrix(self.action,self.state);

        %% Get reward for the action from the current state
        self.reward = self.RewardFunction(self.state, self.action);

        %% update QTable
        self.QValues = self.UpdateQ(self.QValues, self.state, self.action,...
            self.nextState,self.reward);

        %% Record States
        epStateRecord = [epStateRecord;self.state];

        %% update state
        self.state = self.nextState;
```

```
        %% Conditional loop exit condition
        if (self.state == self.goalState) % Enter if current state = goal state
            epStateRecord = [epStateRecord;self.state]; % Store states visited during current episode
            goalFlag = true;                            % set exit statement for the loop
            epSteps = epSteps + 1;                      % increment step
            steps(epSteps) = epSteps-1;                 % store steps taken
            EpRecord = [epStateRecord,steps'];          % Concatinate states visited and steps taken
        else
            epSteps = epSteps + 1;      % increment step
            steps(epSteps) = epSteps-1; % store steps taken
        end
    end
    %%
end
%------------------------------------------------------------%
```

3.6. Run Q-learning

- Run the Q-learning algorithm using:
  - An exploration rate of 0.1
  - A temporal discount rate gamma of 0.9
  - A learning rate alpha of 0.2.
- Analyse the performance of your Q -learning algorithm on the maze by running an experiment with 100 trials of 1000 episodes

```
%___Constants / Params
self.totalNumberOfStates = 100;
self.totalNumberOfActions = 4;
self.alpha = 0.2;
self.gamma = 0.9;
self.eRate = 0.1;
self.explorationRequired = false;
self.rewardType = 0;
self.reqEpisodes = 1000;
self.reqTrials = 100;
self.fixedStartingState = false;

% Run Experiments
[self,self.ExperimentRecord] = self.Experiments();
```

- Generate an array containing the means and standard deviations of the number of steps required to complete each episode

```
%% Calculate mean
function means = calcMean(self,ExperimentRecord)
    for Trial_i = 1:length(ExperimentRecord) % Loop through all trials
        for Episode_i = 1:length(ExperimentRecord(Trial_i).Trials) % Loop through all episodes
            episode = ExperimentRecord(Trial_i).Trials(Episode_i).Episodes; % Grab Episodes
            means(Trial_i,Episode_i) = sum(episode(:,2)) / size(episode,1); % Calculate and store mean
        end
    end
end
%-------------------------------------------------------------%

%% Calculate Standard Deviation
function stdDevs = calcStdDev(self,ExperimentRecord)
    for Trial_i = 1:length(ExperimentRecord) % Loop through all trials
        for Episode_i = 1:length(ExperimentRecord(Trial_i).Trials) % Loop through all episodes

            % Grab Episodes
            episode = ExperimentRecord(Trial_i).Trials(Episode_i).Episodes;% Grab Episodes
            % Calculate Variance
            variance = sum((episode(:,2) - self.mean(Trial_i,Episode_i)).^2) / size(episode,1);
            % Calculate Standard Deviation
            stdDevs(Trial_i,Episode_i) = sqrt(variance);

        end
    end
end
```

- Plot the mean and standard deviation across trials of the steps taken against episodes. Describe what you find.

In Figure 7 we can see the number of episodes performed on the x-axis and the steps taken to solve the maze in the y-axis.We can observe that the performance in the earlier episodes greatly improving, however as the episodes increase the improvement is slower.

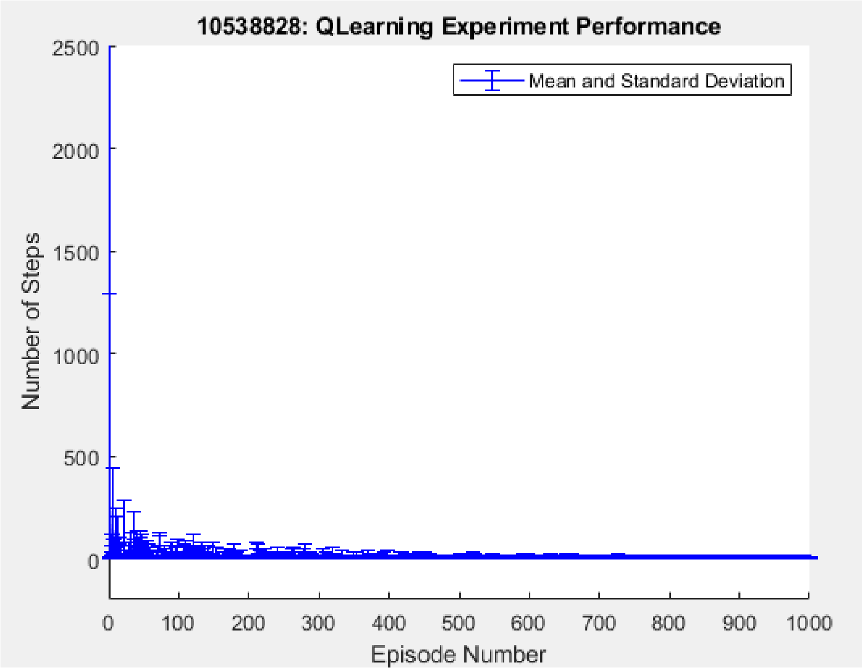


Figure 7 - Q-Learning Performance Improvement

3.7. Exploitation of Q-values

- Record the Q-table at the end of a training trial.

I choose one of the Q-Tables for the episode that had the starting state = 1. This is picked out during last Trial and the last episode with starting state = 1.

```
storeQTableFlag = false;

%% Generate Random Starting States
self.state = self.RandomStartingState(self.fixedStartingState);
if (self.state == 1)
    % Store all Trial and Episode numbers with starting state = 1
    self.startStateIsOne = [self.startStateIsOne;self.current_trials,self.current_episode];
    % set flag to store Q-Table
    storeQTableFlag = true;
end
```

```
if ((storeQTableFlag == true) && (self.current_trials == self.reqTrials))
    self.final_QTable = self.QValues;
end
```

QL.final_QTable

	1	2	3	4
1	0.0206	0.0213	0.0205	0.0203
2	0.0237	0.0218	0.0216	0.0215
3	0.0263	0.0229	0.0231	0.0229
4	0.0677	0.1100	0.1016	0.0882
5	0.0325	0.0291	0.0294	0.0290
6	0.0302	0.0358	0.0305	0.0300
7	0.0401	0.0327	0.0323	0.0327
8	0.0323	0.0322	0.0320	0.0360
9	0.0320	0.0395	0.0318	0.0322
10	0.0446	0.0344	0.0348	0.0341
11	0.0211	0.0236	0.0211	0.0212
12	0.0222	0.0263	0.0223	0.0224
13	0.0246	0.0292	0.0246	0.0241
14	0.0265	0.0325	0.0268	0.0262
15	0.0361	0.0296	0.0295	0.0296
16	0.0883	0.1108	0.1291	0.0564
17	0.0446	0.0352	0.0346	0.0351
18	0.0931	0.0855	0.0642	0.1338
19	0.0995	0.0968	0.1139	0.1212
20	0.0495	0.0371	0.0373	0.0373
21	0.0816	0.0616	0.1439	0.0680
22	0.0207	0.0208	0.0237	0.0205
23	0.0897	0.0855	0.1015	0.0623
24	0.0643	0.1292	0.0641	0.0861
25	0.0321	0.0401	0.0319	0.0321
26	0.0352	0.0446	0.0349	0.0348
27	0.0495	0.0371	0.0365	0.0371
28	0.1223	0.1361	0.0911	0.0824
29	0.0560	0.1279	0.1445	0.0625
30	0.0550	0.0395	0.0399	0.0396
31	0.0778	0.0783	0.0884	0.1097
32	0.0188	0.0188	0.0213	0.0190
33	0.0175	0.0175	0.0177	0.0192

QL.final_QTable

	1	2	3	4
34	0.0987	0.1416	0.1449	0.0623
35	0.1284	0.0562	0.0660	0.1349
36	0.0376	0.0495	0.0371	0.0372
37	0.0402	0.0550	0.0398	0.0396
38	0.0437	0.0611	0.0433	0.0428
39	0.0679	0.0447	0.0453	0.0452
40	0.0421	0.0425	0.0428	0.0611
41	0.0748	0.0975	0.0707	0.0993
42	0.0172	0.0174	0.0192	0.0175
43	0.0164	0.0165	0.0165	0.0173
44	0.0813	0.1328	0.0696	0.1439
45	0.1233	0.0705	0.0667	0.0629
46	0.0884	0.1289	0.0978	0.1406
47	0.1407	0.1157	0.0812	0.1448
48	0.0471	0.0679	0.0466	0.0463
49	0.0755	0.0492	0.0480	0.0485
50	0.1439	0.0642	0.1090	0.0659
51	0.1285	0.1064	0.0560	0.1183
52	0.0160	0.0160	0.0173	0.0162
53	0.0152	0.0155	0.0155	0.0155
54	0.0945	0.0819	0.0667	0.1280
55	0.0550	0.0367	0.0367	0.0368
56	0.1090	0.0764	0.0613	0.1433
57	0.1115	0.1424	0.1161	0.1003
58	0.0494	0.0754	0.0487	0.0495
59	0.0839	0.0518	0.0512	0.0518
60	0.0825	0.1360	0.1065	0.1049
61	0.0213	0.0165	0.0162	0.0164
62	0.0154	0.0152	0.0155	0.0188
63	0.0143	0.0141	0.0143	0.0158
64	0.1444	0.1009	0.1140	0.1169
65	0.0369	0.0611	0.0376	0.0378
66	0.0413	0.0679	0.0411	0.0404

QL.final_QTable

	1	2	3	4
67	0.0755	0.0449	0.0449	0.0449
68	0.0909	0.1392	0.0870	0.0963
69	0.0545	0.0932	0.0539	0.0549
70	0.1036	0.0550	0.0546	0.0556
71	0.0237	0.0174	0.0177	0.0175
72	0.1217	0.0873	0.0812	0.0718
73	0.1029	0.0735	0.0799	0.0847
74	0.0564	0.1309	0.1075	0.1042
75	0.0346	0.0343	0.0550	0.0343
76	0.0859	0.1264	0.0743	0.0950
77	0.0839	0.0484	0.0483	0.0484
78	0.1203	0.0687	0.0838	0.0989
79	0.0851	0.1344	0.1184	0.1411
80	0.1151	0.0705	0.0701	0.0700
81	0.0263	0.0191	0.0187	0.0190
82	0.1113	0.0841	0.0982	0.0965
83	0.1337	0.0697	0.1311	0.0616
84	0.1333	0.1438	0.0622	0.0796
85	0.0313	0.0315	0.0495	0.0319
86	0.0613	0.0863	0.1336	0.0765
87	0.0523	0.0932	0.0521	0.0516
88	0.1036	0.0535	0.0539	0.0535
89	0.0876	0.0622	0.1132	0.1296
90	0.1278	0.0559	0.0560	0.1018
91	0.0204	0.0292	0.0206	0.0206
92	0.0224	0.0325	0.0221	0.0220
93	0.0240	0.0361	0.0244	0.0244
94	0.0267	0.0401	0.0267	0.0261
95	0.0287	0.0291	0.0446	0.0290
96	0.1210	0.1124	0.0598	0.0632
97	0.0530	0.1031	0.0533	0.0531
98	0.0546	0.1151	0.0548	0.0544
99	0.0658	0.1278	0.0650	0.0654
100	0.1421	0.1217	0.1099	0.0904

Figure 8 - Trained Q-Table

- Write an exploitation function that makes use of the Q-values and makes greedy action selection ***without exploration.***
- Select the starting state as the green state shown on the maze

```
%%
function [self,finalPath] = ExploitationQValues(self,QTable)
    %___Constants / Params
    self.explorationRequired = false;
    self.fixedStartingState = true;
```

Exploration Greedy Function

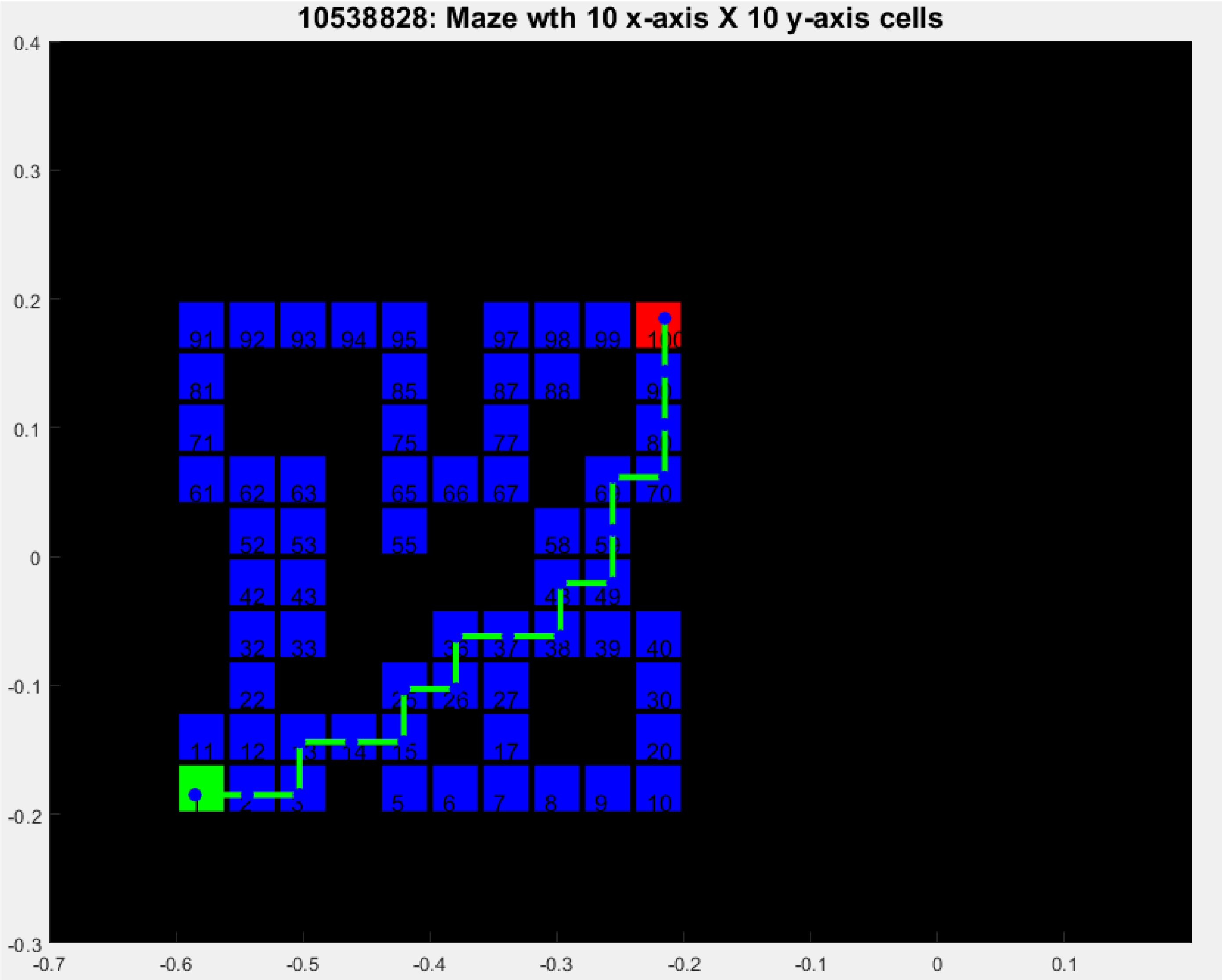
```
% Generate action based on the QValues and if required
% some exploration, where rate of exploration is eRate
function action = E_Greedy(self,QTable,state)
    e = rand(1,1);

    if ((e <= self.eRate) && (self.explorationRequired == true))
        range = [1,4];
        action = randi(range,1,1);
    else
        [~,action] = max(QTable(state,:));
    end
end
```

Starting State Funtcion (Same as Section 3.1)

- Record the visited states for this episode.
- Convert the state into a 2-D coordinate that you can plot out in a matrix.
- This should take the form of a 2xN matrix where the first dimension relates to the (x,y) coordinates of the data points, and the second to the N of steps in the episode
- Try to plot out the path over a drawing of the maze.

Path as states			States to 2D coordinates				2D coordinates to 2D coordinates scaled for the maze plot			
	States	Steps		X	Y	Steps		X	Y	Steps
	1	2		1	2	3		1	2	3
1	1	0	1	1	1	0	1	-0.5850	-0.1850	0
2	2	1	2	2	1	1	2	-0.5439	-0.1850	1
3	3	2	3	3	1	2	3	-0.5028	-0.1850	2
4	13	3	4	3	2	3	4	-0.5028	-0.1439	3
5	14	4	5	4	2	4	5	-0.4617	-0.1439	4
6	15	5	6	5	2	5	6	-0.4206	-0.1439	5
7	25	6	7	5	3	6	7	-0.4206	-0.1028	6
8	26	7	8	6	3	7	8	-0.3794	-0.1028	7
9	36	8	9	6	4	8	9	-0.3794	-0.0617	8
10	37	9	10	7	4	9	10	-0.3383	-0.0617	9
11	38	10	11	8	4	10	11	-0.2972	-0.0617	10
12	48	11	12	8	5	11	12	-0.2972	-0.0206	11
13	49	12	13	9	5	12	13	-0.2561	-0.0206	12
14	59	13	14	9	6	13	14	-0.2561	0.0206	13
15	69	14	15	9	7	14	15	-0.2561	0.0617	14
16	70	15	16	10	7	15	16	-0.2150	0.0617	15
17	80	16	17	10	8	16	17	-0.2150	0.1028	16
18	90	17	18	10	9	17	18	-0.2150	0.1439	17
19	100	18	19	10	10	18	19	-0.2150	0.1850	18

Figure 9 - Path Conversion (Exploitation of Q-Values)


10538828: Maze wth 10 x-axis X 10 y-axis cells
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.7
-0.6
-0.5
-0.4
-0.3
-0.2
-0.1
0
0.1

4. MOVE ARM ENDPOINT THROUGH MAZE

4.1. Generate kinematic control to revolute arm

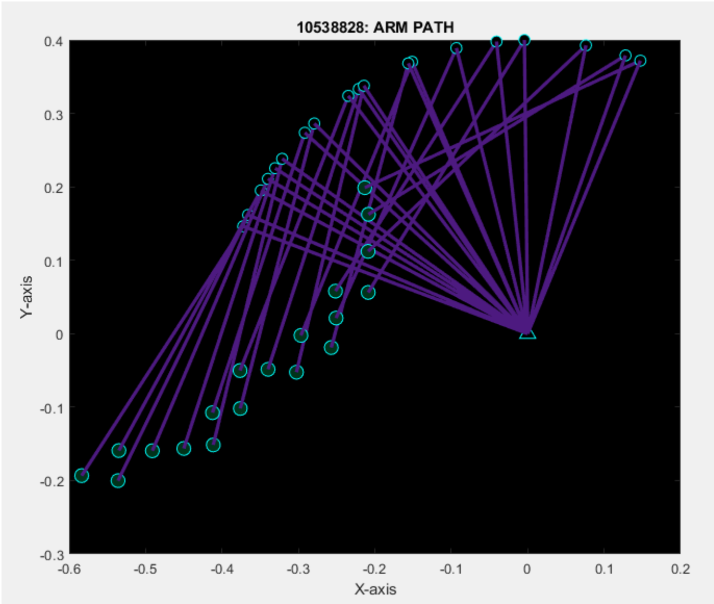
- Finally use the maze path to specify the endpoint trajectory of the 2-joint revolute arm.
- Use the inverse kinematic neural network you trained earlier to generate the arm's joint angles.
- Tip: ensure you have scaled the maze so that it fits into the workspace of the revolute arm!
- Use the forward kinematic function with these angles as input to calculate the arm elbow and endpoint positions.

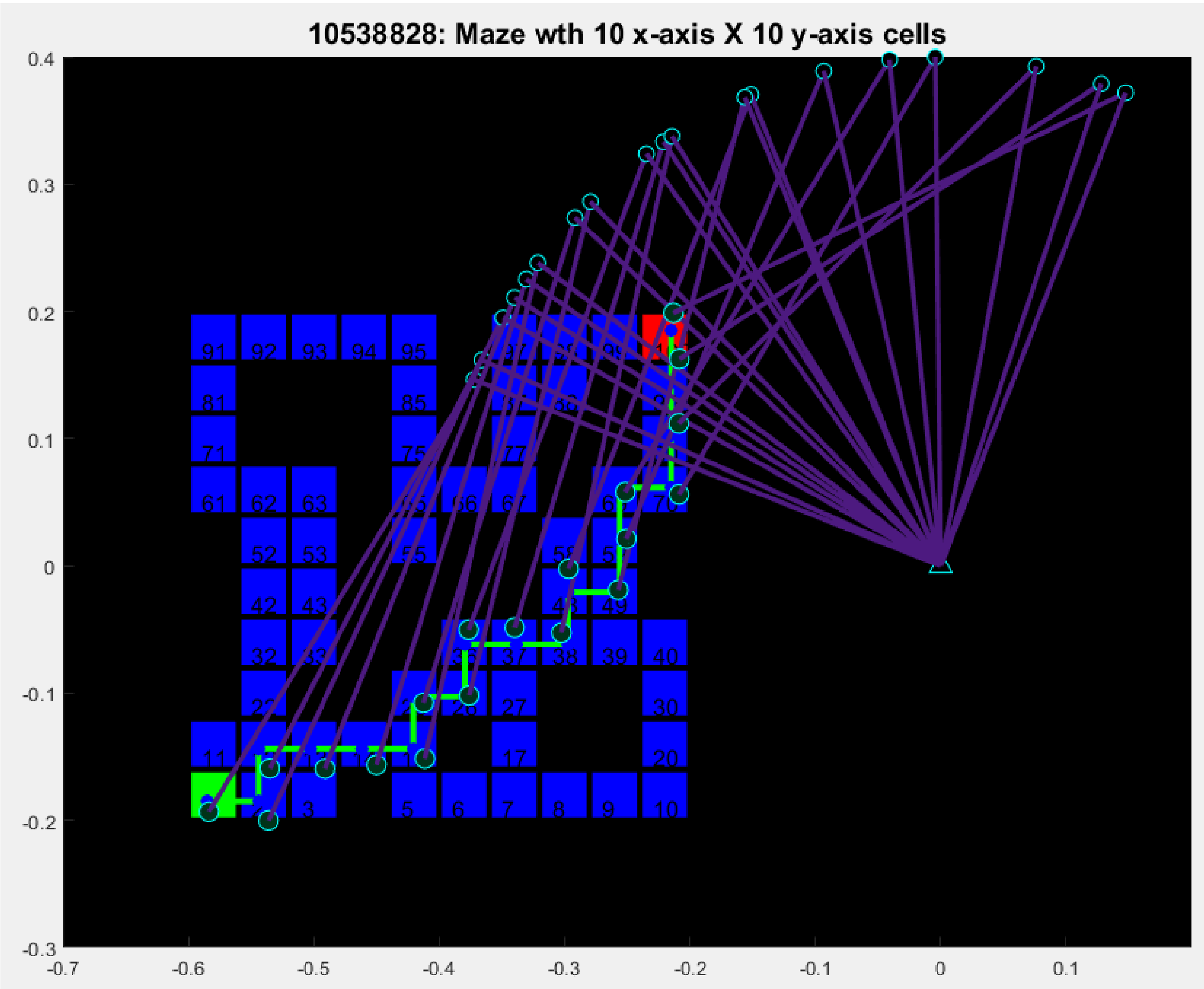
Main

```
limits = [-0.6 ,-0.2; -0.2, 0.2]; % Set maze size / limits
Origin = [0.0,0.0];               % set revolute arm origin
fk = ForwardKinematics;           % get forward kinematics class
maze = CMazeMaze10x10(limits);    % get maze class
QL = QLearning(maze);             % Run Q-Learning to solve the maze

PosEE = QL.ScaledPath';           % get scaled path solution from Q-Learning this will be the input for the FeedForward Pass
fk.plotWorkspaceLine("TEST Inputs",PosEE(1,:),PosEE(2,:),Origin) % Plot scaled path solution

%___Augment inputs
PosEE = [PosEE;ones(1,size(PosEE,2))];

%___FeedForward Pass
net2 = W1*PosEE;                 % Calculate net2
a2 = 1./(1+exp(-net2));           % Sigmoid Activation
a2Hat = [a2;ones(1,size(a2,2))]; % Augment input for top layer
net3 = W2*a2Hat;                      % net3 / output

[NN_P1,NN_P2] = RevoluteForwardKinematics2D(armLen, net3, Origin); % Feed NN joint angles into FKinematics

fk.plotWorkspaceLine("TEST Outputs",NN_P2(1,:),NN_P2(2,:),Origin) % Plot FeedForward Pass outputs after FK

%% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fk.plotArm("ARM PATH",...                          % Plot revolute arm from FeedForward Pass outputs after FK
     NN_P1(1,:),NN_P1(2,:),...
     NN_P2(1,:),NN_P2(2,:),...
     Origin);

maze.DrawMaze(QL.ScaledPath(:,1:2),NN_P1',NN_P2'); % Draw maze with Q-Learning solution

fk.AnimateArm(NN_P1(1,:),NN_P1(2,:),...            % Animate revolute arm on the maze plot
              NN_P2(1,:),NN_P2(2,:),...
              Origin);
```

Scaling Function in Q-Learning

```
function State2Loc = states2locations(self,bestPath)

    X = [];
    Y = [];

    for i = 1:size(bestPath,1)
        X(i) = rem(bestPath(i,1), 10);
        if (X(i) == 0)
            X(i) = 10;
        end
        Y(i) = ((bestPath(i,1) - X(i)) / 10) + 1;
    end

    State2Loc = [X',Y',bestPath(:,2)];

end
```


10538828: ARM PATH
Y-axis
X-axis


10538828: Maze wth 10 x-axis X 10 y-axis cells

4.2. Animated revolute arm movement

- Generate an animation of the endpoint of the revolute arm moving through the maze. Also draw the arm as well.
- Produce a video of your results and put a link to the video uploaded to YouTube in your report.
- Tip A screenshot of my implementation of this animation is shown in Fig. 10.

Please click on the picture below or follow the link in the caption

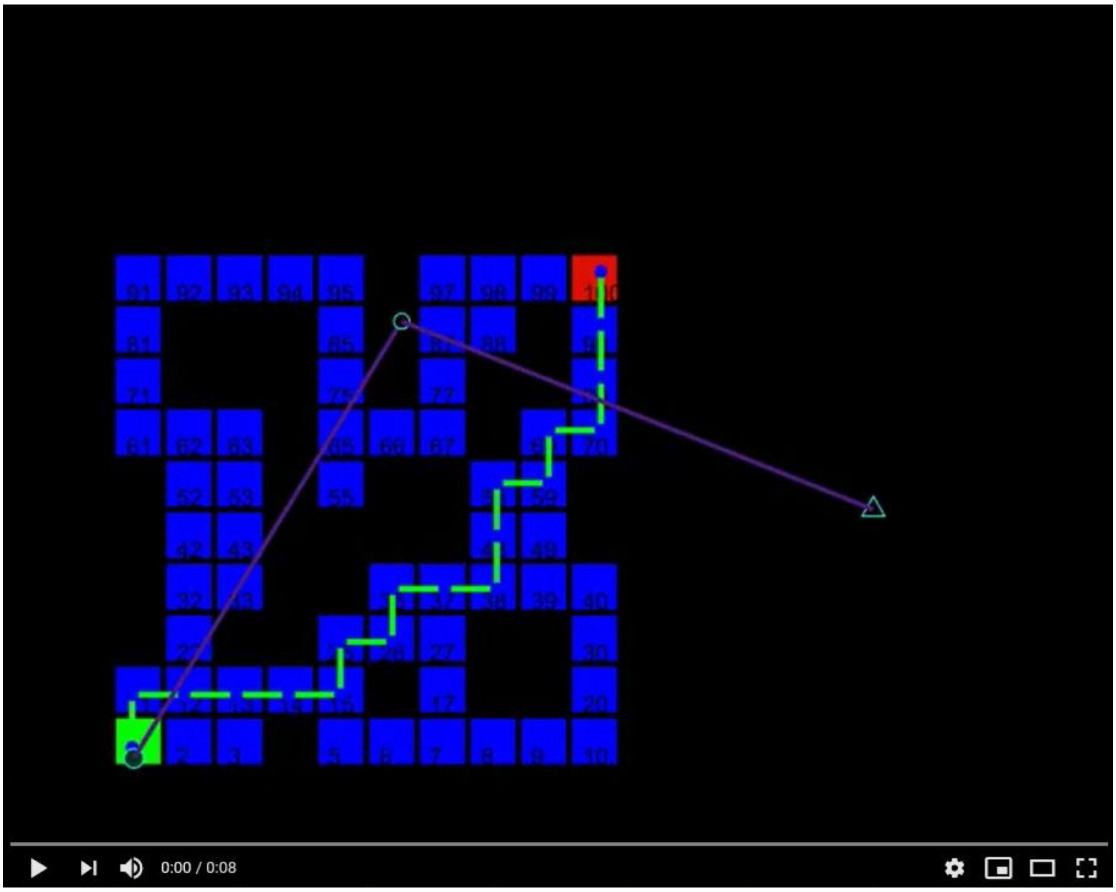


Figure 10 – Animation
(https://www.youtube.com/watch?v=anM-CoG9xyc)