1. Training data generation

1.1 Implement revolute arm forward kinematics

To perform the forward kinematics, the function needs to take the parameters of a robot arm and calculate its end point and elbow positions based on the angle data.

```
function [P1 P2] = RevoluteForwardKinematics2D(armLen, theta, origin)
% calculate relative forward kinematics

%armLen = [L1 L2]
%theta = [theta1 theta2]
%origin = [x0 y0]   base
%P1 = [xp1 yp1]     elbow
%P2 = [xp2 yp2]     end point

%keyboard
Theta12 = theta(:,1) + theta(:,2);

P1(:,1) = origin(1,1).*
ones(size(theta(:,1)))+armLen(1,1)*cos(theta(:,1));
P1(:,2) = origin(1,1).*
ones(size(theta(:,1)))+armLen(1,1)*sin(theta(:,1));


P2(:,1) = origin(1,1).* ones(size(theta(:,1)))
+(armLen(1,1)*cos(theta(:,1))) + (armLen(1,2)*cos(Theta12));
P2(:,2) = origin(1,2).* ones(size(theta(:,1)))
     + (armLen(1,1)*sin(theta(:,1))) + (armLen(1,2)*sin(Theta12));
```

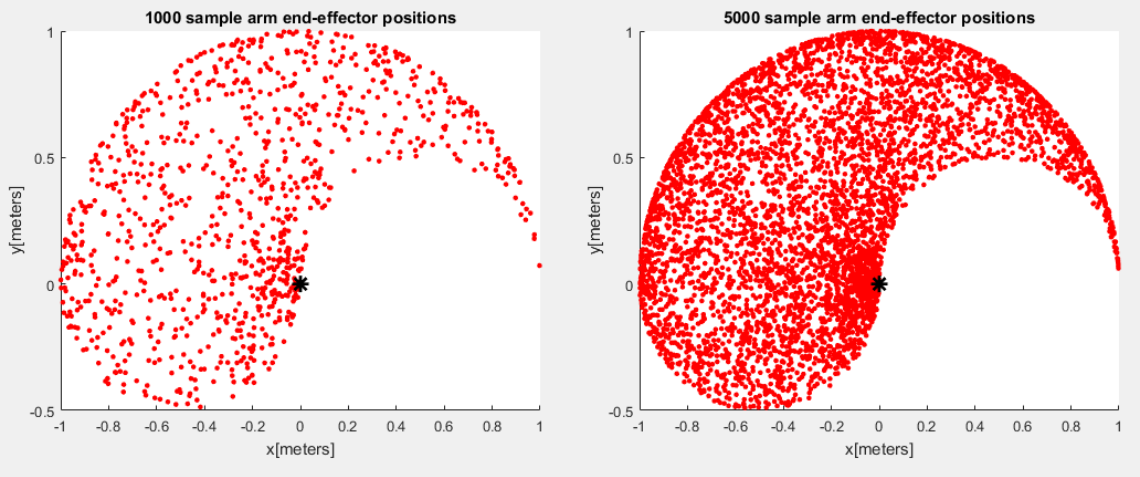
I calculated the X and Y coordinates of each points separately, the X dimention using Cos on the theta angles and Sin for the Y equation. This then provides 2 N-by-2 matrixes, with N being the amount of samples dictated by my parameters.

```
armLen = [0.5 0.5];      % L1, L2
theta = [1.5 -1.5];  % (radians) thetaOne, thetaTwo presets for testing
origin = [0 0];
samples = 50;
angles = (3.14).*rand(samples,2);

  [P1, P2] = RevoluteForwardKinematics2D(armLen, angles, origin);
```

1.2 Display Workspace of Revolute Arm

Next was to display the range of movement available to the robot arm. This was done by plotting the endpoints over a wide sample count.

```
figure
hold on
title('sample arm end-effector positions')
h=plot(P2(:,1),P2(:,2),'r.'); % xy points of the end effector
     h.MarkerSize=10;
xlabel('x[meters]');
ylabel('y[meters]');

h=plot(origin(:,1),origin(:,2),'k*'); % xy points of origin
     h.MarkerSize=10;
     h.LineWidth=2;
```



[samples of angles converted into end-effector points plotted in x-y space]

This data was obtained by setting the sample size to the desired value and using the RevoluteForwardKinematic funtion. From this, we can see the best operating area for the arm is the negative-X area, as the endpoint can reach the most locations within its length's reach. The positive-X area is limited as the joints can only work within a 180 degree range, the circular gap made from the arc of the elbow movement.

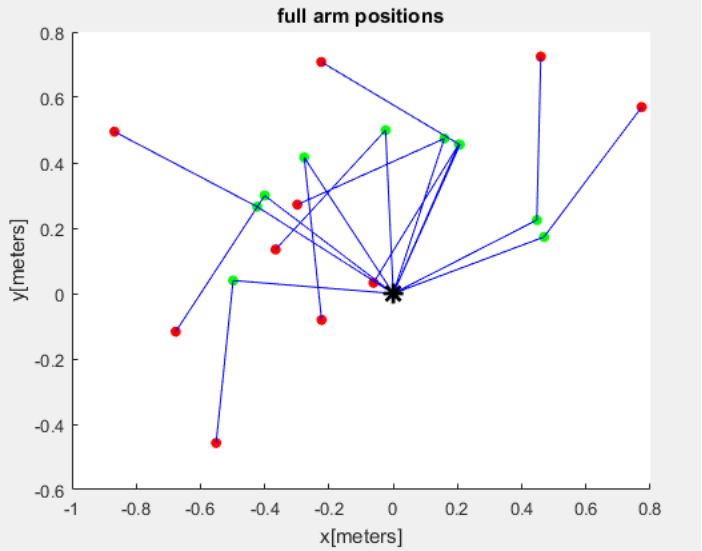
1.3 Configurations of a revolute arm

Through the plot function, we can get a visual approximation of the robot arm by connecting the end-effector, elbow, and origin together with lines.

```
massOrigin = origin(1,1).* ones(samples,2);

XP = [P1(:,1) P2(:,1)];
YP = [P1(:,2) P2(:,2)];


XOP = [massOrigin(:,1) P1(:,1)];
YOP = [massOrigin(:,2) P1(:,2)];
```

Where P1 and P2 are from the `RevoluteForwardKinematics2D` function.

```
figure
hold on
title('full arm positions')
h=plot(P2(:,1),P2(:,2),'r.'); % xy points of the end effector
     h.MarkerSize=20;
h=plot(P1(:,1),P1(:,2),'g.'); % xy points of the elbow
     h.MarkerSize=20;
xlabel('x[meters]');
ylabel('y[meters]');

     for index = 1:samples %for each sample
plot(XP(index,:), YP(index,:), 'b');%plot a line between the end and elbow
plot(XOP(index,:), YOP(index,:), 'b');%plot a line between elbow and origin
     end

     h=plot(origin(:,1),origin(:,2),'k*'); % xy points of the origin
     h.MarkerSize=10;
     h.LineWidth=2;
```



[Arm configurations for 10 data points of angles randomly ranging between 0 and π radians. Enpoints in red, elbow in green, limb connections in blue, origin in black]

2. Analyse a two-layer network

2.1 Lower layer output activation

The first layer output activation comples from sum and sigmoid functions.

a_2 = Sigmoid $(W_1 X_i)$

Sigmoid $(Q) = \frac{1}{1+e^{-Q}}$

Meaning: $a_2 = \frac{1}{1+e^{-W_1 X_i}}$

2.2 Top layer output activation

The end output is a sum function of the previous layer

$$o = W_2 a_2$$

$$o = W_2 \frac{1}{1+e^{-W_1 X_i}}$$

2.3 Network cost function

The squared error of a single point:

$$error = (t_i - o_i)^2$$

For the overall cost for all datapoints:

$$error_c = \sum_{i=1}^{n} (t_i - o_i)^2$$

2.4 Derive the generalised delta rule

The delta rule is the gradient of the cost function with respect to the weight

For respect to the output-layer weights W_2

$e = (t_i - o_i)^2 \quad o = W_2 a_2$

$$\frac{de}{dW_2} = \frac{d}{dW_2}(t_i - o_i)^2 \qquad \frac{de}{dW_2} = \frac{de}{du}\frac{du}{dW_2} \qquad u = t - o, \; e = u^2$$

$$\frac{de}{du} = 2u \qquad \frac{du}{dW_2} = \frac{d}{dW_2}(t_i - o_i) = \frac{d}{dW_2}(t_i - W_2 a_2) = -a_2$$

$$\frac{de}{dW_2} = 2(t_i - o_i)(-a_2) = 2a_2(o_i - t_i)$$

Cancelling out the 2 with a ½ scaler, the equation can be simplified to:

$$\frac{de}{dW_2} = d_3 a_2{}^T \quad \text{with} \quad d_3 = o - t$$

For respect to the input-layer weights W_1

$e = (t_i - o_i)^2 \quad o = W_2 a_2 \quad a_2 = \frac{1}{1+e^{-W_1 X_i}}$

$\frac{de}{dW_1} = \frac{d}{dW_1}(t_i - o_i)^2 \qquad \frac{de}{dW_1} = \frac{de}{du}\frac{du}{dW_1} \qquad u = t - o\,,\ e = u^2$

$\frac{de}{du} = 2u \quad \frac{du}{dW_1} = \frac{d}{dW_1}(t_i - o_i)$

$\frac{du}{dW_1} = \frac{du}{do}\frac{do}{dW_1} \qquad \frac{du}{do} = -1 \quad \frac{du}{dW_1} = -\frac{do}{dW_1} \quad \frac{de}{dW_1} = -2(t_i - o_i)\frac{do}{dW_1}$

$\frac{do}{dW_1} = \frac{d}{dW_1} W_2 a_2 \quad \frac{do}{dW_1} = \frac{do}{da_2}\frac{da_2}{dW_1} \quad \frac{do}{da_2} = {W_2}^T$

$\frac{da_2}{dW_1} = \frac{1}{1+e^{-W_1 X_i}} \quad \frac{da_2}{dW_1} = \frac{da_2}{du}\frac{du}{dW_1} \qquad u = W_1 X_i\,,\ a_2 = \frac{1}{1+e^{-u}}$

$\frac{du}{dW_1} = {X_i}^T \qquad \frac{da_2}{du} = \frac{d}{du}\left(\frac{1}{1+e^{-u}}\right) = \frac{d}{du}(1+e^{-u})^{-1}$

$\gamma = (1+e^{-u}) \quad \frac{da_2}{du} = \frac{da_2}{d\gamma}\frac{d\gamma}{du} \quad \frac{da_2}{d\gamma} = \frac{d}{d\gamma}\gamma^{-1} = -\gamma^{-2} = -\frac{1}{\gamma^2}$

$\frac{d\gamma}{du} = \frac{d}{du}1 + e^{-u} = -e^{-u} \qquad a_2 = \frac{1}{\gamma}$

$\frac{da_2}{du} = \frac{da_2}{d\gamma}\frac{d\gamma}{du} = (-e^{-u})\left(-\frac{1}{\gamma^2}\right) = \frac{e^{-u}}{\gamma^2} = \frac{\gamma - 1}{\gamma^2}$

$= \frac{1}{\gamma}\left(\frac{\gamma}{\gamma} - \frac{1}{\gamma}\right) = a_2(1-a_2)$

$\frac{de}{dW_1} = -2(t_i - o_i)\frac{do}{dW_1} = -2(t_i - o_i)\frac{do}{da_2}\frac{da_2}{du}\frac{du}{dW_1}$

$= -2(t_i - o_i){W_2}^T a_2(1-a_2){X_i}^T = 2{W_2}^T(o_i - t_i)a_2(1-a_2){X_i}^T$

Cancelling out the 2 with a ½ scaler, the equation can be simplified to:

$\frac{de}{dW_1} = d_2 X^T$ with $d_2 = ({W_2}^T d_3)a_2(1-a_2)$ and $d_3 = o - t$

2.5 Weight Update

$\frac{de}{dW_1}$ is the error gradient. By multiplying the gradient by a learning rate α we can modify the rate to a local minimum by subtracting the modified gradient value

3. Implement a two-layer network

Due to time constraints this is the least-developed part of the coursework, focusing on the more visually confirmable sections.

3.1 Implement network recognition

Due to misunderstanding what is meant, the recogniser and training were done at the same time. However it uses the end weights of training, so the used function to find the joint angles to reach an endpoint Xin is:

```
net2 = WO1*Xin;
A2 = 1./(1+exp(-net2));
aA2 = [A2;1];
OutputX = WO2*aA2;
```

3.2 Implement 2-layer network training

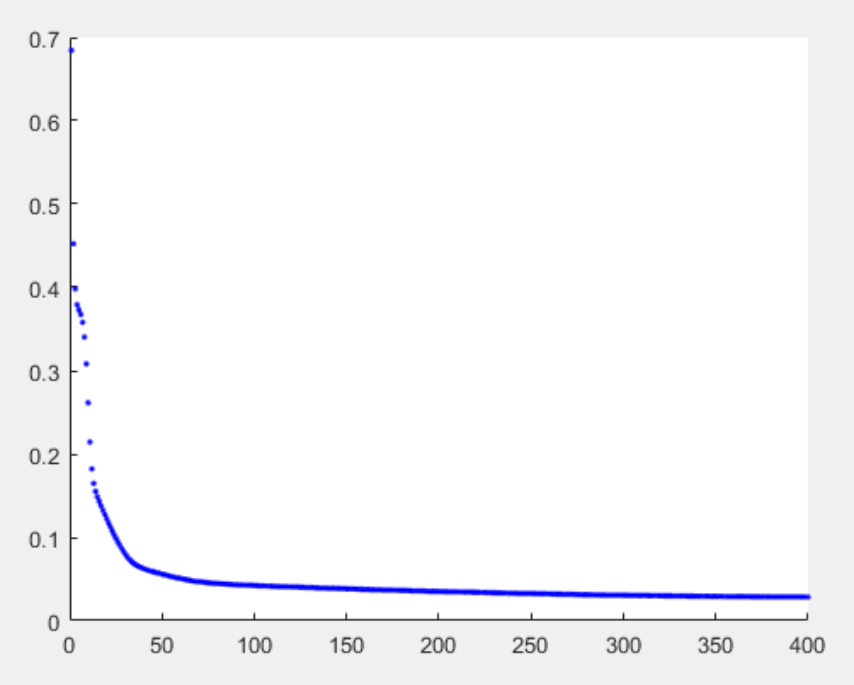
The delta rule is the math behind the learning.

```
testSamples = 1000;
testTheta = (3.14).*rand(testSamples,2);  %radians
[PTest1, PTest2] = RevoluteForwardKinematics2D(armLen, testTheta, origin);

%the point of the network is to get the angle from the endpoint
Target = testTheta';
onerow = ones(1,testSamples);
X = [PTest2'; onerow];
hidden = 10;

W1 = 0.2 * randn(hidden,3);
W2 = 0.2 * randn(2,hidden+1);

Alpha = 0.01;
finish = 400;

[OutputA, WO1, WO2, ErrorOut] = TwoLayerNetwork(X, W1, W2, Target, Alpha,
finish);
```

The inputs for testing were as above, and used below:

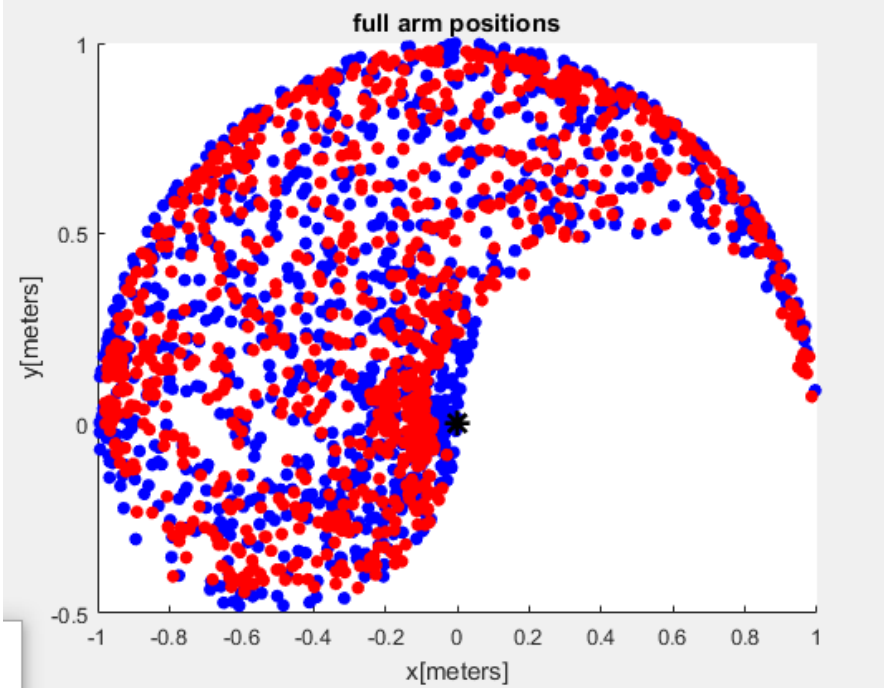
```
function [OutputA, W1, W2, error] = TwoLayerNetwork(X, W1, W2, Target,
Alpha, finish)

%X is input
%W's are weights
%Target is desired output
%Alpha is learning rate

%loop until termination
%a2  1/(1+e^-w1x
%Out = w2*a2
%d3 = 2(Out-Target)
%d2 = w2T*d3*(a2(1-a2))
%de/dw1 = xT*d3
%de/dw2 = a2T*d3
%w1 = w1 - A*de/dw1
%w2 = w2 - A*de/dw2
%end loop


samples = size(X, 2);

for time = 1:finish
for idx = 1:samples


    XV = X(:, idx);
    TV = Target(:, idx);

    net2 = W1*XV;
    A2 = 1./(1+exp(-net2));
    aA2 = [A2;1];
    Output = W2*aA2;

    W2N = W2(:,1:(end-1));

    d3 = 2*(Output-TV);
    d2 = (W2N'*d3).*A2.*(1-A2);

    deltaeW1 = d2*XV';
    deltaeW2  = d3*aA2';

    W1 = W1 - Alpha*deltaeW1;
    W2 = W2 - Alpha*deltaeW2;

    OutputA(:,idx) = Output';

    errorSample(idx,:) = mean((TV-Output).^2);
end
 error(time) = mean(errorSample);
end
```

It uses the error to correct weights. It shares the same weights through all data so that a random input will get a relevant output even if that exact input has not been done before. I have also set it up so that I can view the end output to compare to the target, and so I can see the error.



[Plot of the mean error against time]

Using the neural network output, the kinematic equation can take the output to get the endpoints it derived, and compared it to the target's resultant endpoints:



[Comparison of Target Data endpoints (blue) and Neurel Network angles converted to endpoints (red)]

4. Path planning through a maze

4.1 Random start state

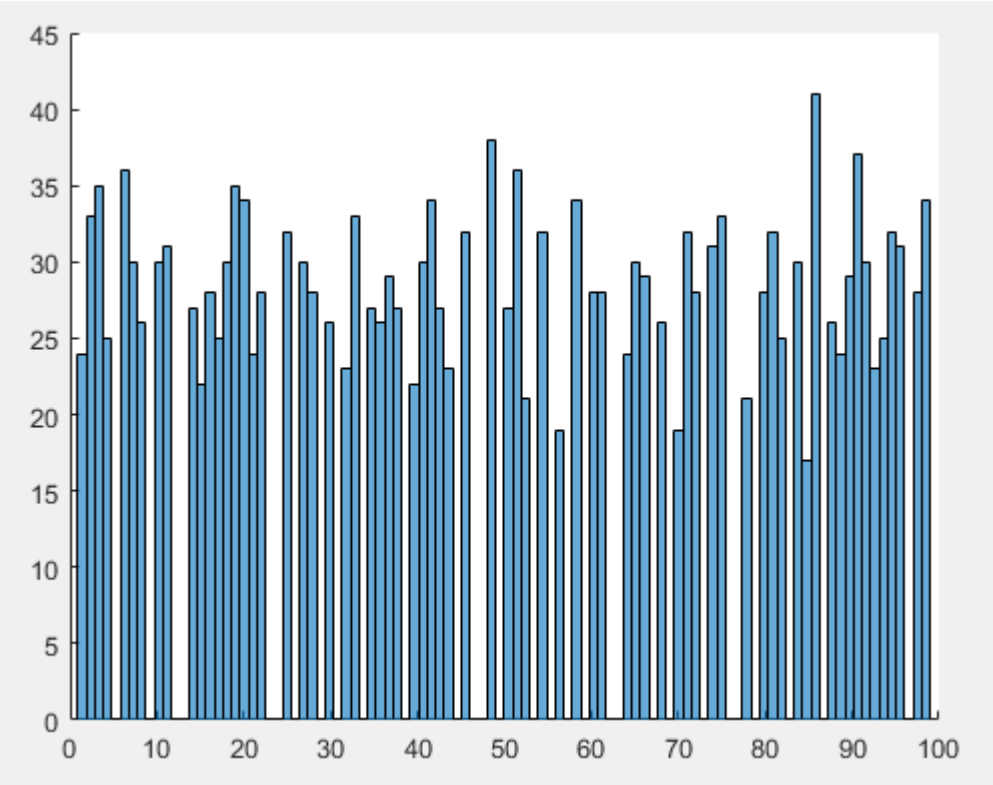
To create a randomised start, the point has to be not blocked by one of the pre-established locations. These include both the blocked states and the end state as that seems pointless.

```
function startingState = RandomStartingState(f)

            Value = 1;
            %xBlock = 0;
            %yBlock = 0;
            while (Value == 1)
                valCheck = 0;
                startingState = randi(100) ;

                d = size(f.blockedLocations);

                for blocked = 1:(d(1))
                    xBlock = f.blockedLocations(blocked,1);
                    yBlock = f.blockedLocations(blocked,2);
                    xyBlocked = xBlock + ((yBlock-1)*10);
                         if (startingState ~= xyBlocked)
                         valCheck = valCheck + 1;
                         end;
                end;
                    xBlock = f.endLocation(1,1);
                    yBlock = f.endLocation(1,2);
                    xyBlocked = xBlock + ((yBlock-1)*10);
                         if (startingState ~= xyBlocked)
                         valCheck = valCheck + 1;
                         end;

                    if valCheck == d(1) +1
                        Value = 0;
                    else Value = 1;
                    end;
            end;
        end
```

By locking it within a while loop, if an incorrect position is selected the process can repeat. It does a ‘for’ loop so that it can go through each blocked location, and uses the size for the loop count so that blocked positions can be added or removed. Due to the blocked locations being inputed as maze co-ordinates, it has to be translated from an x and y co-ordinate to an ID value to compare to the randomly generated start point.

To test the function, a histogram was made by generating 2000 random start points

```
for arrayPoint = 1:2000
RandomValues(arrayPoint) = maze.RandomStartingState;
end

figure
hold on
histogram(RandomValues,100)
```



[histogram of 2000 points divided into 100 bins]

Although some parts of the histogram at this magnification seem inconsistent, when zoomed it it seems as though the columns “drift” from their actual values, an odd wide point forming where the column “snaps” from its furthest to the left to its furthest to the right.

4.2 Build a reward function

The reward function would give the points around it a state-action link that would produce a reward. With a fixed end point, this is simply seeing the viable states around it and setting rewards for if the state-action pair corresponds. However, to make this system modular and so viable for other end points, the state-action pairs had to be dependent of the endpoint mathematically rather than having a massive list of endpoint-surroundingpoint sets.

```
function reward = RewardFunction(f, stateID, action)

            % init to no reward
            reward = 0;

            %find the ID of the end point
            xEnd = f.endLocation(1,1);
                    yEnd = f.endLocation(1,2);
                    endID = xEnd + ((yEnd-1)*10);


            % with ID 100, only two state-action pairs get to the end point
            % however, different endpoint can be chosen, so all rewards
            % are set
            if ((stateID == (endID - 10)) && (action == 1))
                reward = 10;
            end;

            if ((stateID == (endID - 1)) && (action == 2))
                reward = 10;
            end;
            if ((stateID == (endID + 10)) && (action == 3))
                reward = 10;
            end;
            if ((stateID == (endID + 1)) && (action == 4))
                reward = 10;
            end;
        end
```

Through this, a reward is set for each action and the maze ID of the opposite action. For action 1 (move north), the point to the south (ID – 10) is the state that pairs with action 1 for a reward.

4.3 Generate the transition matrix

Similar to the reward function, the transition matrix could be an absurd list for each individual state's possible next states, however that static list would only work if the blocked locations were also static. So instead, the possible next states have to take into account a dynamic block list.

```
function f = BuildTransitionMatrix(f)

        %actions 1234 are NESW
        % allocate
        f.tm = zeros(f.xStateCnt * f.yStateCnt, f.actionCnt);
                    %makes an XY by A matrix (in current setup 100 x 4)

        for StateIDpoint = 1:(f.xStateCnt * f.yStateCnt)

        %step 1: setup a default transition matrix
        %that will have points assume no blockage or wall

            f.tm(StateIDpoint,1) = StateIDpoint + f.xStateCnt;
            f.tm(StateIDpoint,2) = StateIDpoint + 1;
            f.tm(StateIDpoint,3) = StateIDpoint - f.xStateCnt;
            f.tm(StateIDpoint,4) = StateIDpoint - 1;

            %step 2: over-write with blocked positions as self

            %for start
            for ActionMove = 1:4

                d = size(f.blockedLocations);
                for blocked = 1:(d(1))
                    xBlock = f.blockedLocations(blocked,1);
                    yBlock = f.blockedLocations(blocked,2);
                    xyBlocked = xBlock + ((yBlock-1)*10);
                         if (f.tm(StateIDpoint,ActionMove) == xyBlocked)
                             f.tm(StateIDpoint,ActionMove) = StateIDpoint;
                        end;
                end;
            end;
            %for end
```

[code continued on next page]

```
            %step 3: fix end wall locations
            stateXY = IDtoStatePosition(f,StateIDpoint);

            %go north off map (up)
                if ((stateXY(1,2)+1) == 11)
                    f.tm(StateIDpoint,1) = StateIDpoint;
                end;
            %go east off map (right)
                if ((stateXY(1,1)+1) == 11)
                    f.tm(StateIDpoint,2) = StateIDpoint;
                end;
            %go south off map (down)
                if ((stateXY(1,2)-1) == 0)
                    f.tm(StateIDpoint,3) = StateIDpoint;
                end;
            %go west off map (left)
                if ((stateXY(1,1)-1) == 0)
                    f.tm(StateIDpoint,4) = StateIDpoint;
                end;

        end;

    end
```

Step 1 makes multiple assumptions that are disproven later, but by setting up a simplistic math system that applies to a majority of points, it fills the table with default values that would have the outer edge either loop from left to right, or fall out of bounds. Step 2 then eliminates the blocked locations by stating IF the state-action goes to a blocked location, the state-action instead keeps it in the current state. Step 3 then fixes the edge boundary problems. It checks if the state's XY values either exceed the 10x10 grid to 11 or fall off at 0 and sets those transitions to the current state.

4.4 Initialize Q-Values

In the main, a suitable range is chosen.

```
minVal = 0.01;
maxVal = 0.1;
maze = maze.InitQTable(minVal, maxVal);
```

While inside the maze function, that range is used to generate a random selection within that range

```
    function f = InitQTable(f, minVal, maxVal)
                    % allocate
            f.QValues = zeros(f.xStateCnt * f.yStateCnt, f.actionCnt);

            range = maxVal - minVal;
            mean = (minVal+maxVal)/2;
y = range * (rand((f.xStateCnt * f.yStateCnt),f.actionCnt)-0.5) + mean;
            f.QValues =y;
    end
```

4.5 Implement Q-learning algorithm

The Q-learning runs in multiple layers, a single episode unable to refine the q-table, while a trial of multiple episodes reduces steps taken from hundreds to low tens. For an experiment, multiple trials are done.

```
alpha = 0.2;
gamma = 0.9;
explorationRate = 0.1;
episodes = 1000;
trials = 10;
startingPoint = maze.stateStartID;
endPoint = maze.

[TrialEpisodeSteps, Q] = Experiment(maze, qTable, startingPoint, endPoint,
alpha, gamma, trials, episodes, explorationRate);
```

These values are carried throughout the layers, with the initial qTable being updated with the end fully updates and more reliable version being the output Q

```
function [stepsAcrossTrials, Q] = Experiment(MazeIn, Q, startingPoint,
EndPoint, alpha, gamma, trialCnt, episodeCnt, explorationRate)

for trials = 1:trialCnt
       [stepsAcrossTrials(:,trials), Q] = Trial(MazeIn, Q, startingPoint,
EndPoint, alpha, gamma, episodeCnt, explorationRate);

end
```

```
function [steps, Q] = Trial(MazeIn, Q, startingPoint, EndPoint, alpha,
gamma, episodes, explorationRate)

% set termination state
xEnd = EndPoint(1,1);
yEnd = EndPoint(1,2);
   xyEnd = xEnd + ((yEnd-1)*10);
terminationState = xyEnd;

% trial function that runs episodes.
for tidx = 1:episodes
% run an episode and record steps
[Q, steps(tidx)] = Episode(MazeIn, Q, startingPoint, alpha, gamma,
explorationRate,terminationState);
end

end
```

[In hindsight, having the endpoint be translated within the trial stage isn't the best course of action. The maze function itself has been modified to take in the XY input of the endpoint and blocked states:

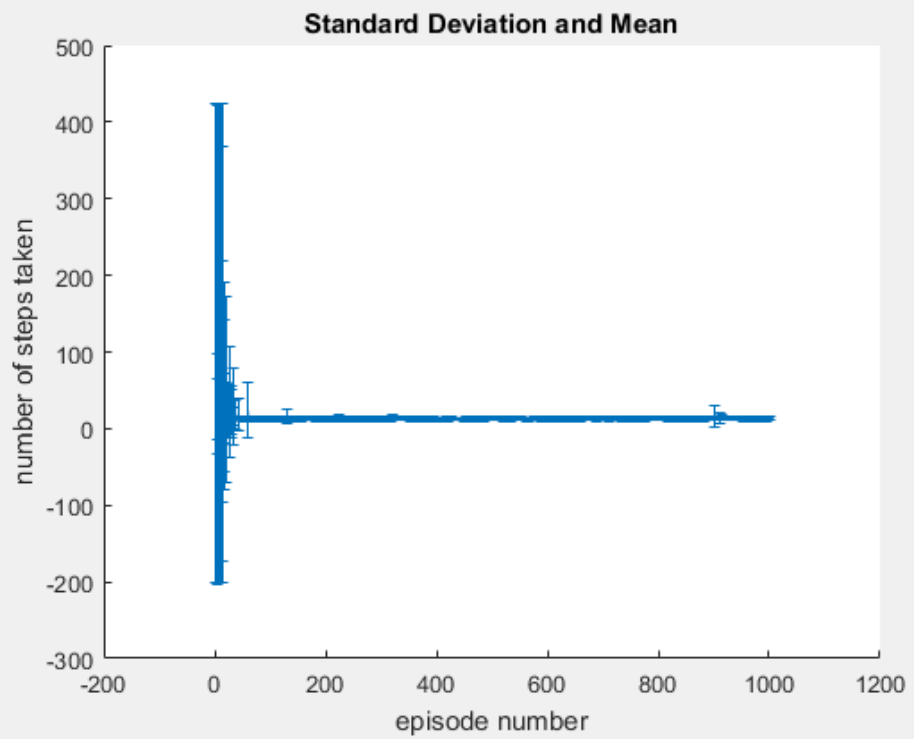
```
maze = CMazeMaze10x10(limits,blockPosition, endPoint);
```

But at the time of writing, it is only a minor inefficiency that can be left untouched with no critical danger.]

```
function [Q, steps] = Episode(MazeIn, Q, startingPoint, alpha, gamma,
explorationRate,terminationState)
% implements a Q-learning episode
% initialize state to a random starting state at the start of each episode
state = startingPoint;
% loop that runs until the goal state is reached
running=1;
steps=0;

while(running==1)
% your code here in code step
action = GreedyActionSelection(Q, state, explorationRate);

% get the next state due to that action
nextState = MazeIn.tm(state,action);

% get the reward from the action on the current state
reward = MazeIn.RewardFunction(state, action);

% update the Q- table
Q = UpdateQ(Q, state, action, nextState, reward, alpha, gamma);

% termination if reaches goal state
if (nextState == terminationState);
    running = 0;
end;

% update steps
steps = steps +1;
% update the state
state = nextState;

if (steps == 1000);
    running = 0;
end;
end
```

When the Q-table is used in the first few trials, the step count can get absurb, so a limit of 1000 is built into the episode function, curtailing the start. It then goes through the standard order for a Q movement with exploration, with action, reward, and update stages to the function loop.

4.6 Run Q-learning



[Error bar plot of standard deviation and mean]

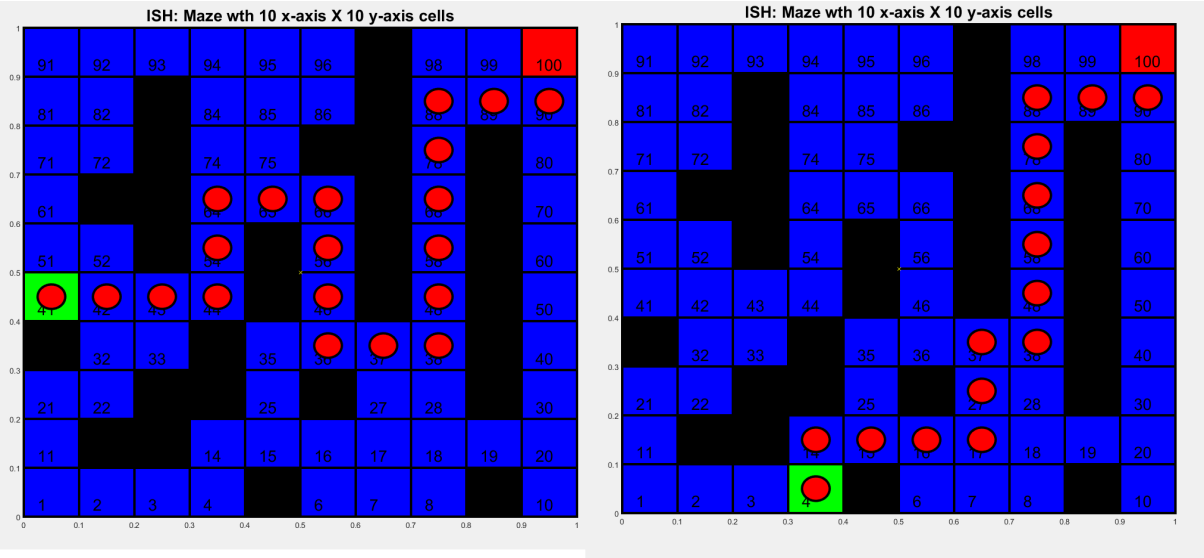
Early on, the error is extremely eratic, but the effect narrows as it progresses. The thickness of the line is due to exploration, as the blocked locations can cause the movement to take certain paths, which may lead to less efficient dirrecitons, such as looping under an obstacle rather than over, wasting steps going down then up rather than just going upward.

4.7 Exploitation of Q-Values

By setting the exploration rate to 0, we can follow the current best q-value path from startpoint to endpoint. This can then be seen by plotting it upon the maze grid.

```
state = startingPoint;
% loop that runs until the goal state is reached
running=1;
steps=0;

while(running==1)
% your code here in code step
action = GreedyActionSelection(Q, state, 0);

% get the next state due to that action
nextState = maze.tm(state,action);

% get the reward from the action on the current state

% termination if reaches goal state
if (nextState == terminationState);
    running = 0;
end;

% update steps
steps = steps +1;


%plotting on maze
StepPlot(steps) = state;
XYstate = maze.IDtoStatePosition(state);
xy = maze.cursorCentre(XYstate', :);
maze.DrawCircle([xy(1,1) xy(2,1) 0.05 0.05], 'r')


%update the state
state = nextState;

if (steps == 400);
    running = 0;
end;
end
```

This makes use of a modified episode, with exploration set to 0 and the reward and update functions removed. StepPlot is an array that holds a 1XN array, containing the N positions travelled through to reach the end point. For safety and bug-testing, it still contains the step limiter, but at a lower 400. With the fully updated Q-Table ‘Q’, it does not run into this error.



[Two random starting points with the path taken plotted using red circles]

Part 5 was not able to be completed in time for the deadline, but it can be theoretically done by scaling the maze and arm to inhabit an effective range, then plotting the end effector to the cursorCentre values. The neural network would then do the inverse kinematics required to get the joint angles and the arm would be moved between each of the positions. At this time, I would not know how to scale the arm, how to have the arm and maze inhabit the same figure, or to transition between animation frames.

If the code needs to be checked to function as my code snippets may not be clear, I have created a public github repository containing the final version of this courseworks files: https://github.com/Retep345/AINT351
When the main is run, it will display the maze with path dots, histogram of random start points, a scatter graph comparing the training data angles (angle 1 by angle 2, in black) against the neural network output (angle 1 by angle 2, in red), the mean error plot, the comparison between target and output endpoints done through the kinematic equation, and the standard deviation and mean for steps taken against episode number.