Faculty of Science and Engineering

Referred Coursework – 2019/20 Academic Year

PLEASE NOTE: If you have been referred in the COURSEWORK element of this module, please complete this referred work.

Module Code: **AINT351**

Module Title: **Machine Learning**

Module Leader: **Dr Ian Howard**

School: **School of Engineering, Computing and Mathematics**

DEADLINE FOR SUBMISSION: Thursday 20th August 2020 at 15:00

SUBMISSION INSTRUCTIONS FOR CANDIDATES
Referred coursework must be submitted electronically using the online submission facility in the DLE by the published deadline. If you have any queries on submission or in relation to the referred work, please contact the Module Leader in the first instance, if they are unavailable please contact the Faculty Office on 01752 584584 immediately so any problems can be rectified.

If you require any part of this publication in larger print, or an alternative format, please contact:

Faculty of Science and Engineering
T: +44 (0) 1752 584584
F: +44 (0) 1752 584540
E: science.engineering@plymouth.ac.uk

Instructions to candidates:

IMPORTANT NOTICE

Submit this report to the DLE by specified referred deadline
You will receive feedback within 20 working days

Please only submit your report for this coursework in PDF format. You must include your student number in a header on every page of the report and also include your student number in the document filename! Please use the report template file if possible.

This coursework should contain a few pages of explanation as required (although it can be more than this is necessary) and a set of images showing the plots requested by the individual parts of the practical exercises, as well as embedded equations and Matlab code that you developed to implement your solutions. You must include your student ID into the title of every figure you generate to prove it was from you.

The report **must** be a stand-alone document. Please embed images in-line in the report.

In order to show video an animation of the final maze movement task, please use Internet links to videos that you have been uploaded to YouTube. Please do not submit more than the single PDF file for coursework 1.

Overview

This coursework consists of four tasks that will result in a simple kinematic controller that will operate a simulated 2-joint revolute arm and getting it to trace a path through a maze.

.

1. First, implement the forward kinematics for the specified 2-joint revolute arm. This involves setting its joint angles and calculating the location of its endpoint.

2. Use an optimization algorithm to find the mapping in the opposite direction - namely from end effector position to arm joint angles. This will enable you to compute the necessary arm joint angles that control the revolute arm, so it generates the desired trajectory of its endpoint

3. You are then provided with a simple 2-dimentional grid representing a maze. You will implement Q-learning to find a good path between a start location and a goal location.

4. Finally use the path trajectory to generate the control angles for the 2-joint arm and generate an animation of it moving between the start and goal states.

- First download and expand the file AINT351Coursework 2019R.zip.

- There is a main Matlab script that provides the first steps you need in maze generation that will be used with your Q-Learning algorithm.

- There is a maze class containing both useful functions to build and draw a maze. It also has several empty Matlab functions that you will also need to fill-in with your code during this assignment.

- There are empty functions to generate the forward and inverse kinematics that you will need to fill-in with your code during this assignment.

- There is also an amply function you will need to fill-in with your code during this assignment to interpolate between points.

Part 1: Forward kinematics

We will now move on to doing something useful with the network: Learning inverse kinematics of a simple robotic arm. However, before we start, we need to state the problem and define the forward kinematics of the arm.

Consider the two-joint revolute arm shown in **Fig. 1.** Given we know the lengths of the links, the endpoint of the arm is determined by the arm's joint angles. The end effector position in (x,y) extrinsic cartesian coordinates is given by:

$$x = l_1 \cos(\theta_1) + l_2 \cos(\theta_1 + \theta_2)$$

$$y = l_1 \sin(\theta_1) + l_2 \sin(\theta_1 + \theta_2)$$

The relationship between the input and angles and the output end effector position is known as the forward kinematics of the arm. Thus, given the input angle we can use the forward kinematics to tell us the endpoint of the arm.

[20 marks]

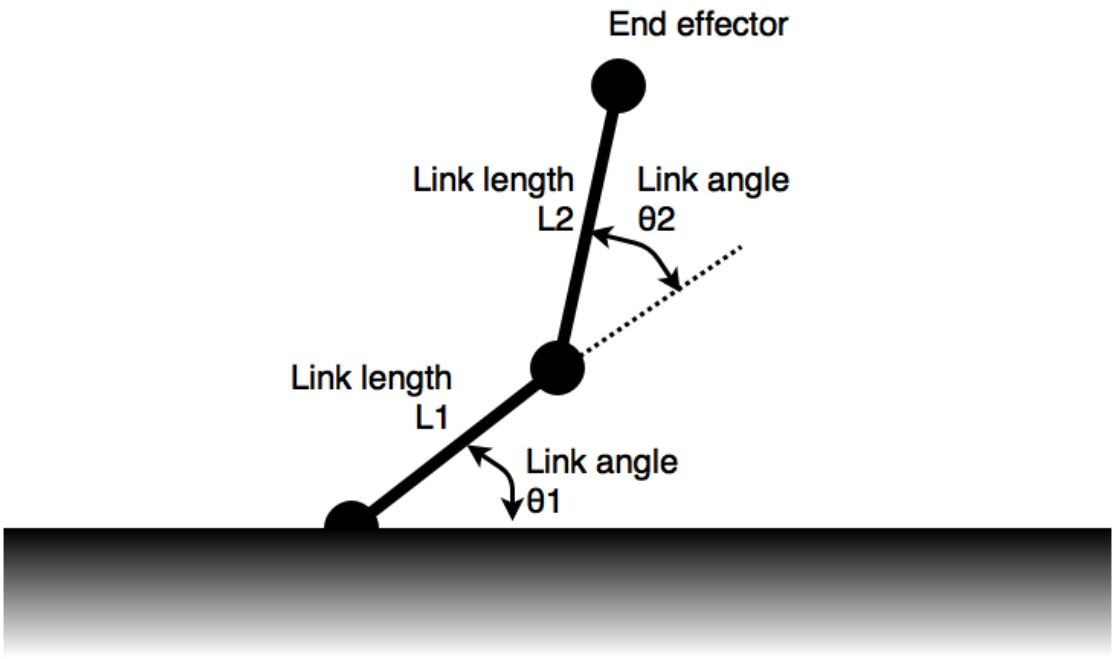

Figure 1. Plan view of a revolute 2-joint arm. The lower link has a length L1 and the upper link length L2. The position of the end effector is affected by both link angles θ_1 and θ_2.

1.1 Implement revolute arm forward kinematics

- Write a Matlab function **RevoluteForwardKinematics2D**, with parameters shown below, that takes arm lengths in the array L, the input angles in the array theta the location of the arm base in the array origin.

- The function returns the location of the end point p2 (x_{p2},y_{p2}) and the elbow joint p2 (x_{p1},y_{p1})

[p1 p2] = **RevoluteForwardKinematics2D**(armLen, theta, origin)

Where the parameters are:

armLen = [L_1 L_2];

theta = [θ_1 θ_2];

origin = [x_o y_o];

p2 = [x_{p2} y_{p2}];

p1 = [x_{p1} y_{p1}];

- Include your full Matlab implementation inline in the report document.

[10 marks]

1.2 Display workspace of revolute arm

- We wish first to show the range of the endpoint when the arm angles move between the limits of 0 to π radians.

- Set the arm lengths to **0.4m** and the arm base origin to (0, 0).

- Generate 1000 uniformly distributed set of training angle data points in the array theta over the range 0 to π radians using the Matlab **rand** command.

- Also, generate a testing dataset of 1000 samples.

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

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

- This should generate a plot like that shown below in **Fig. 2.** (overleaf)

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

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

[5 marks]

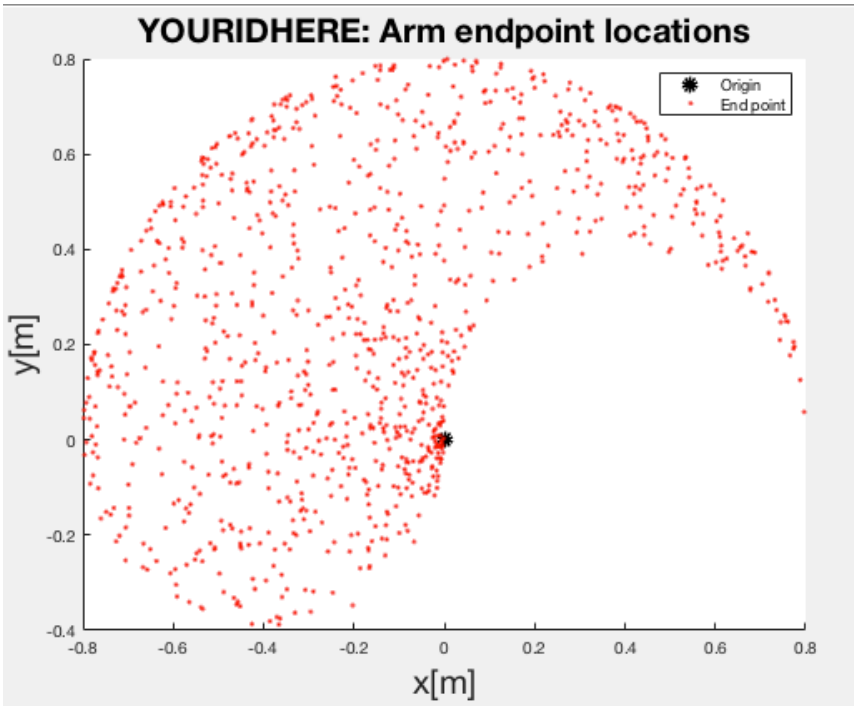

Figure 2. Endpoint locations (red dots) for 1000 data points when the arm angles are randomly distributed between 0 – $\pi\iota$ radians. Origin of the arm also shown.

1.3 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 to π 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.

- This should generate a plot like that shown in **Fig. 3.**

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

[5 marks]

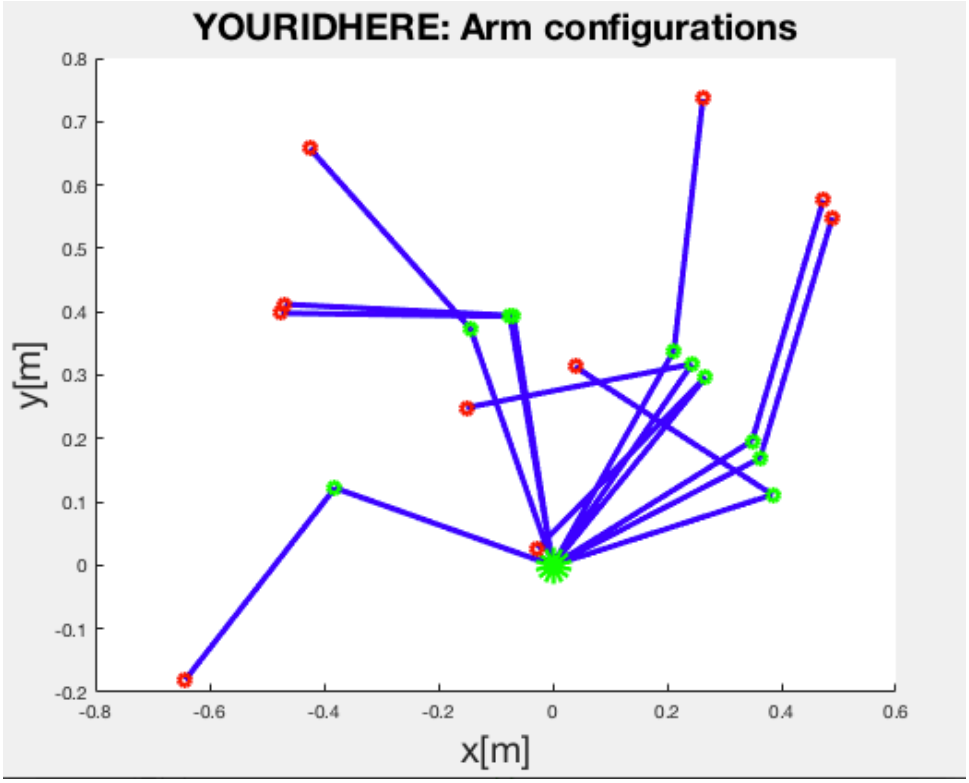

Figure 3. Arm configurations for 10 data points when the arm angles are randomly distributed between 0 – $\pi\iota$ radians. Endpoint locations are red, elbow is green, and the arm is in blue. Origin of the arm also shown (black).

Part 2: Inverse kinematics

You will now implement inverse kinematics by using an optimization algorithm. Thais is, you will supply the endpoint location of the revolute arm and your function will return the joint angles that will achieve this.

The Matlab **fmincon** optimization function makes use of an objective call-back function.

2.1 Build objective function for inverse kinematic optimization

- Write an objective function that takes the two arm joint angles as input and uses forward kinematics to generate an error between the resulting endpoint position and a target position.

- The objective function needs to take as input the desired joint angles, link lengths and target endpoint location.

- It should return an error based on the absolute distance between the computed endpoint location and the target endpoint location.

- Tip: You will need to call your **RevoluteForwardKinematics2D** function in the objective function in order to compute the endpoint location based on the joint angles.

[15 marks]

2.2 Compute inverse kinematics by optimisation

- Write a Matlab function RevoluteInverseKinematics2D that computes the inverse kinematics given an array of input points by running an optimization in Matlab using the fmincon function using the objective function your wrote above.

- Your function is required to return the arm joint angles given the arm endpoint and arm link lengths.

- The exact specification of the inverse kinematics function should be as follows:

function [theta] = RunInverseKinOptimize(armLen, target, origin)

Where the parameters:

- armLen is an array that specified the arm link lengths.
- target is an 2xN array giving the (x,y) coordinates of the of N successive target data points.
- origin specifies the shoulder location.
- theta s an 2xN array holding two joint angles corresponding to the N successive target data positions.

[15 marks]

2.3 Build a test target trajectory

We now wish to test the inverse model more rigorously using a realistic task.

- Write a function **GeneratePointsTrajectory** that takes a list of data points and generates a continuous trajectory by interpolating between these data points. The exact specification of the function should be as follows:

pointsData = GeneratePointsTrajectory(unitSamples, pointList, origin)

Where the parameters:

unitSamples specified the number of samples per unit length between the points
pointList is an 2xN array giving the (x,y) coordinates of the of N successive data points
origin specifies an offset

- Include your full derivation in the report document
- Call the function to generate a set of connected data points that consist of squares with an increasing size. For example, you can use code something like this (which will run).
- This will generate a list of data points.

```
% specify the origin of the trajectory
 origin = [-0.5 0];
 % init the list
 pointList = [];
 % number of samples to interpolate per unit length
 unitSamples=100;
 % points representing a unit square
 unitSquare = [-1 -1; -1 1; 1 1; 1 -1; -1 -1;];

 % loop over a range of square sizes
 for halfSideLen = 0.1 : 0.05 : 0.3
     % scale the unit square
     pointTmp = halfSideLen * unitSquare;
     % append the scaled squares into a trajectory list
     pointList = [pointList; pointTmp;] ;
 end

 % build the trajectories
 pointsData = GeneratePointsTrajectory(unitSamples, pointList, origin);
 % record the number of samples in the trajectory
 samples = length(pointsData);
```

- Plot the (x,y) value you will get something like the following plot **(Fig. 4)**) that is a trajectory centred on the area of the workspace our 2-D revolute arm can reach

[10 marks]

2.4 Testing the inverse kinematics

- Run your inverse kinematics function on the trajectory data to compute the corresponding arm joint angles.

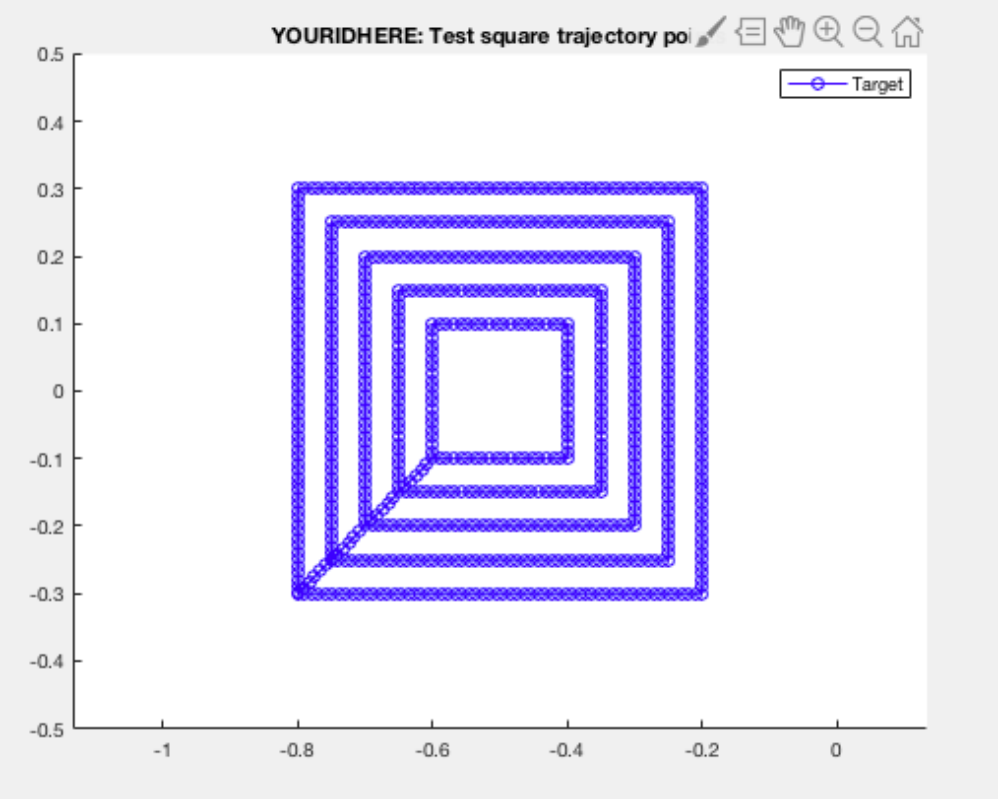

Figure 4. Target trajectory centred on the area of the workspace our 2-D revolute arm can reach:

- Run the **RevoluteForwardKinematics2D** function with the specified parameters and the optimized estimates of joint angles to re-generate the corresponding elbow and endpoint locations.
- Plot the new estimate of end position on top of the target positions used as input to the optimizing inverse model.
- This should lead to very good agreement between the target and estimated values, like that shown in **Fig. 5.** (overleaf)
- What are the issues with using optimization to implement inverse kinematics?

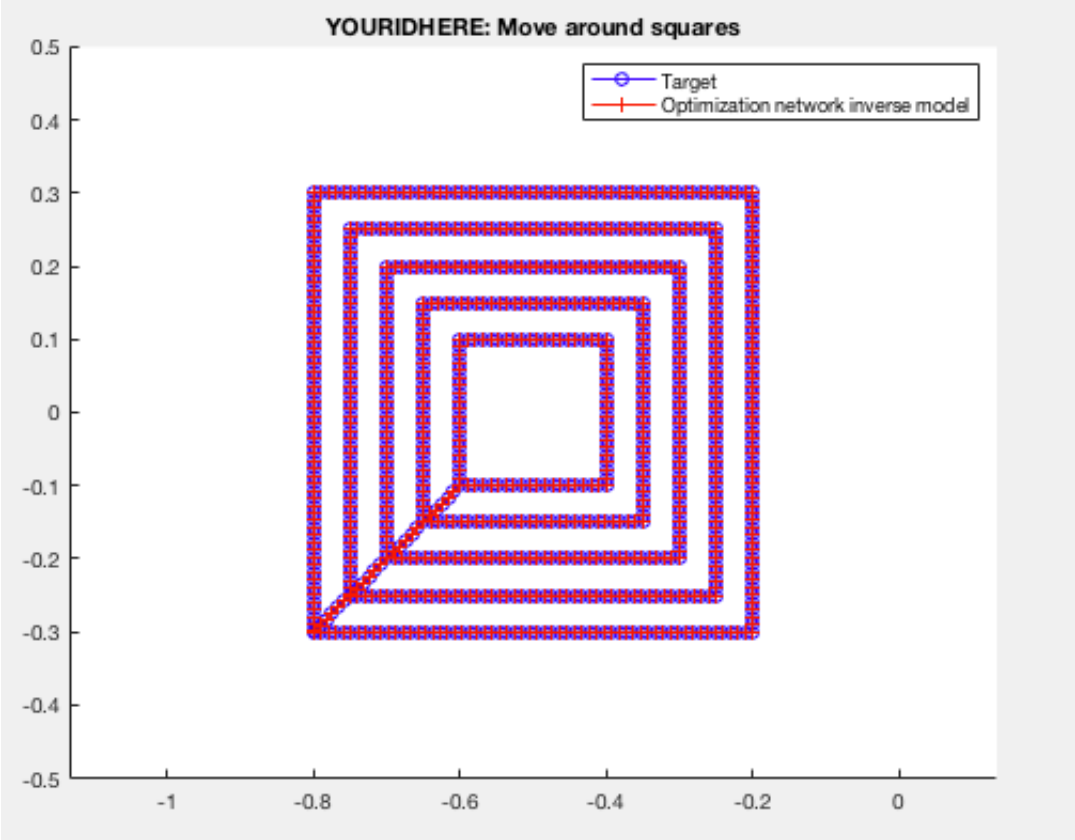

Figure 5. 2-D revolute arm testing trajectory output plotted on top of target trajectory [15 marks]

Part 3: Path planning through a maze

Now we tackle a movement planning problem. You will be required to develop a Q-learning algorithm from first principles that can generate a path through a maze. **You must use the maze layout shown in Fig. 6.** The overall goal of this coursework is then to get the endpoint of the 2-joint revolute arm to move along this maze trajectory.

We will consider the maze shown in **Fig. 6.**

- The states are numbered 1 to 225 as indicated.
- The goal of the Q-learning algorithm is finding the optimal path from a given goal state to the red goal state.
- At the goal 15 there is a reward of 10,
- At the goal 230 there is a reward of 15
- At all other states the reward is 0.
- The green state is one possible start state, that we will make use of later.
- The black cells are blocked states that cannot be entered.
- In general, an action from a state can involve moving north, south, east of west, or staying put if it would lead to a blocked state or edge.

You are now required to implement Q-learning for the maze shown in **Fig. 6.**
Note that during training:

- A step is the execution of a single action of the Q-learning algorithm. That is, from its current state, it executes a greedy action with a small random element, often then moving to a new state, and updating a Q-value.
- An episode starts from a random starting state and runs for many steps until the algorithm reaches a goal state of the maze, at which point the episode terminates.
- A trial starts with naïve Q-values and involves running the Q-learning algorithm for a given number of episodes, updating the same Q-value function across episodes.
- An experiment involves running a given number of trials.

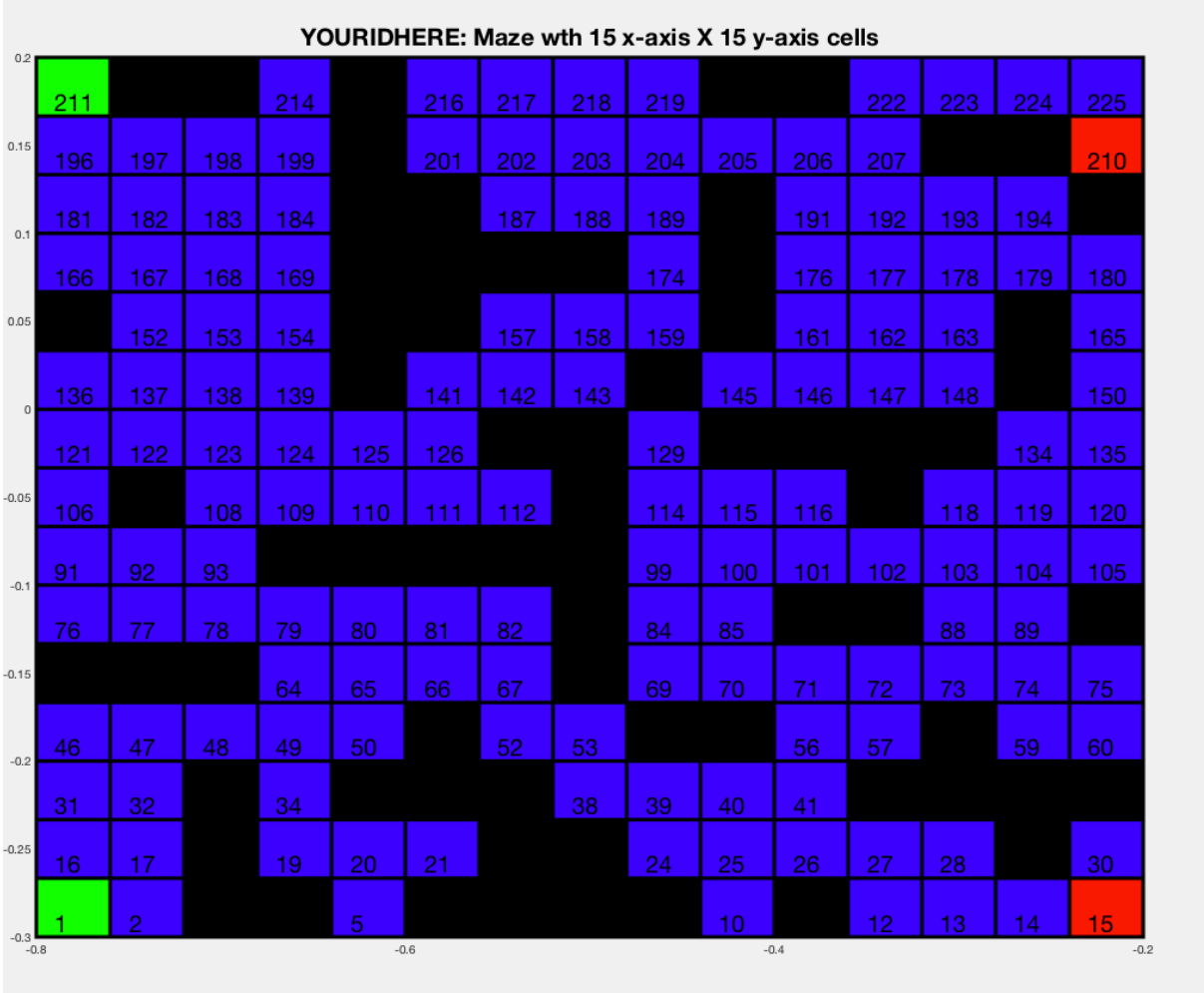

Figure 6. 15x15 grid maze with specified start states shown in green (no reward). The goal states are shown in red. The end state 15 and has a reward of 15 and the state 210 has a reward of 10. The blue states are allowed states with no reward associated with them. The black cells are blocked states and cannot be entered. An action towards a blocked or out of cell state results in the same state the current state.

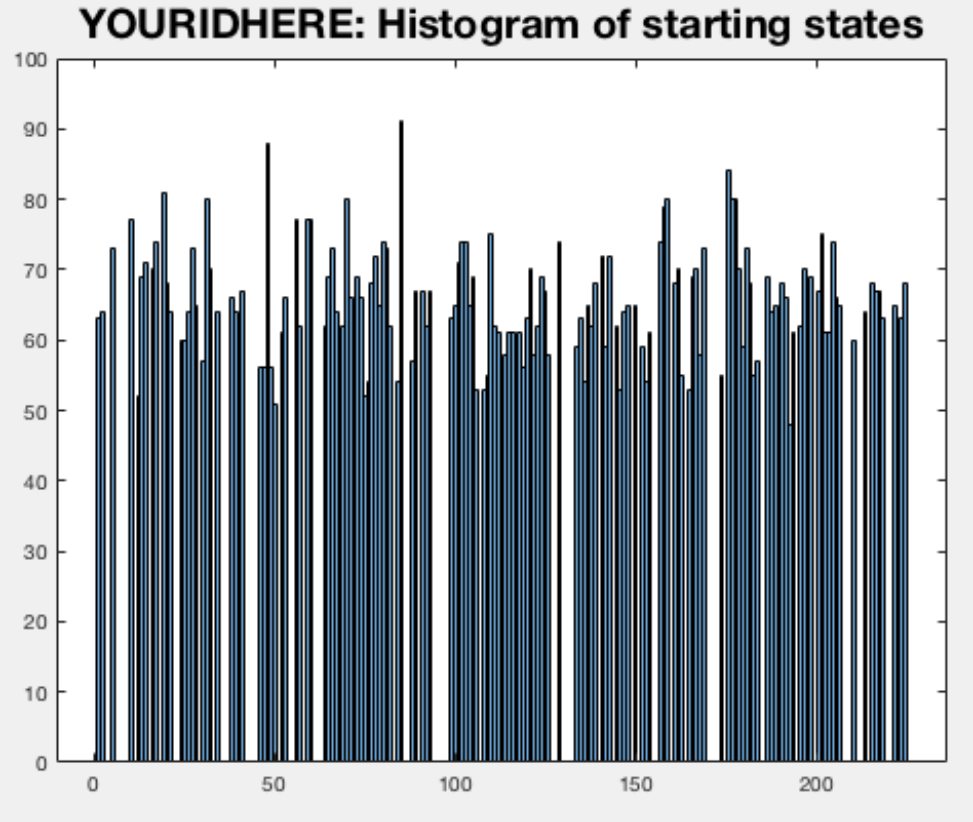

Figure 7. Histogram of 1000 randomly generated starting states, plotted with 225 bins corresponding to one of the maze's 225 states.

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 a goal state.
- Generate 1000 starting states and plot a histogram to check your function is working.
- You should end up with a histogram like the one shown in **Fig 7.**
- Comment on how the displayed state occurrences align with the maze.

[5 marks]

3.2 Implement Q-learning

To implement Q-learning you will also need to:

- Specify a transition matrix for the maze.
- **Tip: It's easy to generate it automatically after you represent the maze appropriately using a matrix.**
- Specify the reward function for the maze
- Initialize the Q-values matrix to sensible numbers
- 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 Q-learning policy is becoming, so it can be used as an indication of algorithm performance on the training data.

[20 marks]

3.3 Run Q-learning

- Run the Q-learning algorithm using:
  An exploration rate of 0.1
  A temporal discount rate gamma of 0.8
  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
- Generate an array containing the means and standard deviations of the number of steps required to complete each episode
- Plot the mean and standard deviation across trials of the steps taken against episodes. Describe what you find. You should end up with a plot like the one shown in **Fig.8.** (overleaf)

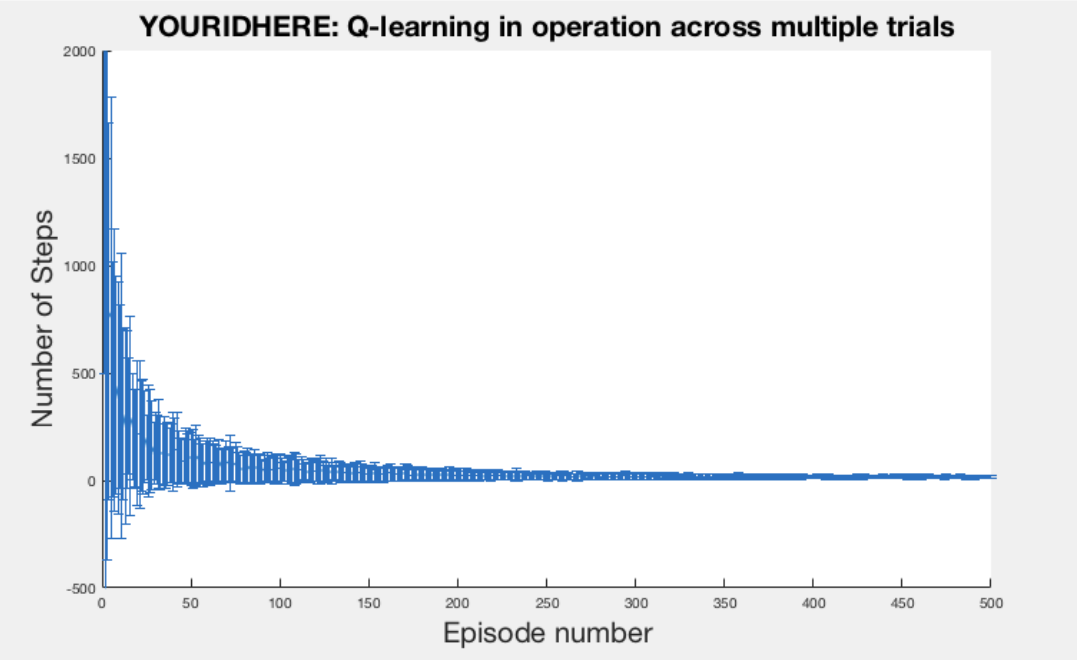

Figure 8. Mean and SD of steps plotted against episode.

We now wish to make use of the learned Q-values to compute the optimal path through the maze for any give stating point.

[15 marks]

3.4 Exploitation of Q-values

- Record the Q-table at the end of a training trial.
- Write an exploitation function that makes use of the Q-values and makes greedy action selection **without exploration.**
- You will need to run two exploitation experiments, one from each of the starting states as the green cells shown on the maze
- For each experiment, record the visited states for this episode.

[15 marks]

3.5 Display greedy actions

- Plot out the greedy optimal action over a drawing of the maze.
- You should end up with a plot is like that shown in **Fig. 9.**

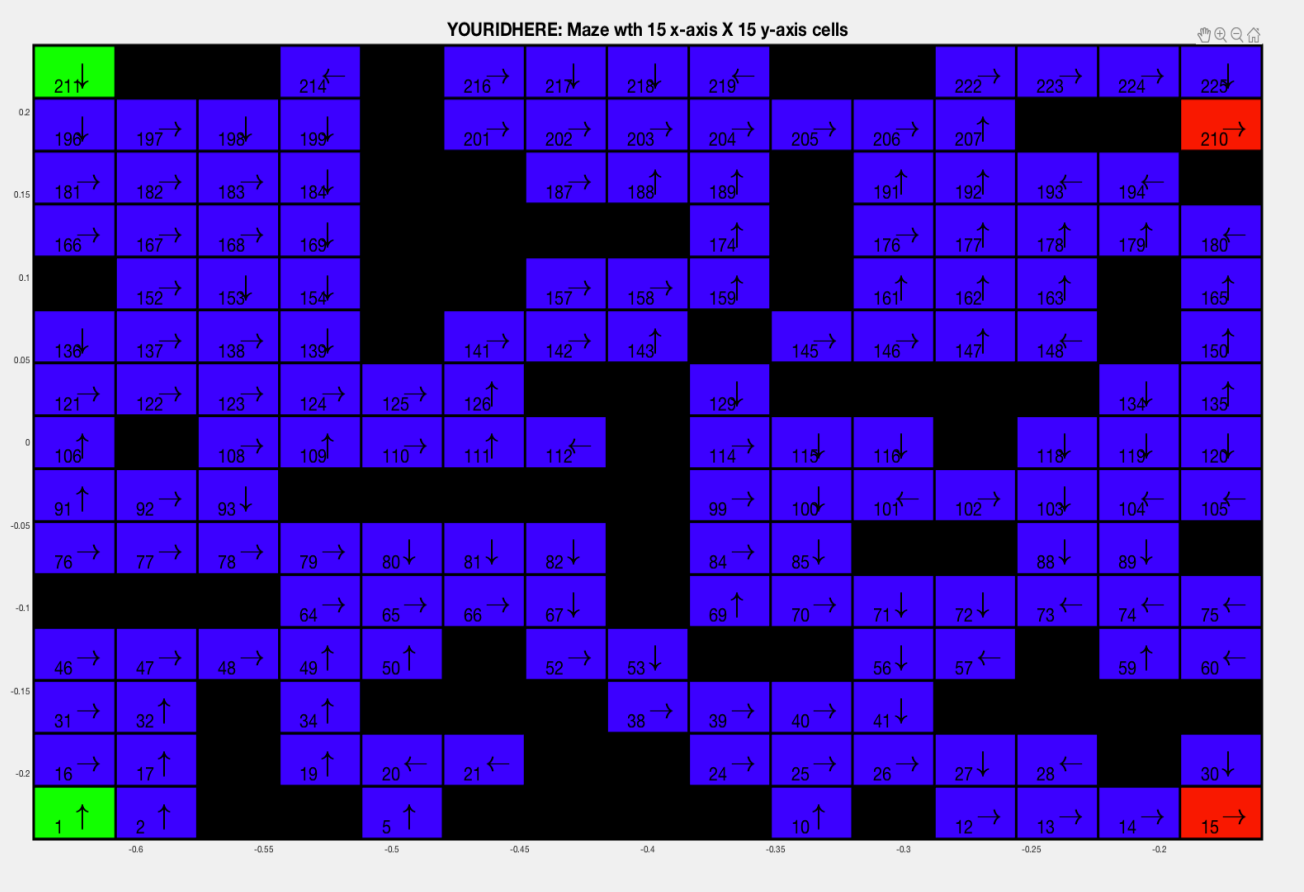

Figure 9. Greedy optimal action after Q-learning

[10 marks]

Part 4: Move arm endpoint through maze

4.1 Calculate arm endpoint movement

- 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.
- For both starting points, plot out the path over a drawing of the maze.
- This should lead to two plots like the one as shown in **Fig. 10** for the state 1 starting position.

[15 marks]

4.2 Animated revolute arm movement

- Finally use the maze paths to specify the endpoint trajectory of the 2-joint revolute arm.
- Tip: you need to scale the maze so that it falls into the workspace of the arm.
- Use the inverse kinematic function to calculate the arm joints.
- Then use the forward kinematic function running with these angles as input to calculate the arm elbow and endpoint positions.
- 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.
- A screenshot of such an animation is shown in **Fig. 11.**

[20 marks]

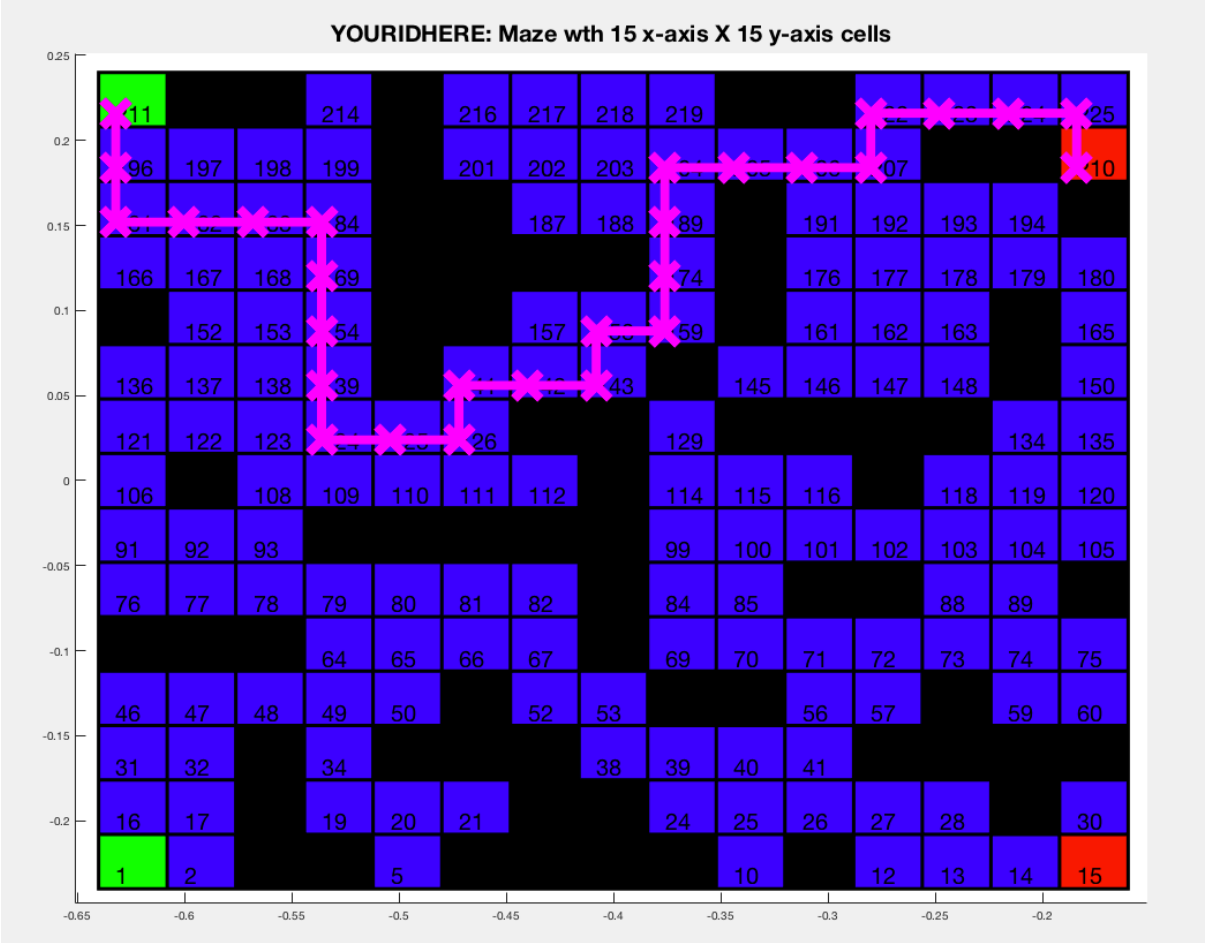

Figure 10. Example path through the maze found by the Q-learning algorithm. This path (shown in magenta starts) from the green cell at state 211 and finally reaches the red destination cell at state 210.

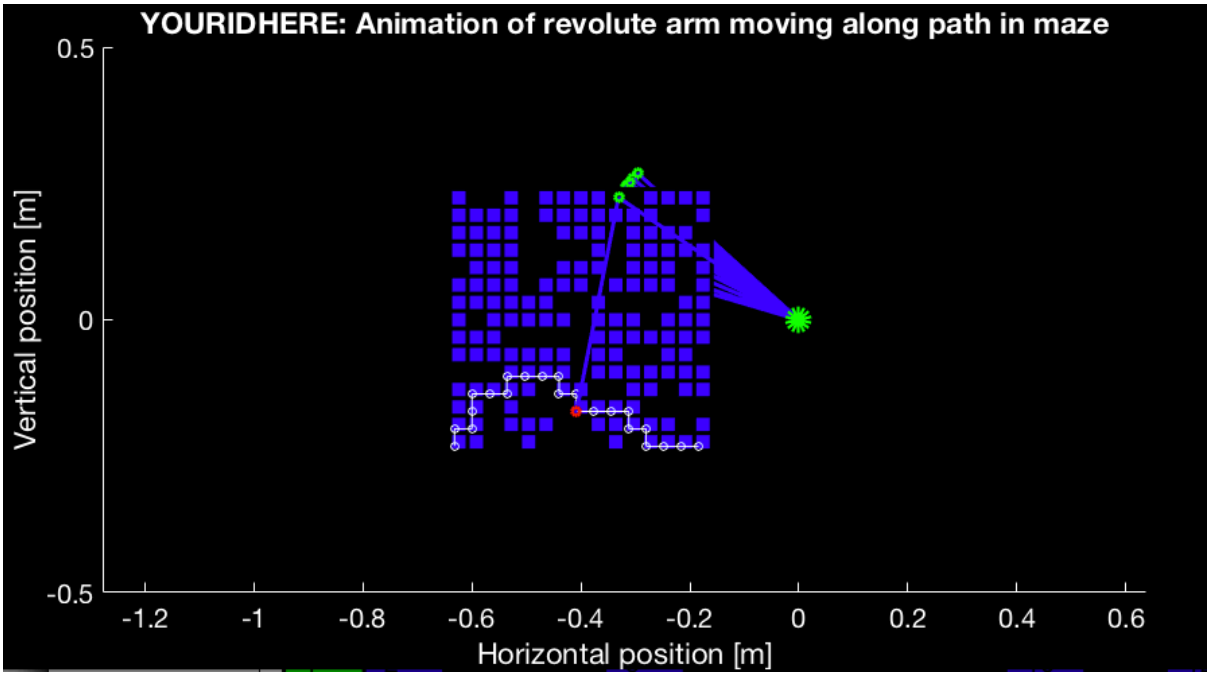

Figure 11. Screenshot of animation of 2-joint arm moving through path on maze. The base of the arm is represented by the large green dot and is located at the coordinates (0,0). The elbow joint is represented by the small green circle. The arm endpoint is given by the red dot and the arm links are shown in blue. Notice that the maze has been scaled to fit into workspace of the robotic arm.