and SECTION 1.

Display workspace of the Revolute Arm

The code in figure# initialises the arm used. It plots arms with length L1, and L2 which are shown above. The two joint angles are randomly generated using the ‘generateUniformDistribution’ shown below. Chart

Description automatically generated with medium confidence These angles are between zero and pi. These angles are used in the provided ‘RevoluteForwardKinematics2D’ function, which calculates and returns the location of the arm’s elbow and end effector, point1 and point 2 respectively. This can be seen below.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Once the data for the arm was generated, the useful range of this arm was outputted using the code above, this can be seen in the figure below

Chart, scatter chart

Description automatically generated

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

SECTION 2

2.1 Implement 2-layer network training

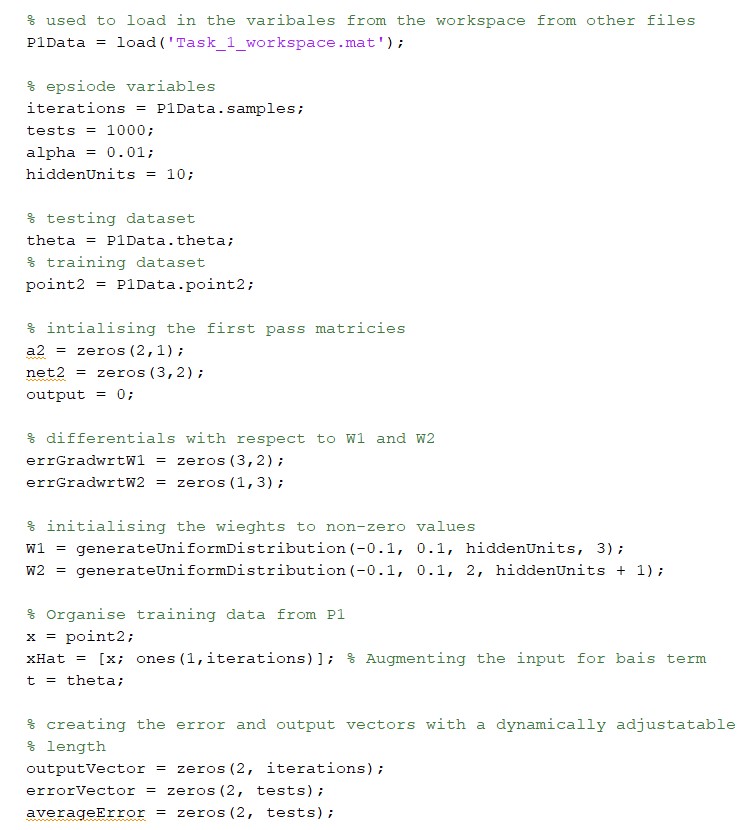
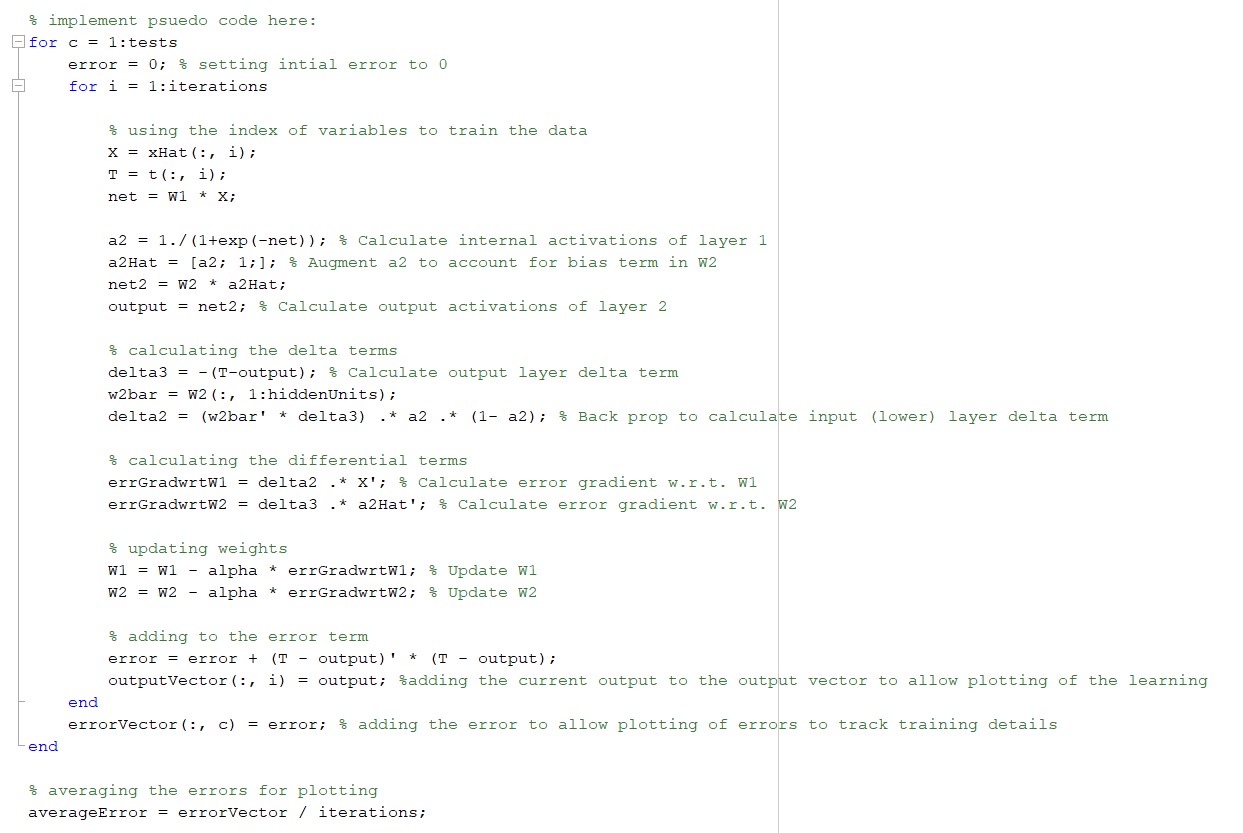


Figure: Setting up and importing constants and variables used for training the network

The weights are initialised using the same ‘generateUniformDistribution’ showing in Figure#. The values are initialised to be -0.1<x<0.1. W1 is setup as a ‘hiddenUnits’x3 matrix, and W2 is setup as a 2x’hiddenUnits + 1’ matrix. The ‘hiddenUnits’ variable has been used so these values can be adjusted easily to test different configurations. It is important for the weights to be close to but not equal to zero, this was cause symmetry in the model that would mean all the inputs to the nodes are zero.



Figure# - 2-layer network with 10 hidden units, and a linear output

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

SECTION 2.2 – Train Network Inverse Kinematics

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

Graphical user interface, text, application, email

Description automatically generated

The corresponding Error over iterations and the output workspace of the arm trained using the network can be seen below

Chart

Description automatically generated with medium confidence

It can be seen from the diagram that at 1000 tests, the output error has dropped to 0.03109. This has been evaluated against other configurations ie. No. of tests and hidden layers, these results are shown below.

SECTION 2.3 – Test and improve the inverse mode

A picture containing scatter chart

Description automatically generated

In figure# the top set of plots show the attainable locations of the arm along with the angles of each of the points used to get to those locations, before the network was trained. The bottom set of these plots show the same information after the network was trained. This figure is significant because it shows that the overall shape of the workspace of the robot has been preserved from the random angles initially used. There are however some differences in the plot of the arm, primarily between [0, -0.2] and [0, 0.2], this could be for several reasons. I believe that it is primarily because of the size of the data set being used to train the model. Ideally the model would be trained with a much larger data set which would give more of a comprehensive set of data to train the model. This could alleviate those areas which contained gaps in the model.

To make a more comprehensive dataset for training the arm, we need to increase the number of samples of angles. To increase the usefulness of the task, and to make the dataset more representative, it would be useful to be able to create more samples inside the larger end of the teardrop. This is where most of the movement of the arm will take place. This would create finer granularity in the areas of the robot’s workspace that would be most often utilised. This could lead to an increased accuracy in the robot’s movement within the maze, which is the useful range of this application.

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

Chart, scatter chart

Description automatically generated

Figure# - No. of hidden nodes changed from 10 to 100

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

Chart

Description automatically generated

Figure# - No. of hidden nodes changed from 10 to 2

Chart

Description automatically generated

Figure – 10000 tests

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

Chart

Description automatically generated

Figure – alpha at 0.0001

Chart

Description automatically generated

Figure – error with alpha of 0.0001

As can be seen from the diagrams above, for this neural network, the biggest influence on the output (for 1000) tests is the value of alpha. With the alpha value of 0.0001, not only is the workspace of the arm affected but so is the error, after 1000 tests. This workspace is much less comparable to the original. The error after 100 tests can also be seen to be over 5 times larger for the same number of tests run.

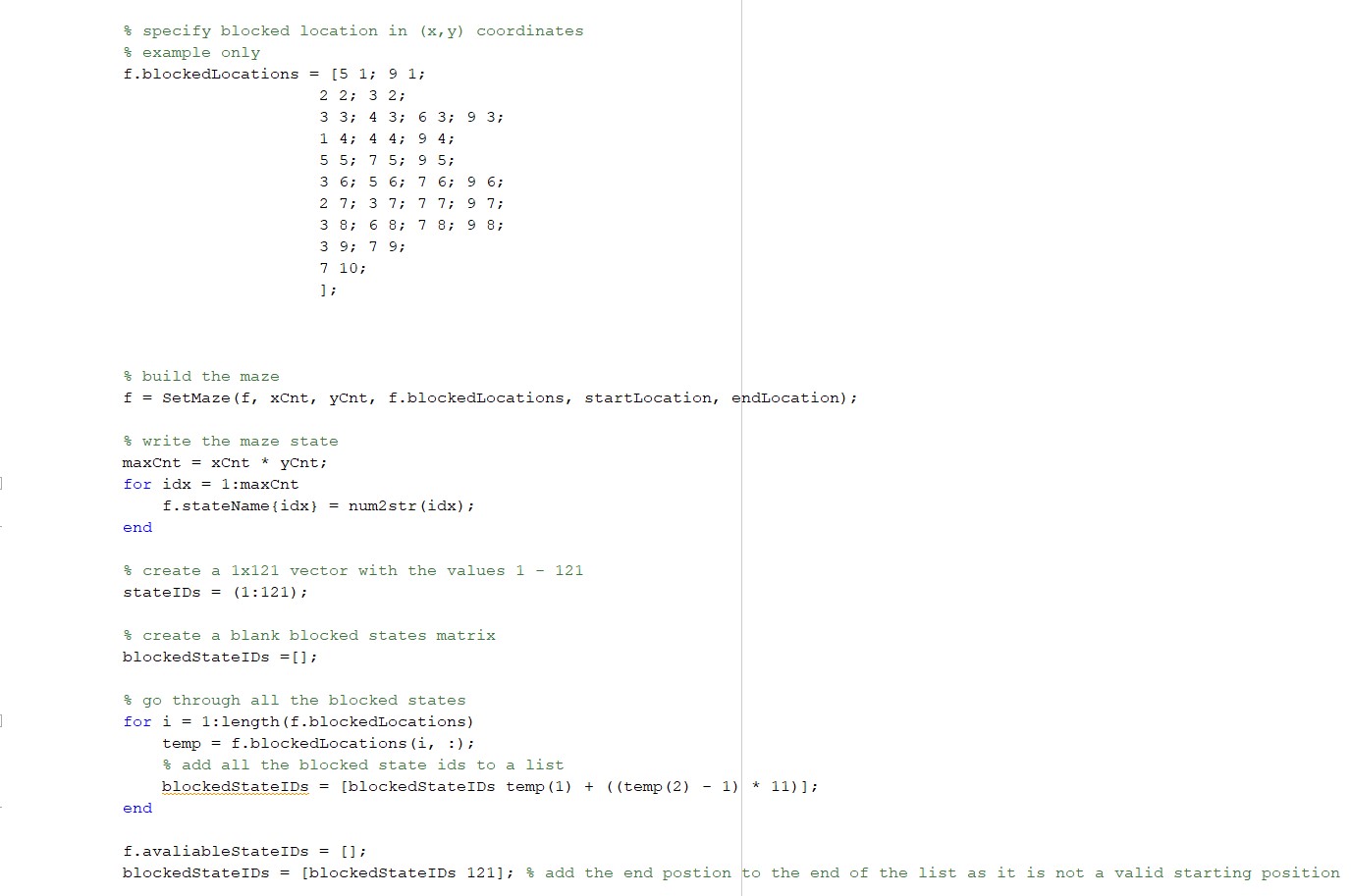
3. Path through the maze

3.1 Random Start State

Text

Description automatically generated

Figure# - Random starting state selected from ‘avaliableStateIDs’ (creation of ‘avaliableStateIDs’ is shown below)



Figure# - Code to create blocked ids and ‘blockedStateIDs’

Text

Description automatically generated

Figure# - Code to create ‘avaliableStateIDs’

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

Description automatically generated

Figure# - Histogram of 1000 generated starting state ids

The histogram in figure~\ref{fig:random\_state\_ids\_histogram} shows 1000 randomly generated values for the maze. Each of the bins in the maze correspond to a state id. There are gaps throughout the histogram. These gaps coincide with blocked or invalid starting state ids. The values 1 and 121 are also blank as these are not valid starting ids for the random starting state selector. The distribution of these values is not uniform, but as the number of random starting state ids increases, the distribution would tend towards a more uniform distribution.

The code to display this histogram is shown below.

Graphical user interface, text, application, email

Description automatically generated

Figure# - Code to generate the starting state histogram

3.2 – Build a reward function

Graphical user interface, text, email

Description automatically generated

Figure# - Reward function that evaluates current state and proposed action to allocate reward

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

3.3 Generate the Transition Matrix

Text

Description automatically generated

Figure# - Function to build the transition matrix from the size of the maze and the available state ids

The transition matrix (tm) is built in the function above. It starts by adding 121 to the available state ids list as for the tm it is a valid state. It then iterates through the length of the empty tm and looks at the current value. If the value is 121, all the values in the tm go back to 121, as it cannot move out that state. The second check is to check if the current id of the tm is a member of the available state ids, if it isn’t then do nothing (the values all stay equal to zero). If the state is an available state id, then the function checks if it can move north, east, south, or west and updates the tm as appropriate. If one or more of the north, east, south, or west moves cannot be achieved, the tm puts back in the value of the state id for the relative movement. This can be seen with the comments in the tm function code. This is done for all values in the tm.

3.4 Initialise Q-Values

Text

Description automatically generated with medium confidence

Figure# - Function to initialise the Q-Values

Graphical user interface

Description automatically generated with low confidence

Figure# - Q-Table initialiser values

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

3.5 Implement Q-Learning Algorithm

Text

Description automatically generated

Figure# - Initialisation variables for the Q-Learning

Graphical user interface, text, application, email

Description automatically generated

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

The main Q-learning algorithm runs inside of the ‘Main\_P3\_RunGridworld2021.m’ file. The initial variables are setup as shown in figure~\ref{fig:q-learning} above. The purpose of this main algorithm is to update the Q-Values, which are based on a multitude of different variables. These include the \epsilon-greedy function, outlined in figure~ref{fig:e-greedy\_function}, the reward function, the learning rate alpha, and gamma. The while loop keeps the Q-learning updating the Q-Values and traversing the maze until the goal state is reached. Once this occurs, the Q-learning will exit that loop and restart the learning from a new random starting state with the updated Q-Values from the old run. This loop will continue until it has completed the full episode number of loops, continuing to optimise the Q-Values whilst solving the maze.

Text

Description automatically generated

Figure# - E(epsilon)-Greedy Action Selector

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

Text

Description automatically generated with medium confidence

Figure# - Update Q-Table function

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

3.6 Run Q-Learning

Graphical user interface

Description automatically generated with medium confidence

Figure# - Trial loops for Q-learning episodes

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

Chart

Description automatically generated

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

\begin{figure}[H]

\centerline{**\includegraphics**[width=15cm]{Q-learning\_in\_operation\_across\_multiple\_trials}}

\caption{Error plots of mean and standard deviation of the Q-learning over multiple trials}

**\label{fig:error\_plot\_across\_multiple\_trials}**

\end{figure}

Figure~ref{fig: error\_plot\_across\_multiple\_trials} above shows the error plots of the mean and standard deviation of the steps taken by the Q-Learning algorithm against the Episode Number. From Episode 250 onwards the number of steps seems to remain consistent. This shows that the Q-Learning algorithm has likely found the most efficient way to get to the end state from any starting state.

3.7 Exploitation of Q-Values

Text, application, email

Description automatically generated

Figure# - Running Q-Learning with epsilon set to zero

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

A screenshot of a computer

Description automatically generated with low confidence

Figure# - Plot of maze with path from highest Q-Values - 1

A picture containing chart

Description automatically generated

Figure# - Plot of maze with path from highest Q-Values - 2

A picture containing text, indoor

Description automatically generated

Figure# - Plot of maze with path from highest Q-Values – 3

4. Move arm endpoint through maze

4.1 generate Kinematic control to revolute arm

Text

Description automatically generated

Figure# - Setting up maze limits and plotting it in the workspace of the arm

A picture containing chart

Description automatically generated

Figure# - Plot of robot arm workspace encompassing the maze

The figure above show the scaling of the maze is correct. This is shown by the maze falling is within the workspace of the robot arm. This means that the arm will be able to reach all locations of the maze.

Graphical user interface, text, application, email

Description automatically generated

Figure# - Calculating the paths joint angles using a forward pass of the trained neural network

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

Text

Description automatically generated

Figure# - Code to output the arm frames onto the maze

Diagram

Description automatically generated

Figure# - Superimposed Arm locations for each step on the maze

4.2 Animated Revolute Arm Movement

The animation of the arm moving through the maze can be seen here:- <https://youtu.be/_ZtAD1VYlzo>

VIDEO - <https://youtu.be/_ZtAD1VYlzo> – don’t lose that link you twat