Kevin Bradley Dsouza
93157162

Part 2a - Reinforcement Learning using a Look-Up-Table (LUT)

2) Once you have your robot working, measure its learning performance as follows:

a) Draw a graph of a parameter that reflects a measure of progress of learning and comment on the convergence of learning of your robot.

I chose to reduce the state space by doing the following:

1. Quantize the X position of my robot to 8 values
2. Quantize the Y position of my robot to 6 values
3. Quantize the distance to enemy to 4 values
4. Quantize the absolute bearing to enemy to 4 values
5. Choose from 4 actions

The Q value is updated using the following rule

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

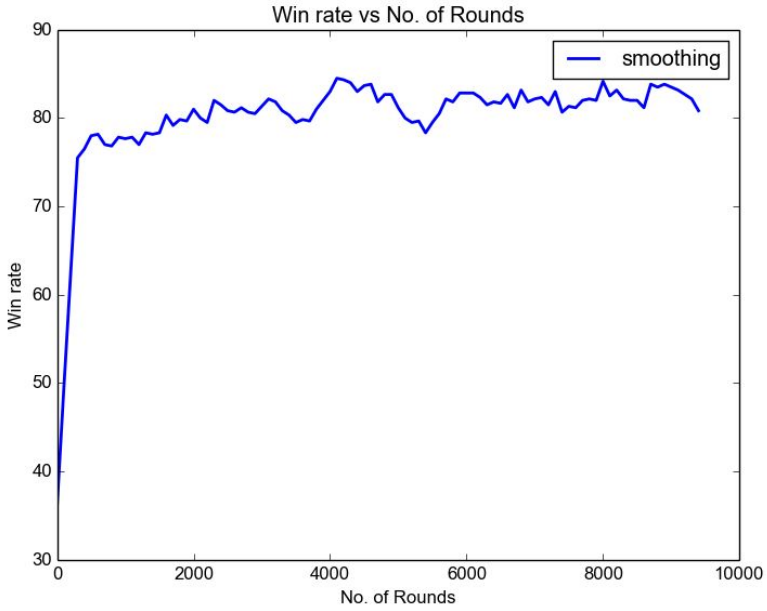
Source:Wikipedia

I chose **winning rate** as the parameter to show the learning progress of my robot. Every Win event is recorded and later the winning rate is calculated for every 100 rounds i.e

$$\text{Win Rate} = \text{Number of Wins per 100 rounds}/100$$

The values of the set parameters are:
Epsilon = 0.2
Alpha (learning rate) = 0.1
Gamma (Discount factor) = 0.9
Number of rounds = 10000

The values of **α** and **γ** are chosen by trial and error after running for multiple rounds. This is the graph showing the plot of winning rate vs number of rounds for the RL robot

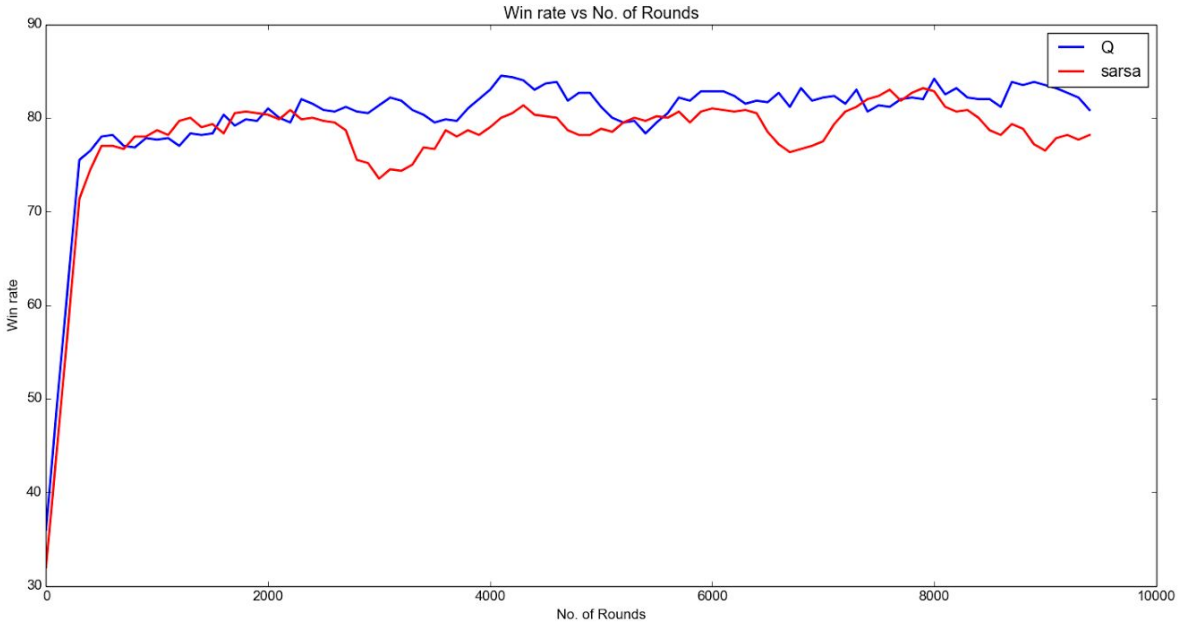


This graph was plotted considering a moving average with a window size of 6 to smoothen out the irregularities. As can be seen from the plot, initially the robot starts off with a poor winning rate. But slowly starts picking up and after about 500 rounds starts to plateau in its performance. It further shows an **increasing trend** with intermediate variations and at the end of 10000 rounds reaches to a **winning rate of 84%**.

b) Using your robot, show a graph comparing the performance of your robot using on-policy learning vs off-policy learning.

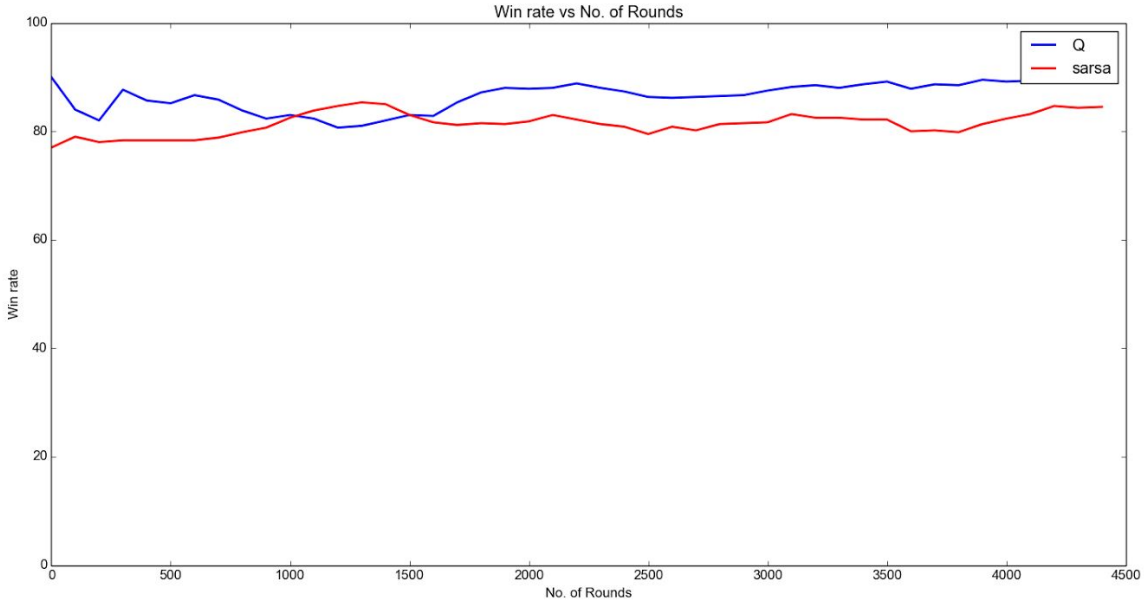
This is a plot showing the learning progress of the on-policy **Sarsa learning** algorithm and the off-policy **Q learning** algorithm.

Again the parameter values are the same as in sub problem a) for both Q and Sarsa learning and the number of rounds is equal to 10000.



As observed from the graph, Q learning learns much better than Sarsa learning for a major portion of the learning time.

Also a plot showing the performance of the two algorithms with the learnt Look Up Table (LUT) and with Epsilon = 0 (greedy) is shown below. All the other parameter values remain the same. The number of rounds is equal to 5000.



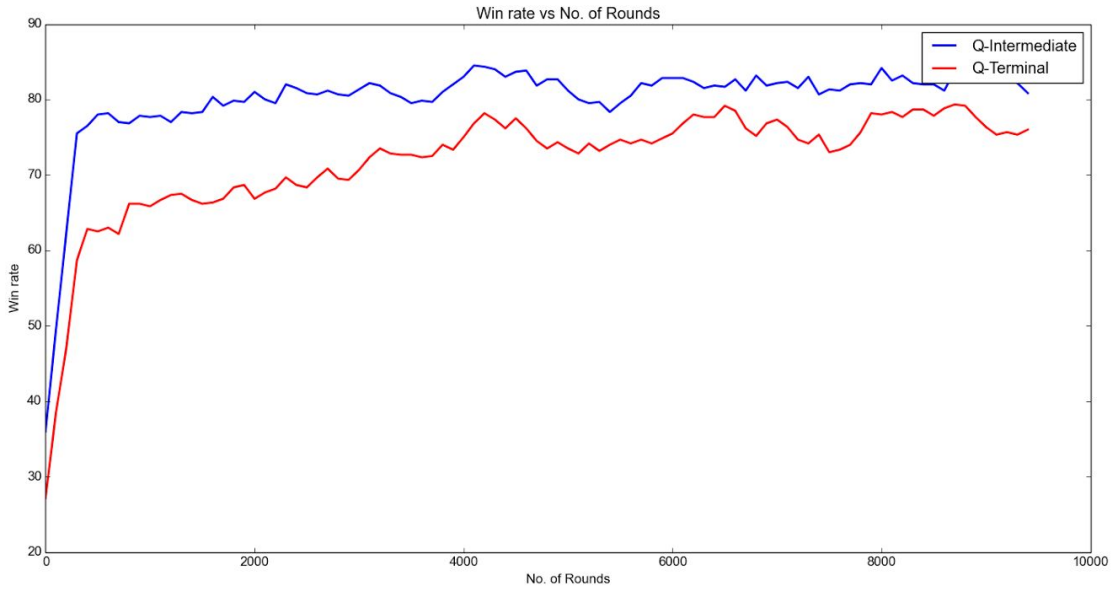
As can be seen, Q learning shows a better performance when compared to Sarsa learning for a major portion of the measured time. This can be attributed to the fact that the Q learning takes the **greedy approach to update its Q value** and is a lot more bold in updating its Q table than the Sarsa algorithm which sticks to the epsilon greedy method and takes a safer approach. The Sarsa algorithm also does sufficiently well as the robot is equipped with multiple actions which are effective by themselves.

c) Implement a version of your robot that assumes only terminal rewards and show & compare its behaviour with one having intermediate rewards.

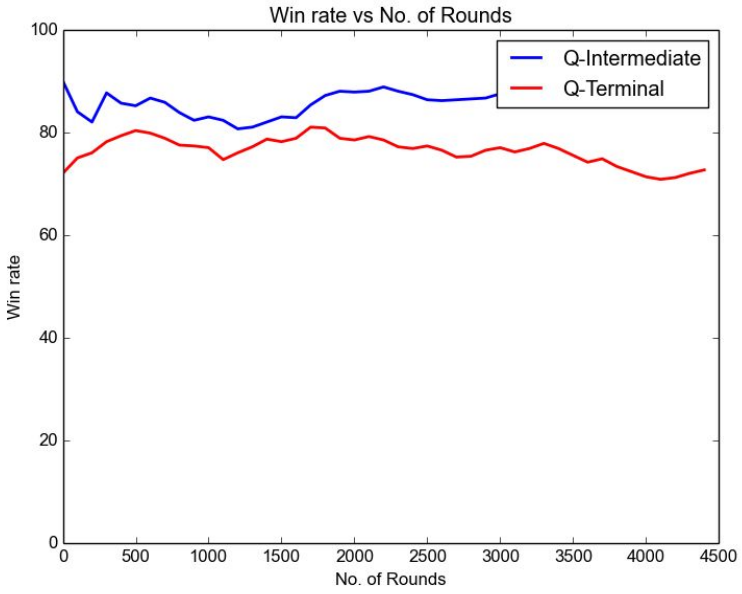
Here the Q learning algorithm is considered for comparison. The Q learning with terminal rewards is modified to generate rewards only on a **Win or a Death event.** No rewards will be given for bullet hitting the opposition and no rewards will be removed on being rammed by the opposition, being hit by a bullet etc.

The following plot shows the learning progress for the two cases, one with Intermediate rewards and final rewards and one with only the final rewards.

Epsilon = 0.2 is used for comparison. All the other parameter values remain the same. Number of rounds is equal to 10000.



It follows from the graph that the intermediate rewards perform much better than the terminal rewards in the case of Robocode. This can be attributed to the credit assignment problem as the credit for the victory/defeat would only be given to the immediate previous state and all the previous bullet hits and rams that might have played a significant role in victory/defeat would be ignored. Also, the LUT will be much less populated with rewards and the robot won't learn effectively during the course of the battle.



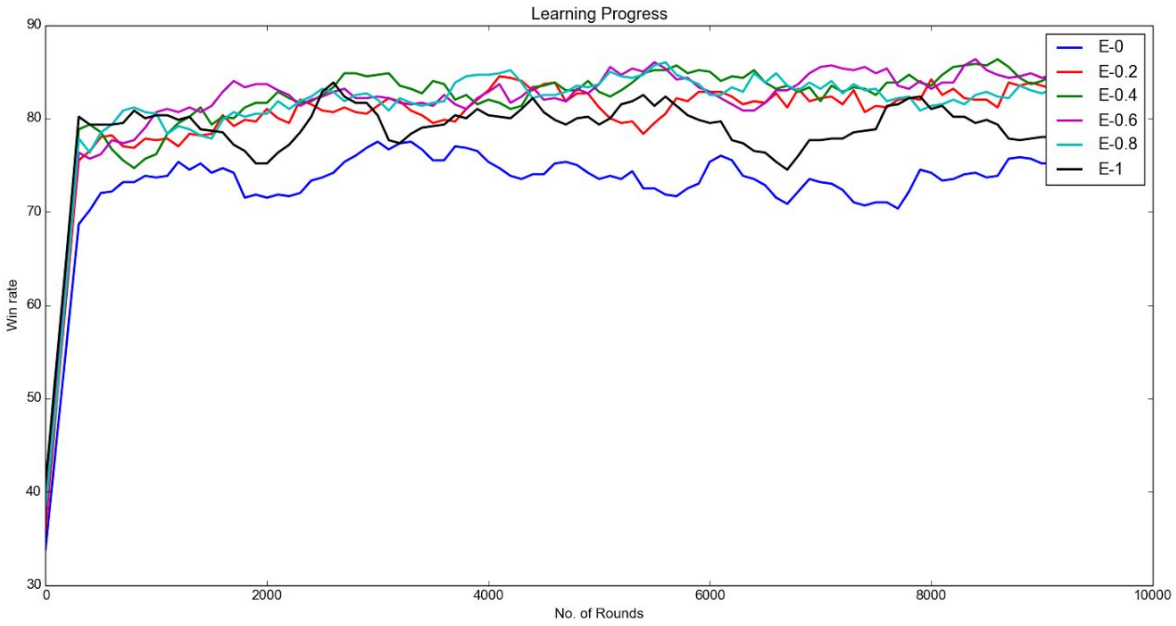
The graph above shows the measured performance of the two robots with the learnt LUT and Epsilon being set to 0. All other parameters stay the same. The number of rounds is set to 5000.

The robot with the intermediate rewards outperforms the one with terminal rewards convincingly. The robot with the terminal rewards does not show much significant improvement from the baseline (with the multiple effective actions).

3) This part is about exploration. While training via RL, the next move is selected randomly with probability ε and greedily with probability 1 − ε

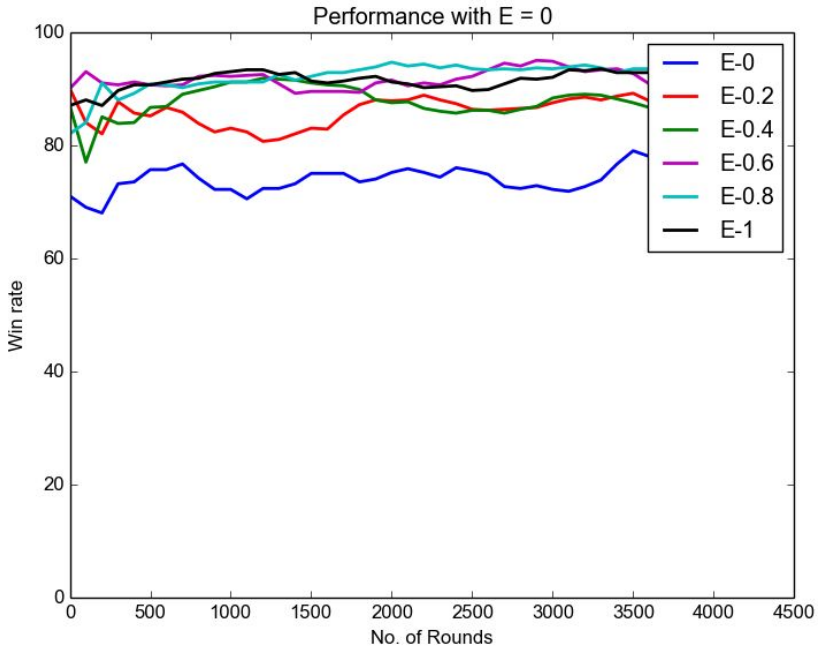
a) Compare training performance using different values of ε including no exploration at all. Provide graphs of the measured performance of your tank vs ε.

Now the Q learning algorithm is trained and tested for different values of Epsilon. Six values of Epsilon - 0 (no exploration,greedy), 0.2, 0.4, 0.6, 0.8, 1 (full exploration) are considered for evaluation. Number of rounds is set to 10000.



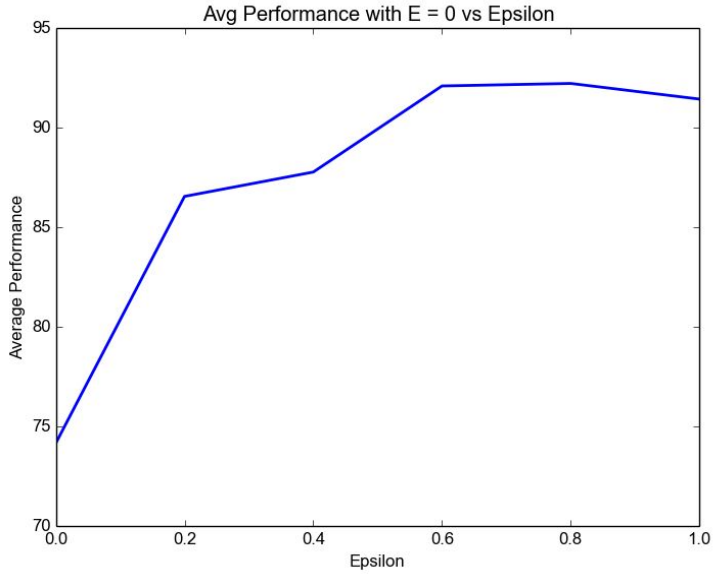
This graph shows that **E = 0.6** and **E = 0.4** have a better learning progress compared to rest of the epsilon values.

Now the performance is measured by setting the epsilon value to 0 and using the learnt LUT's. Number of rounds is set to 5000.



This graph shows that **E = 0.8** dominates for the most part but **E = 0.6** also has a good performance throughout. To sort out the ambiguity, I take the average of the measured performance and plot it vs epsilon.

The following graph shows that the peaking of the performance occurs at **E = 0.6** and stays relatively constant up to **E = 0.8**.



Appendix

Here only the code for the Q learning with intermediate rewards is shown. The other questions are only extensions of this code. For the robot only with terminal rewards, the intermediate rewards are removed. For the sarsa algorithm an additional on policy section is added which will be shown.

```
package sample.Kevin.Try2;

import static robocode.util.Utils.normalRelativeAngleDegrees;
import java.awt.Color;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.PrintStream;
import com.sun.javafx.geom.Point2D;

import robocode.AdvancedRobot;
import robocode.BattleEndedEvent;
import robocode.BulletHitEvent;
import robocode.DeathEvent;
import robocode.HitByBulletEvent;
import robocode.HitRobotEvent;
import robocode.HitWallEvent;
import robocode.RobocodeFileOutputStream;
import robocode.RoundEndedEvent;
import robocode.ScannedRobotEvent;
import robocode.WinEvent;

import java.util.ArrayList;
import java.util.Random;

@SuppressWarnings("unused")
public class Rl_lut extends AdvancedRobot {

	// declare variables
```

```
        final double epsilon = 0;
        final double gamma = 0.9;
        final double alpha = 0.1;

//LUT table initialization
int[] action=new int[4];
int[] actionsAll=new int[4];
int[] statesActionsCombo=new int[8*6*4*4*action.length];
String[][] LUT=new String[statesActionsCombo.length][2];
double[][] doubleLUT=new double[statesActionsCombo.length][2];

//standard robocode parameters
double turnGunValue;
        double bearing;
double absbearing=0;
double distance=0;
private double getVelocity;
        private double getBearing;

//quantized parameters
int quantX=0;
int quantY=0;
int quantDist=0;
int quantBearingAbs=0;

//initialize reward related variables
double reward=0;
String presentQValue=null;
double doublePresentQValue=0;
int actionEpsRandom=0;
int chosenAction = 0;
String stateActionCombo=null;
int saComboInLUT=0;
String stateActionComboNext=null;
        int saComboInLUTNext=0;
        String NextQValue=null;
        double doubleNextQValue=0;
        int actionQGreedy=0;
        int[] matchActions=new int[actionsAll.length];
```

```
        double[] optionsQ=new double[actionsAll.length];

        //counting variables
        int count = 0;
        static int [] winsRate = new int[10000];


public void run(){

        if(getRoundNum() == 0){

                //initialize LUT for learning
                initialiseLUT();
                save();

                //load LUT for testing
                /*try {
                        load();
                        }
                catch (IOException e) {
                        e.printStackTrace();
                        }
                save();*/
   }

        count+=1;
        try {
                load();
                }
        catch (IOException e) {
                e.printStackTrace();
                }

        //set colour
   setColors(null, new Color(200,0,192), new Color(0,192,192), Color.black, new
Color(0, 0, 0));
   setBodyColor(new java.awt.Color(192,100,100,100));

   while(true){
```

```
        //draw a random value for E greedy
        Random rand = new Random();
        double epsilonCheck = rand.nextDouble();

    // load everytime you enter the while loop
    try {
                load();
                }
        catch (IOException e) {
                e.printStackTrace();
                }

    //turn gun to scan
        turnGunRight(360);

        //random action with prob = epsilon
        if (epsilonCheck <= epsilon) {
                actionEpsRandom=intRandom(1,actionsAll.length);

stateActionCombo=quantX+""+quantY+""+quantDist+""+quantBearingAbs+""+action
EpsRandom;
        }

        //greedy with prob = 1 - epsilon
        else if (epsilonCheck > epsilon) {
                stateActionCombo = Qpolicy();
        } // back outside

        /*--------------common code--------------*/

        //find the match for the state action combo in LUT
        for(int i=0;i<LUT.length;i++){
                if(LUT[i][0].equals(stateActionCombo))
                {
                        saComboInLUT=i;
                        break;
                }
        }
```

```
//measure the Q value in the present state before executing action
presentQValue = LUT[saComboInLUT][1];
doublePresentQValue=Double.parseDouble(presentQValue);
//reset reward to 0
reward=0;

/*---------common code ends------------*/

//take random action
if (epsilonCheck <= epsilon) {
actionSet(actionEpsRandom);
chosenAction  = actionEpsRandom;
}

//take greedy action
else if (epsilonCheck > epsilon) {
actionSet(actionQGreedy);
chosenAction = actionQGreedy;
}

/*----------------common code----------------*/

//scan again
turnGunRight(360);

//off-policy Q learning (update state with greedy policy) (This part changes for
Sarsa)
stateActionComboNext = Qpolicy();

//find the match for the state action combo in LUT
for(int i=0;i<LUT.length;i++){
                if(LUT[i][0].equals(stateActionComboNext))
                {
                        saComboInLUTNext=i;
                        break;
                }
}
```

```
        //find the next Q value (greedy)
        NextQValue = LUT[saComboInLUTNext][1];
        doubleNextQValue=Double.parseDouble(NextQValue);

        //update the present Q value according to Q learning equation

doublePresentQValue=doublePresentQValue+alpha*(reward+gamma*doubleNextQVal
ue-doublePresentQValue);
        LUT[saComboInLUT][1]=Double.toString(doublePresentQValue);

        //save the LUT before exiting the while loop
        save();

}//while loop ends

}//run function ends

//function to support greedy policy
public String Qpolicy()
{
            // find action that gives maximum Q value
                        for(int j=1;j<=actionsAll.length;j++)
                        {

stateActionCombo=quantX+""+quantY+""+quantDist+""+quantBearingAbs+""+j;

                                    for(int i=0;i<LUT.length;i++){
                                            if(LUT[i][0].equals(stateActionCombo))
                                            {
                                                    matchActions[j-1]=i;
                                                    break;

                                            }
                                    }

                        }

                        //convert LUT to double
                        for(int i=0;i<statesActionsCombo.length;i++){
```

```
                    for(int j=0;j<2;j++){
                doubleLUT[i][j]= Double.valueOf(LUT[i][j]).doubleValue();
                    }
                }

                //get options for actions
                for(int k=0;k<actionsAll.length;k++){
                    optionsQ[k]=doubleLUT[matchActions[k]][1];
                }

                //finding action that produces maximum q
                actionQGreedy=getMax(optionsQ)+1;

stateActionCombo=quantX+""+quantY+""+quantDist+""+quantBearingAbs+""+action
QGreedy;

        return stateActionCombo;

}

//Intermediate reward functions: (turn this off for terminal rewards)
public void onHitRobot(HitRobotEvent event){reward-=2;}
public void onBulletHit(BulletHitEvent event){reward+=3;}
public void onHitByBullet(HitByBulletEvent event){reward-=3;}

//function to chose random action
public static int intRandom(int min, int max) {
  Random rand = new Random();
  int randomNum = rand.nextInt((max - min) + 1) + min;
  return randomNum;
}

//function to get the maximum Q value
public static int getMax(double[] array){

        int indc = 0;
        double maxValue = array[0];
        for (int i = 1; i < array.length; i++) {
          if ( array[i] >= maxValue ) {
```

```
            maxValue = array[i];
            indc = i;
          }
        }
        return indc;
}

//function to quantize distance to 4 values
private int quantDistance (double dist) {

        if((dist > 0) && (dist<=250)){
                quantDist=1;
                }
        else if((dist > 250) && (dist<=500)){
                quantDist=2;
                }
        else if((dist > 500) && (dist<=750)){
                quantDist=3;
                }
        else if((dist > 750) && (dist<=1000)){
                quantDist=4;
                }

        return quantDist;
}

//function that return values on scanning the enemy
public void onScannedRobot(ScannedRobotEvent e)
        {
        double getVelocity=e.getVelocity();
        this.getVelocity=getVelocity;
        double getBearing=e.getBearing();
        this.getBearing=getBearing;

        //got from trial and error
        this.turnGunValue = normalRelativeAngleDegrees(e.getBearing() + getHeading()
- getGunHeading() -15);

        //distance to enemy
```

```
        distance = e.getDistance(); //distance to the enemy
        quantDist=quantDistance (distance);

        //fire depending on the quantized distance
        if(quantDist==1){fire(3);}
        if(quantDist==2){fire(2);}
        if(quantDist==3){fire(1);}

        //My robot
        quantX=quantPos(getX());
        quantY=quantPos(getY());

        //Calculate the coordinates of the robot
        double enemyAngle   = e.getBearing();
        double angle = Math.toRadians((getHeading() + enemyAngle   % 360));
        double oppoX = (getX() + Math.sin(angle) * e.getDistance());
        double oppoY = (getY() + Math.cos(angle) * e.getDistance());

        //absolute angle to enemy
        absbearing=getAbsBearing((float) getX(),(float) getY(),(float) oppoX,(float)
oppoY);
        quantBearingAbs=quantAngle(absbearing); //state number 4

}

//find absolute bearing (borrowed from robocode wiki)
double getAbsBearing(float xPos, float yPos, float oppX, float oppY) {
        double xo = oppX-xPos;
        double yo = oppY-yPos;
        double hyp = Point2D.distance(xPos, yPos, oppX, oppY);
        double arcSin = Math.toDegrees(Math.asin(xo / hyp));
        double bearing = 0;

        if (xo > 0 && yo > 0) { // both pos: lower-Left
                bearing = arcSin;
        } else if (xo < 0 && yo > 0) { // x neg, y pos: lower-right
                bearing = 360 + arcSin; // arcsin is negative here, actuall 360 - ang
        } else if (xo > 0 && yo < 0) { // x pos, y neg: upper-left
                bearing = 180 - arcSin;
```

```
        } else if (xo < 0 && yo < 0) { // both neg: upper-right
                bearing = 180 - arcSin; // arcsin is negative here, actually 180 + ang
        }

        return bearing;
}

//function to quantize bearing to 4 values
private int quantAngle(double bearAbs) {

        if((bearAbs > 0) && (bearAbs<=90)){
                quantBearingAbs=1;
                }
        else if((bearAbs > 90) && (bearAbs<=180)){
                quantBearingAbs=2;
                }
        else if((bearAbs > 180) && (bearAbs<=270)){
                quantBearingAbs=3;
                }
        else if((bearAbs > 270) && (bearAbs<=360)){
                quantBearingAbs=4;
                }
        return quantBearingAbs;
}

//function to chose from 4 set of actions
public void actionSet(int x)
                        {
        switch(x){
                case 1:
                        int moveDirection=+1;
                        setTurnRight(getBearing + 90);
                        setAhead(150 * moveDirection);
                        break;
                case 2:
                        int moveDirection1=-1;
                        setTurnRight(getBearing + 90);
                        setAhead(150 * moveDirection1);
                        break;
```

```
                case 3:
                        setTurnGunRight(turnGunValue);
                        turnRight(getBearing-25);
                        ahead(150);
                        break;
                case 4:
                        setTurnGunRight(turnGunValue);
                        turnRight(getBearing-25);
                        back(150);
                        break;
        }
}

//function to quantize position into 8 values (X) or 6 values (Y)
private int quantPos(double pos) {

        if((pos > 0) && (pos<=100)){
                quantX=1;
                }
        else if((pos > 100) && (pos<=200)){
                quantX=2;
                }
        else if((pos > 200) && (pos<=300)){
                quantX=3;
                }
        else if((pos > 300) && (pos<=400)){
                quantX=4;
                }
        else if((pos > 400) && (pos<=500)){
                quantX=5;
                }
        else if((pos > 500) && (pos<=600)){
                quantX=6;
                }
        else if((pos > 600) && (pos<=700)){
                quantX=7;
                }
        else if((pos > 700) && (pos<=800)){
                quantX=8;
```

```
                }
        return quantX;

}

//lut initialization for learning
public void initialiseLUT() {
   int[] statesActionsCombo=new int[8*6*4*4*action.length];
   LUT=new String[statesActionsCombo.length][2];
   int r=0;
   for(int i=1;i<=8;i++){
        for(int j=1;j<=6;j++){
                for(int k=1;k<=4;k++){
                        for(int l=1;l<=4;l++){
                                for(int m=1;m<=action.length;m++){
                                        LUT[r][0]=i+""+j+""+k+""+l+""+m;
                                        LUT[r][1]="0";
                                        r=r+1;
                                }
                        }
                }
        }
   }
}

//load LUT from file with IO exception
public void load() throws IOException {
        BufferedReader fileReader = new BufferedReader(new
FileReader(getDataFile("LookUpTable.txt")));
        String rowLine = fileReader.readLine();
        try {
    int u=0;
    while (rowLine != null) {
        String splitLine[] = rowLine.split("    ");
        LUT[u][0]=splitLine[0];
        LUT[u][1]=splitLine[1];
        u=u+1;
        rowLine= fileReader.readLine();
    }
```

```
        } catch (IOException e) {
                e.printStackTrace();
        }finally {
                fileReader.close();
        }
}

//save the LUT to txt file
public void save() {
        PrintStream S = null;
        try {
                S = new PrintStream(new
RobocodeFileOutputStream(getDataFile("LookUpTable.txt")));
                for (int i=0;i<LUT.length;i++) {
                        S.println(LUT[i][0]+"    "+LUT[i][1]);
                }
        } catch (IOException e) {
                e.printStackTrace();
        }finally {
                S.flush();
                S.close();
        }
}

//save win rate
public void saveWinRate()
{
 PrintStream w = null;
 try
 {
  w = new PrintStream(new RobocodeFileOutputStream(getDataFile("winsRate.txt")));
  for(int i=0; i<winsRate.length; i++)
          w.println(winsRate[i]);
 }
 catch (IOException e) {
                e.printStackTrace();
        }finally {
                w.flush();
                w.close();
```

```
        }
}


public void onBattleEnded(BattleEndedEvent e) {
        saveWinRate();
        save();
}

public void onDeath(DeathEvent event)
{
        reward += -5;
        winsRate[getRoundNum()] = 0;
}

public void onWin(WinEvent event)
{
        reward += 5;
        winsRate[getRoundNum()] = 1;
}

}//Rl_check class
```

Sarsa Policy

```
stateActionComboNext = sarsaPolicy(epsilonCheck);

//sarsa policy function
public String sarsaPolicy(double epsilonCheck)
{
   if (epsilonCheck <= epsilon) {

                actionEpsRandom=intRandom(1,actionsAll.length);

stateActionComboNext=quantX+""+quantY+""+quantDist+""+quantBearingAbs+""+ac
tionEpsRandom;
        }
   else if (epsilonCheck > epsilon) {
```

```
                stateActionComboNext = Qpolicy();
        }
        return stateActionComboNext;
}
```

Plotting in Python

```
# -*- coding: utf-8 -*-
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.legend_handler import HandlerLine2D
from scipy.interpolate import spline
from pandas import Series
import pandas as pd

winRate = []
numRounds = []

winValueFile =
open("/home/kevin/Robocode%20/robocode/robots/sample/Kevin/Try2/Rl_check_q
_terminal.data/E0.2/winsRate.txt", "r")
winValue =  winValueFile.readlines()
winValue = winValue[0:9999]
rounds = len(winValue)
numRounds = np.arange(0,rounds,100)
print(len(numRounds))

for i in range(0,(rounds/100)+1):
      winValueTemp = winValue[100*(i):100*(i+1)]
         #print(len(winValueTemp))
         winValueTemp = map(lambda s: s.strip(), winValueTemp)
         winValueTemp = [map(int, x) for x in winValueTemp]
         winNum = np.count_nonzero(winValueTemp)
         winRate.append((winNum/100.0)*100)
```

```
print(len(winRate))

def movingaverage(interval, window_size):
    window = np.ones(int(window_size))/float(window_size)
    return np.convolve(interval, window, 'same')

winRateMov = movingaverage(winRate, 6)
winRateMov = winRateMov[0:95]
numRoundsNew = np.arange(0,len(winRateMov)*100,100)

plt.plot(numRoundsNew, winRateMov,c='b',linewidth=2.0,label="smoothing")
plt.plot(numRounds,winRate,c='r',linewidth=2.0,label="without smoothing")
plt.xlabel('No. of Rounds')
plt.ylabel('Win rate')
plt.title('Win rate vs No. of Rounds')
plt.legend()
plt.show()
```