Part_1/a_star_adam_brendan_diffdrive.py

```
## Brendan Neal and Adam Lobo
##ENPM661 Project 3 Phase 2 Part 1 Main Script

##------------------------Importing Libraries-------------------------##
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
import math
import timeit
import queue
from queue import PriorityQueue



##------------Defining Node Class (From Previous Project)-----------------##

class Node():
    #Initializing Function
    def __init__(self, state, parent, move, C2C, TotalCost):
        self.state = state
        self.parent = parent
        self.move = move
        self.C2C = C2C
        self.TotalCost = TotalCost

    #---Methods for this Class---#
    def ReturnState(self): #Returns Node State X and Y
        return self.state

    def ReturnParent(self): #Returns the Parent Node
        return self.parent

    def ReturnParentState(self): #Returns the Parent Node's State
        if self.ReturnParent() is None:
            return None
        return self.ReturnParent().ReturnState()

    def ReturnMove(self): #Returns Move
        return self.move

    def ReturnC2C(self): # Returns C2C
        return self.C2C

    def ReturnTotalCost(self): #Returns the Total Cost
        return self.TotalCost

    def __lt__(self, other): #OOP Definition for Less than. Required for Priority
Queue.
        return self.TotalCost < other.TotalCost

    ##--------------BACKTRACKING FUNCTION Integrated into Class--------##
```

```
    def ReturnPath(self):
        CompletedMoves = [] #Initialize Move Array
        NodePath = [] #Initialize the Node Path
        CurrentNode = self
        #while(CurrentNode.ReturnMove() is not None): #For move that a Node has made
        while(CurrentNode.ReturnMove() != [0,0]):
            CompletedMoves.append(CurrentNode.ReturnMove()) #Append the previous move
            NodePath.append(CurrentNode) #Append Node to Path
            CurrentNode = CurrentNode.ReturnParent() #Backtrack to the Parent before
repeating Process
        NodePath.append(CurrentNode) #Append the starting point after path is derived.
        NodePath.reverse() #Reverse Order to get front to back path
        CompletedMoves.reverse() #Reverse Order to get front to back path

        return CompletedMoves, NodePath

##---------------------Defining Obstacle Space Setup Functions--------------------##
def setup(s, r):

    global arena

    #Colors
    white = (255, 255, 255)
    gray = (177, 177, 177)
    darkGray = (104, 104, 104)

    #Draw Radial Clearance
    for x in range(0, 600):

        for y in range(0, 200):

            if checkClearance(x, y, s, r):
                arena[y, x] = darkGray

    #Draw Obstacle Borders
    for x in range(0, 600):

        for y in range(0, 200):

            if checkBorder(x, y, s):
                arena[y, x] = gray

    #Draw Obstacles
    for x in range(0, 600):

        for y in range(0, 200):

            if checkObstacle(x, y):
                arena[y, x] = white

#Checks to see if a point is within an obstacle
def checkObstacle(x, y):

    #Left Rectangle
    if x >= 150 and x < 165:
```

```
            if y < 200 and y >= 75:
                return True

        #Right Rectangle
        if x >= 250 and x < 265:

            if y < 125 and y >= 0:
                return True

        #Circle
        if (x - 400) * (x - 400) + (y - 110) * (y - 110) <= 50*50:
            return True

        return False

    #Checks to see if a point is within the border of an obstacle
    def checkBorder(x, y, s):

        #Left Rectangle
        if x >= 150 - s and x < 165 + s:

            if y < 200 + s and y >= 75 - s:
                return True

        #Right Rectangle
        if x >= 250 - s and x < 265 + s:

            if y < 125 + s and y >= 0:
                return True

        #Circle
        if (x - 400) * (x - 400) + (y - 110) * (y - 110) <= (50 + s) * (50 + s):
            return True

        return False

    #Checks to see if a point is within radial clearance of a border
    def checkClearance(x, y, s, r):

        rr = r - 1

        if rr == 0:
            return False

        #Left Rectangle
        if x >= 150 - s - rr and x < 165 + s + rr:

            if y < 200 + s + rr and y >= 75 - s - rr:
                return True

        #Right Rectangle
        if x >= 250 - s - rr and x < 265 + s + rr:

            if y < 125 + s + rr and y >= 0:
```

```
            return True

    #Circle
    if (x - 400) * (x - 400) + (y - 110) * (y - 110) <= (50 + s + rr) * (50 + s + rr):
        return True

    return False

#Checks to see if a point is valid (by checking obstacle, border, and clearance, as
well as making sure the point is within arena bounds)
def checkValid(x, y, s, r):

    if checkObstacle(x, y):
        return False

    if checkBorder(x, y, s):
        return False

    if checkClearance(x, y, s, r):
        return False

    if (x < 0 + s + r or x >= 600 - s - r or y < 0 + s + r or y >= 200 -s - r):
#Accounting for Robot size and Clearance on Borders.
        return False

    return True


##---------------------------------Defining my Action
Set-----------------------------------------##

def ReturnPossibleStates(CurrentNodeState, Wheel_RPMS, RobotRadius, ObsClearance,
WheelRad, WheelDist):
    RPM1 = Wheel_RPMS[0]
    RPM2 = Wheel_RPMS[1]
    ActionSet = [[RPM1, RPM1], [RPM2,RPM2],[RPM1, RPM2], [RPM2, RPM1], [0,RPM1],
[RPM1,0], [0,RPM2], [RPM2,0]] #Differential Drive Action Set
    NewNodeStates = [] #Init List of States

    for action in ActionSet: #For each differential drive action
        NewNodeState, Cost = CalcMoveWithCost(CurrentNodeState, action, RobotRadius,
ObsClearance, WheelRad, WheelDist) #Calculate the state and cost
        if NewNodeState is not None:
            NewNodeStates.append([NewNodeState, Cost, action]) #Append Chile Node
States
    return NewNodeStates

##---------------------------------Defining my Cost and NewNodeState
Function---------------------------------------##

def CalcMoveWithCost(CurrentNodeState, WheelAction, RobotRadius, ObsClearance,
WheelRad, WheelDist):
    t = 0
    dt = 0.1
    Curr_Node_X = CurrentNodeState[0] #Grab Current Node X
    Curr_Node_Y = CurrentNodeState[1] #Grad Current Node Y
```

```
    Curr_Node_Theta = np.deg2rad(CurrentNodeState[2]) #Grab Current Node Theta, 
convert to radians.

    MoveCost = 0.0 #Init Cost

    New_Node_X = Curr_Node_X #Set New Node Start Point X
    New_Node_Y = Curr_Node_Y #Set New Node Start Point Y
    New_Node_Theta = Curr_Node_Theta #Set New Node Start Point Theta

    ##----------------Euler Integration to Generate Curvature----------------##
    while t < 1:
        t += dt
        ChangeX = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.cos(New_Node_Theta)*dt
        ChangeY = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.sin(New_Node_Theta)*dt
        ChangeTheta = (WheelRad/WheelDist)*(WheelAction[0]-WheelAction[1])*dt

        # ChangeX/dt
        # Ydot = ChangeY/dt
        # Thetadot = ChangeTheta/dt

        New_Node_X += ChangeX
        New_Node_Y += ChangeY
        New_Node_Theta += ChangeTheta

        MoveCost += np.sqrt((ChangeX)**2 + (ChangeY)**2)

        ##-----------Why CheckValid is inside the loop--------------##
        '''Inside the loop because if we only checked final, the intermediate steps 
would sometimes be in the obstacle space.'''
        if checkValid(New_Node_X, New_Node_Y, ObsClearance, RobotRadius) == False:
            return None, None

    New_Node_Theta = int(np.rad2deg(New_Node_Theta)) #Convert back to Degrees

    ##-----Wrap to -360-360-----##
    if New_Node_Theta >= 360:
        New_Node_Theta = New_Node_Theta - 360
    if New_Node_Theta < -360:
        New_Node_Theta = New_Node_Theta + 360

    return [New_Node_X, New_Node_Y, New_Node_Theta], MoveCost

##--------------------------Defining my Cost to Go 
Calculation--------------------------##
def Calculate_C2G(CurrentNodeState, GoalNodeState):
    C2G = 0.0
    X_Current = CurrentNodeState[0]
    Y_Current = CurrentNodeState[1]
    X_Goal = GoalNodeState[0]
    Y_Goal = GoalNodeState[1]
    if CurrentNodeState is not None:
        C2G = np.sqrt((X_Goal-X_Current)**2 + (Y_Goal- Y_Current)**2) #Euclidian 
Distance Heuristic function
    return C2G
```

```
##----------------------Defining my Compare to Goal
Function---------------------------##

def CompareToGoal(Current_Node_Position, Goal_Node_Position, ErrorThreshold):
    Dist2Goal = (Goal_Node_Position[0] - Current_Node_Position[0])**2 +
(Goal_Node_Position[1] - Current_Node_Position[1])**2 #Euclidian Distance
    if Dist2Goal < ErrorThreshold**2: #Error less than threshold PLUS the angle has to
be equal
        return True
    else:
        return False

##------------------------Defining my Round to Half
Function-------------------------##
''' This function is Required for "Check Visited" Capabilities'''
def Round2Half(number):
    testvalue = np.round(2*number)/2
    if (testvalue == 10):
        testvalue = testvalue - 0.5
    return testvalue

##--------------------------Defining my Check Visited
Function----------------------##
def CheckIfVisited(Current_Node_State, Node_Array, XYThreshold, ThetaThreshold):
    X = Current_Node_State[0]
    Y = Current_Node_State[1]
    Theta = Current_Node_State[2]
    X = int(Round2Half(X)/XYThreshold)
    Y = int(Round2Half(Y)/XYThreshold)
    Theta = int(Round2Half(Theta)/ThetaThreshold)
    if Node_Array[Y,X,Theta] == 1:
        result = True
    else:
        result = False
    return result



##-------------------------Defining my Plotting Functions-------------------------##
'''For Integers'''
def WSColoring(Workspace, Location, Color):
    x,_,_ = Workspace.shape #Get Shape of Workspace
    translation_x = Location[1] #Where in X
    translation_y = Location[0] #Where in Y
    Workspace[translation_x,translation_y,:] = Color #Change the Color to a set Color
    return Workspace

'''For Curves'''
#Relatively the same function as the cost and state function, but with modifications
to just plot.
#Plots Curve from Parent to New State
def PlotCurves(ParentNodeState, WheelAction, WheelRad, WheelDist, Color, RobotRadius,
ObsClearance):
    t = 0
    dt = 0.1
    Curr_Node_X = ParentNodeState[0]
```

```
    Curr_Node_Y = ParentNodeState[1]
    Curr_Node_Theta = np.deg2rad(ParentNodeState[2])

    New_Node_X = Curr_Node_X
    New_Node_Y = Curr_Node_Y
    New_Node_Theta = Curr_Node_Theta

    while t < 1:
        t += dt
        X_Start = New_Node_X
        Y_Start = New_Node_Y
        ChangeX = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.cos(New_Node_Theta)*dt
        ChangeY = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.sin(New_Node_Theta)*dt
        ChangeTheta = (WheelRad/WheelDist)*(WheelAction[0]-WheelAction[1])*dt

        New_Node_X += ChangeX
        New_Node_Y += ChangeY
        New_Node_Theta += ChangeTheta
        if checkValid(New_Node_X, New_Node_Y, ObsClearance, RobotRadius) == True:
            plt.plot([X_Start, New_Node_X], [Y_Start, New_Node_Y], color = Color,
linewidth = 0.75)





##------------------------Defining my GetInitialState
Function---------------------##
def GetInitialState():
    print("Enter Initial Node X, Y, and Theta separated by spaces: ")
    Init_State=[int(x) for x in input().split()]
    return Init_State

##------------------------Defining my GetGoalState
Function------------------------##
def GetGoalState():
    print("Enter Goal Node X and Y, separated by spaces: ")
    Goal_State=[int(x) for x in input().split()]
    return  Goal_State
##-------------------------Defining my Get Robot Radius
Function-------------------##
def GetClearance():
    print("Enter Desired Clearance From Obstacles.")
    Clearance=int(input())
    return  Clearance

##--------------------------Defining my GetWheelRPMS
Function----------------------##
def GetWheelRPM():
    print("Enter Wheel RPMS, 2 Unique, Separated By Spaces")
    WheelRPMS = [int(x) for x in input().split()]
    return  WheelRPMS
```

```
##-----------------------------------"Main"
Script-------------------------------------##

##-------Getting Parameters from Burger TurtleBot Dimensions-------##

WheelRadius = 3.8 #cm
RobotRadius = 17.8 #cm
WheelDistance = 35.4 #cm

##----------------------Arena Setup------------------##

arena = np.zeros((200, 600, 3), dtype = "uint8")
InitState = GetInitialState()
GoalState =GetGoalState()
DesClearance = GetClearance()
WheelRPMS = GetWheelRPM()

#-----Check Valid Initial State-------##
if not checkValid(InitState[0], InitState[1], RobotRadius, DesClearance):
    print("Your initial state is inside an obstacle or outside the workspace. Please
retry.")
    exit()

##----Check Valid Goal State----------##
if not checkValid(GoalState[0], GoalState[1], RobotRadius, DesClearance):
    print("Your goal state is inside an obstacle or outside the workspace. Please
retry.")
    exit()

setup(RobotRadius, DesClearance) #Arena Setup

WSColoring(arena, InitState, (0,255,0)) #Plot Initial State
WSColoring(arena, GoalState, (0,255,0)) #Plot Goal State

plt.imshow(arena, origin='lower') #Show Initial Arena Setup
plt.show()

#Initialize Arena and Thresholds
SizeArenaX = 600
SizeArenaY = 200
ThreshXY = 0.5
ThreshTheta = 30
ThreshGoalState = 3

# Initialize Node Array
node_array = np.array([[[ 0 for k in range(int(360/ThreshTheta))]
                        for j in range(int(SizeArenaX/ThreshXY))]
                        for i in range(int(SizeArenaY/ThreshXY))])

Open_List = PriorityQueue() #Initialize list using priority queue.
traversed_nodes = [] #Traversed nodes is for visualization later.
starting_node_Temp = Node(InitState, None, [0,0], 0, Calculate_C2G(InitState,
GoalState)) #Generate temp starting node based on the initial state given above.
starting_node = Node(InitState, starting_node_Temp, [0,0], 0, Calculate_C2G(InitState,
GoalState)) #Generate starting node based on the initial state given above.
Open_List.put((starting_node.ReturnTotalCost(), starting_node)) #Add to Open List
```

```
GoalReach = False #Initialze Goal Check Variable
Closed_List= np.array([])#Initialize Closed List of nodes. Closed list is based on
node states
starttime = timeit.default_timer() #Start the Timer when serch starts
print("A* Search Starting!!!!")

while not (Open_List.empty()):
    current_node = Open_List.get()[1] #Grab first (lowest cost) item from Priority
Queue.
    #PlotCurves(current_node.ReturnParentState(), current_node.ReturnMove(),
WheelRadius, WheelDistance, 'g', RobotRadius, DesClearance) #Plot Explored States
Green

    traversed_nodes.append(current_node) #Append the explored node (for visualization
later)
    print(current_node.ReturnState(), current_node.ReturnTotalCost()) #Print to show
search is working.
    np.append(Closed_List, current_node.ReturnState()) #Append to Closed List
    goalreachcheck = CompareToGoal(current_node.ReturnState(), GoalState,
ThreshGoalState) #Check if we have reached goal.

    if goalreachcheck: #If we have reached goal node.
        print("Goal Reached!")
        print("Total Cost:", current_node.ReturnTotalCost()) #Print Total Cost


        MovesPath, Path = current_node.ReturnPath() #BackTrack to find path.
        for nodes in Path: #For Each node in ideal path
            PlotCurves(nodes.ReturnParentState(), nodes.ReturnMove(), WheelRadius,
WheelDistance, 'm', RobotRadius, DesClearance)




    else: #If you have NOT reached the goal node
        NewNodeStates_and_Cost = ReturnPossibleStates(current_node.ReturnState(),
WheelRPMS, RobotRadius, DesClearance, WheelRadius, WheelDistance)#Generate New Nodes
from the possible moves current node can take.
        ParentC2C = current_node.ReturnC2C() #Get Parent C2C
        if NewNodeStates_and_Cost not in Closed_List: #Check to see if the new node
position is currently in the closed list
            for State in NewNodeStates_and_Cost: #For each new node generated by the
possible moves.
                ChildNode_C2C = ParentC2C + State[1] #Get C2C for the child node
                ChildNode_Total_Cost = ChildNode_C2C + Calculate_C2G(State[0],
GoalState) #Get Total Cost for Child Node

                NewChild = Node(State[0], current_node, State[2] ,ChildNode_C2C,
ChildNode_Total_Cost) #Generate New Child Node Class
                if CheckIfVisited(NewChild.ReturnState(), node_array, ThreshXY,
ThreshTheta) ==  False: #If the node has not been visited before
                    #Mark in Node Array
                    node_array[int(Round2Half(NewChild.ReturnState()[1])/ThreshXY),
int(Round2Half(NewChild.ReturnState()[0])/ThreshXY),
int(Round2Half(NewChild.ReturnState()[2])/ThreshTheta)] = 1
                    Open_List.put((NewChild.ReturnTotalCost() , NewChild)) #Put it
into the Open list
```

```
                if CheckIfVisited(NewChild.ReturnState(), node_array, ThreshXY,
ThreshTheta) ==  True: #If you have visited before:
                    if NewChild.ReturnTotalCost() > current_node.ReturnC2C() +
State[1]: #If the current total cost is greater than the move
                        NewChild.parent = current_node #Update Parent
                        NewChild.C2C = current_node.ReturnC2C() + State[1] #Update
C2C
                        NewChild.TotalCost = NewChild.ReturnC2C() +
Calculate_C2G(NewChild.ReturnState(), GoalState) #Update Total Cost

    if goalreachcheck: #If you reach goal
        break #Break the Loop

stoptime = timeit.default_timer() #Stop the Timer, as Searching is complete.
print("That took", stoptime - starttime, "seconds to complete")


#Show the Completed Searched Arena
plt.imshow(arena, origin='lower')
plt.show()

##------------------Visualization------------------##
print("Visualization Starting!")
plt.plot(InitState[0], InitState[1], 'go', markersize = 0.5) #plot init state
plt.imshow(arena, origin = 'lower')

for node in traversed_nodes: #Plots the search area
    curr_node_state = node.ReturnState()
    parent_node_state = node.ReturnParentState()
    PlotCurves(node.ReturnParentState(), node.ReturnMove(), WheelRadius,
WheelDistance, 'g', RobotRadius, DesClearance)
    plt.pause(0.0000000000000001)


for node in Path: #Plots the ideal path
    curr_node_state = node.ReturnState()
    parent_node_state = node.ReturnParentState()
    PlotCurves(node.ReturnParentState(), node.ReturnMove(), WheelRadius,
WheelDistance, 'm', RobotRadius, DesClearance)
    plt.pause(0.0001)

plt.show()
plt.close()
```

src/part_2/src/TurtleBot_Astar.py

```
#!/usr/bin/env python3

##----------------Importing Libraries------------------##
import rospy
from geometry_msgs.msg import Twist
import time
import numpy as np
import cv2 as cv
import timeit
import queue
from queue import PriorityQueue
import sys



class Node():
    #Initializing Function
    def __init__(self, state, parent, move, C2C, TotalCost):
        self.state = state
        self.parent = parent
        self.move = move
        self.C2C = C2C
        self.TotalCost = TotalCost

    #---Methods for this Class---#
    def ReturnState(self): #Returns Node State X and Y
        return self.state

    def ReturnParent(self): #Returns the Parent Node
        return self.parent

    def ReturnParentState(self): #Returns the Parent Node's State
        if self.ReturnParent() is None:
            return None
        return self.ReturnParent().ReturnState()

    def ReturnMove(self): #Returns Move
        return self.move

    def ReturnC2C(self): # Returns C2C
        return self.C2C

    def ReturnTotalCost(self): #Returns the Total Cost
        return self.TotalCost

    def __lt__(self, other): #OOP Definition for Less than. Required for Priority
Queue.
        return self.TotalCost < other.TotalCost

    ##--------------BACKTRACKING FUNCTION Integrated into Class--------##
    def ReturnPath(self):
```

```
        CompletedMoves = [] #Initialize Move Array
        NodePath = [] #Initialize the Node Path
        CurrentNode = self
        #while(CurrentNode.ReturnMove() is not None): #For move that a Node has made
        while(CurrentNode.ReturnMove() != [0,0]):
            CompletedMoves.append(CurrentNode.ReturnMove()) #Append the previous move
            NodePath.append(CurrentNode) #Append Node to Path
            CurrentNode = CurrentNode.ReturnParent() #Backtrack to the Parent before
repeating Process
        NodePath.append(CurrentNode) #Append the starting point after path is derived.
        NodePath.reverse() #Reverse Order to get front to back path
        CompletedMoves.reverse() #Reverse Order to get front to back path

        return CompletedMoves, NodePath



##---------------------Defining Obstacle Space Setup Functions-------------------##


#Checks to see if a point is within an obstacle
def checkObstacle(x, y):

    #Left Rectangle
    if x >= 1 and x < 1.15:

        if y < 1 and y >= -0.25:
            return True

    #Right Rectangle
    if x >= 2 and x < 2.15:

        if y < 0.25 and y >= -1:
            return True

    #Circle
    if (x - 3.5) * (x - 3.5) + (y - 0.1) * (y - 0.1) <= 0.5*0.5:
        return True

    return False

#Checks to see if a point is within the border of an obstacle
def checkBorder(x, y, s):

    #Left Rectangle
    if x >= 1 - s and x < 1.15 + s:

        if y < 1 + s and y >= -0.25 - s:
            return True

    #Right Rectangle
    if x >= 2 - s and x < 2.15 + s:

        if y < 0.25 + s and y >= -1:
            return True
```

```
    #Circle
    if (x - 3.5) * (x - 3.5) + (y - 0.1) * (y - 0.1) <= (0.5 + s) * (0.5 + s):
        return True

    return False

#Checks to see if a point is within radial clearance of a border
def checkClearance(x, y, s, r):

    rr = r - 0.01

    if rr == 0:
        return False

    #Left Rectangle
    if x >= 1 - s - rr and x < 1.15 + s + rr:

        if y < 1 + s + rr and y >= -0.25 - s - rr:
            return True

    #Right Rectangle
    if x >= 2 - s - rr and x < 2.15 + s + rr:

        if y < 0.25 + s + rr and y >= -1:
            return True

    #Circle
    if (x - 3.5) * (x - 3.5) + (y - 0.1) * (y - 0.1) <= (0.5 + s + rr) * (0.5 + s +
rr):
        return True

    return False

#Checks to see if a point is valid (by checking obstacle, border, and clearance, as
well as making sure the point is within arena bounds)
def checkValid(x, y, s, r):

    if checkObstacle(x, y):
        return False

    if checkBorder(x, y, s):
        return False

    if checkClearance(x, y, s, r):
        return False

    if (x < -0.5 + r + s or x >= 5.5 - r - s or y < -1 + r + s or y >= 1 - r - s):
        return False

    return True


##---------------------------------Defining my Action
Set----------------------------------------##
```

```
def ReturnPossibleStates(CurrentNodeState, Wheel_RPMS, RobotRadius, ObsClearance,
WheelRad, WheelDist):
    RPM1 = Wheel_RPMS[0]
    RPM2 = Wheel_RPMS[1]
    ActionSet = [[RPM1, RPM1], [RPM2,RPM2],[RPM1, RPM2], [RPM2, RPM1], [0,RPM1],
[RPM1,0], [0,RPM2], [RPM2,0]] #Differential Drive Action Set
    NewNodeStates = [] #Init List of States

    for action in ActionSet: #For each differential drive action
        NewNodeState, Cost = CalcMoveWithCost(CurrentNodeState, action, RobotRadius,
ObsClearance, WheelRad, WheelDist) #Calculate the state and cost
        if NewNodeState is not None:
            NewNodeStates.append([NewNodeState, Cost, action]) #Append Chile Node
States
    return NewNodeStates

##---------------------------------Defining my Cost and NewNodeState
Function--------------------------------------##

def CalcMoveWithCost(CurrentNodeState, WheelAction, RobotRadius, ObsClearance,
WheelRad, WheelDist):
    t = 0
    dt = 0.1
    Curr_Node_X = CurrentNodeState[0] #Grab Current Node X
    Curr_Node_Y = CurrentNodeState[1] #Grad Current Node Y
    Curr_Node_Theta = np.deg2rad(CurrentNodeState[2]) #Grab Current Node Theta,
convert to radians.

    MoveCost = 0.0 #Init Cost

    New_Node_X = Curr_Node_X #Set New Node Start Point X
    New_Node_Y = Curr_Node_Y #Set New Node Start Point Y
    New_Node_Theta = Curr_Node_Theta #Set New Node Start Point Theta

    ##----------------Euler Integration to Generate Curvature----------------##
    while t < 1:
        t += dt
        ChangeX = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.cos(New_Node_Theta)*dt
        ChangeY = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.sin(New_Node_Theta)*dt
        ChangeTheta = (WheelRad/WheelDist)*(WheelAction[0]-WheelAction[1])*dt

        New_Node_X += ChangeX
        New_Node_Y += ChangeY
        New_Node_Theta += ChangeTheta

        MoveCost += np.sqrt((ChangeX)**2 + (ChangeY)**2)

        ##-----------Why CheckValid is inside the loop---------------##
        '''Inside the loop because if we only checked final, the intermediate steps
would sometimes be in the obstacle space.'''
        if checkValid(New_Node_X, New_Node_Y, ObsClearance, RobotRadius) == False:
            return None, None

    New_Node_Theta = int(np.rad2deg(New_Node_Theta)) #Convert back to Degrees
```

```
    ##-----Wrap to -360-360-----##
    if New_Node_Theta >= 360:
        New_Node_Theta = New_Node_Theta - 360
    if New_Node_Theta < -360:
        New_Node_Theta = New_Node_Theta + 360

    return [New_Node_X, New_Node_Y, New_Node_Theta], MoveCost

##---------------------------Defining my Cost to Go
Calculation---------------------------##
def Calculate_C2G(CurrentNodeState, GoalNodeState):
    C2G = 0.0
    X_Current = CurrentNodeState[0]
    Y_Current = CurrentNodeState[1]
    X_Goal = GoalNodeState[0]
    Y_Goal = GoalNodeState[1]
    if CurrentNodeState is not None:
        C2G = np.sqrt((X_Goal-X_Current)**2 + (Y_Goal- Y_Current)**2) #Euclidian
Distance Heuristic function
    return C2G

##-----------------------Defining my Compare to Goal
Function---------------------------##

def CompareToGoal(Current_Node_Position, Goal_Node_Position, ErrorThreshold):
    Dist2Goal = (Goal_Node_Position[0] - Current_Node_Position[0])**2 +
(Goal_Node_Position[1] - Current_Node_Position[1])**2 #Euclidian Distance
    if Dist2Goal < ErrorThreshold**2: #Error less than threshold PLUS the angle has to
be equal
        return True
    else:
        return False

##-------------------------Defining my Round to Half
Function-------------------------##
''' This function is Required for "Check Visited" Capabilities'''
def Round2Half(number):
    testvalue = np.round(2*number)/2
    if (testvalue == 10):
        testvalue = testvalue - 0.5
    return testvalue

##---------------------------Defining my Check Visited
Function-----------------------##
def CheckIfVisited(Current_Node_State, Node_Array, XYThreshold, ThetaThreshold):
    X = Current_Node_State[0]
    Y = Current_Node_State[1]
    Theta = Current_Node_State[2]
    X = int(Round2Half(X)/XYThreshold)
    Y = int(Round2Half(Y)/XYThreshold)
    Theta = int(Round2Half(Theta)/ThetaThreshold)
    if Node_Array[Y,X,Theta] == 1:
        result = True
    else:
        result = False
    return result
```

```
##-------------------------Defining my Plotting Functions-------------------------##
'''For Integers'''
def WSColoring(Workspace, Location, Color):
    x,_,_ = Workspace.shape #Get Shape of Workspace
    translation_x = Location[1] #Where in X
    translation_y = Location[0] #Where in Y
    Workspace[translation_x,translation_y,:] = Color #Change the Color to a set Color
    return Workspace

##-------------------------Defining my Velocity Commands
Function-------------------------##
global XVelocity2Publish
global Omega2Publish
XVelocity2Publish = []
Omega2Publish = []

def DeriveVelocity(ParentNodeState, WheelAction, WheelRad, WheelDist):
    t = 0
    dt = 0.1
    Curr_Node_X = ParentNodeState[0]
    Curr_Node_Y = ParentNodeState[1]
    Curr_Node_Theta = np.deg2rad(ParentNodeState[2])

    New_Node_X = Curr_Node_X
    New_Node_Y = Curr_Node_Y
    New_Node_Theta = Curr_Node_Theta

    while t < 1:
        t += dt
        ChangeX = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.cos(New_Node_Theta)*dt
        ChangeY = 0.5*WheelRad*
(WheelAction[0]+WheelAction[1])*np.sin(New_Node_Theta)*dt
        ChangeTheta = (WheelRad/WheelDist)*(WheelAction[0]-WheelAction[1])*dt

        Xdot = ChangeX/dt
        Ydot = ChangeY/dt
        ThetaDot = ChangeTheta/dt

        V = np.sqrt(Xdot**2 + Ydot**2)

        XVelocity2Publish.append(V)
        Omega2Publish.append(ThetaDot)

        New_Node_X += ChangeX
        New_Node_Y += ChangeY
        New_Node_Theta += ChangeTheta

##-----------------------------------"Main"
Script-------------------------------------##

if __name__=='__main__':
    msg = Twist()
    pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
```

```
    rospy.init_node('robot_talker', anonymous=True)

    while not rospy.is_shutdown():
        try:

            ##-------Getting Parameters from Burger TurtleBot Dimensions-------##

            WheelRadius = 0.038 #m
            RobotRadius = 0.178 #m
            WheelDistance = 0.354 #m

            InitState = [float(sys.argv[1]), float(sys.argv[2]), float(sys.argv[3])]
            GoalState = [float(sys.argv[4]), float(sys.argv[5])]
            DesClearance = float(sys.argv[6])
            WheelRPMS = [float(sys.argv[7]), float(sys.argv[8])]

            #-----Check Valid Initial State-------##
            if not checkValid(InitState[0], InitState[1], RobotRadius, DesClearance):
                print("Your initial state is inside an obstacle or outside the
workspace. Please retry.")
                exit()

            ##----Check Valid Goal State---------##
            if not checkValid(GoalState[0], GoalState[1], RobotRadius, DesClearance):
                print("Your goal state is inside an obstacle or outside the workspace.
Please retry.")
                exit()


            #Initialize Arena and Thresholds
            SizeArenaX = 600
            SizeArenaY = 200
            ThreshXY = 0.5
            ThreshTheta = 30
            ThreshGoalState = 0.03

            # Initialize Node Array
            node_array = np.array([[[ 0 for k in range(int(360/ThreshTheta))]
                                    for j in range(int(SizeArenaX/ThreshXY))]
                                    for i in range(int(SizeArenaY/ThreshXY))])

            Open_List = PriorityQueue() #Initialize list using priority queue.
            traversed_nodes = [] #Traversed nodes is for visualization later.
            starting_node_Temp = Node(InitState, None, [0,0], 0,
Calculate_C2G(InitState, GoalState)) #Generate temp starting node based on the initial
state given above.
            starting_node = Node(InitState, starting_node_Temp, [0,0], 0,
Calculate_C2G(InitState, GoalState)) #Generate starting node based on the initial
state given above.
            Open_List.put((starting_node.ReturnTotalCost(), starting_node)) #Add to
Open List
            GoalReach = False #Initialze Goal Check Variable
            Closed_List= np.array([])#Initialize Closed List of nodes. Closed list is
based on node states
            starttime = timeit.default_timer() #Start the Timer when serch starts
            print("A* Search Starting!!!!")
```

```
            while not (Open_List.empty()):
                current_node = Open_List.get()[1] #Grab first (lowest cost) item from
Priority Queue.
                print(current_node.ReturnState(), current_node.ReturnTotalCost())
#Print to show search is working.
                np.append(Closed_List, current_node.ReturnState()) #Append to Closed
List
                goalreachcheck = CompareToGoal(current_node.ReturnState(), GoalState,
ThreshGoalState) #Check if we have reached goal.

                if goalreachcheck: #If we have reached goal node.
                    print("Goal Reached!")
                    print("Total Cost:", current_node.ReturnTotalCost()) #Print Total
Cost




                    MovesPath, Path = current_node.ReturnPath() #BackTrack to find
path.
                    for nodes in Path: #For Each node in ideal path
                        DeriveVelocity(nodes.ReturnParentState(), nodes.ReturnMove(),
WheelRadius, WheelDistance)



                else: #If you have NOT reached the goal node
                    NewNodeStates_and_Cost =
ReturnPossibleStates(current_node.ReturnState(), WheelRPMS, RobotRadius, DesClearance,
WheelRadius, WheelDistance)#Generate New Nodes from the possible moves current node
can take.
                    ParentC2C = current_node.ReturnC2C() #Get Parent C2C
                    if NewNodeStates_and_Cost not in Closed_List: #Check to see if the
new node position is currently in the closed list
                        for State in NewNodeStates_and_Cost: #For each new node
generated by the possible moves.
                            ChildNode_C2C = ParentC2C + State[1] #Get C2C for the
child node
                            ChildNode_Total_Cost = ChildNode_C2C +
Calculate_C2G(State[0], GoalState) #Get Total Cost for Child Node


                            NewChild = Node(State[0], current_node, State[2]
,ChildNode_C2C, ChildNode_Total_Cost) #Generate New Child Node Class
                            if CheckIfVisited([100*NewChild.ReturnState()[0],
100*NewChild.ReturnState()[1], NewChild.ReturnState()[2]], node_array, ThreshXY,
ThreshTheta) ==  False: #If the node has not been visited before
                                #Mark in Node Array
                                node_array[int(Round2Half(100*NewChild.ReturnState()
[1])/ThreshXY), int(Round2Half(100*NewChild.ReturnState()[0])/ThreshXY),
int(Round2Half(NewChild.ReturnState()[2])/ThreshTheta)] = 1
                                Open_List.put((NewChild.ReturnTotalCost() , NewChild))
#Put it into the Open list


                            if CheckIfVisited([100*NewChild.ReturnState()[0],
100*NewChild.ReturnState()[1], NewChild.ReturnState()[2]], node_array, ThreshXY,
ThreshTheta) ==  False: #If the node has not been visited before
                                    if NewChild.ReturnTotalCost() >
current_node.ReturnC2C() + State[1]: #If the current total cost is greater than the
move
                                        NewChild.parent = current_node #Update Parent
                                        NewChild.C2C = current_node.ReturnC2C() +
State[1] #Update C2C
```

```
                                        NewChild.TotalCost = NewChild.ReturnC2C() +
Calculate_C2G(NewChild.ReturnState(), GoalState) #Update Total Cost

                if goalreachcheck: #If you reach goal
                    break #Break the Loop

            stoptime = timeit.default_timer() #Stop the Timer, as Searching is
complete.
            print("That took", stoptime - starttime, "seconds to complete")

            print("Simulation beginning! Please refer to Gazebo to watch.")

            CommandArray = np.column_stack((Omega2Publish, XVelocity2Publish))

            ##--------------------ROS Publisher Function Setup and
Definition------------------##

            rate = rospy.Rate(10)

            for item in CommandArray:
                msg.angular.x = 0
                msg.angular.y = 0
                msg.angular.z = item[0]
                # print("angular is ", msg.angular.z)
                msg.linear.x = item[1]
                # print("linear is ", msg.linear.x)
                msg.linear.y = 0
                msg.linear.z = 0
                pub.publish(msg)
                rate.sleep()

        except rospy.ROSInternalException:
            print("Here is your error")
```