NAME : ARPIT SAVARKAR

Algorithamic Motion Planning

HW-3

Q.1

Ans a) A planning algorithm is called a "complete" planning algorithm if it can find a path from start configuration to the goal configuration if such a path exists in finite time or responds with a failure condition in case of infeasibility of a path.

b) A planning algorithm is called an optimal, in the sense that ~~is~~ given a cost function $c: T \rightarrow \mathbb{R}^{\geq 0}$, it can find a path $\gamma^*: [0,1] \rightarrow Q_{free}$ such that

$$\gamma^* = \underset{\gamma \in T}{\arg\min} \{ c(\gamma(t)) \mid \gamma(0) = q_{start} \ \& \ \gamma(1) = q_{goal} \}$$

c)

Ans In terms of completeness the alg wavefront planner will find a path from the goal to the start if such a path exists. This assumes the fact that the obstacles, start & goal are not dynamic and ~~f~~ poses of all obstacles are known. And thus it is a complete algo

In sense of optimality, the wavefront planner is optimal in the sense of grid world (discritization) and could be sub-optimal in the continious domain. It is a kind of BFS (Breadth first search) assiging and manipulating weights along the search.

~~A~~ Wavefront algorithm satisfies the 3 conditions ~~to~~

1) It provides a stage index indicating current plan step
2) It moves over a cost function to ~~ex~~ optimize current stage index
3) It consists of a termination condition / action when it is time to stop the plan & fix the cost

Q.2

a) (i) Attached as a color pdf quiver plot. ~~at the~~

(ii) The parameters ~~selecte~~ ~~were~~ selected were

DSTAR = 8

ξ, Attraction Gain = 10

η, Repulsive Gain = 100

$Q^* = [1, 2]$

Since the start was closer to ~~a~~ the 1^{st} obstacle, a large repulsive gain ~~ha~~ had to chosen with varying Q^* start values to account for the path generated to follow in between the obstacles

Additionally, to prevent excess quadratic flow of attraction potential, ~~a~~ d^* was selected to be 8 units.

iv) length of the path generated = 6.14 units

iii) Path generated and vector field have been combined in the same plot.

v) Absolutely NOT, for different values of d^* & Q_i^* different path length are generated.

b)

Ans $\boxed{\text{For Workspace - 1}}$

The following parameters were selected

$d^* = 3$

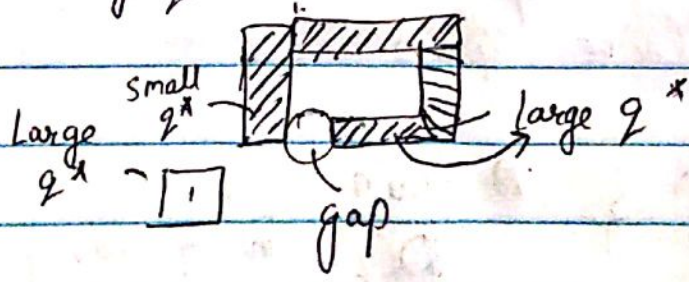
ζ = Attraction Gain = 9.2

η = Repulsive Gain = 2

$Q^* = [2.7, 0.5, 1.2, 1.2, 5]$

(i) d^* was chosen closer to goal to get quadratic convergence away from the goal, this did not significantly affect the result

For obstacles closer to start large Q^* had to be chosen, to prevent local minimas, and for the small gap, one obstacle was given a large q^*



(ii) Attached
(iii) length of the path generated = 8.31 units
(iv) Absolutely NOT, different path lengths are generated for different values

For Workspace 2

Classic Gradient Descent approach was unable to find path and was regularly getting stuck on the local minima.

Additional Repulsive Potential function along with the original Potential fields had to be used to create a elliptical potential fields around the obstacles.

The base for these additional function was considered as a distance of the center of the obstacles from the current state of the robot and was used as an additional repulsive heuristic adding to the repulsive Gain

i.e,

$$U'_{rep}(q) \begin{cases} \dfrac{\text{Gain}}{\left(D(q - \text{center of gravity of the obstacle})\right)^2} & \text{if } D(q) \leq r \\ 0 & \text{if } D(q) > r \end{cases}$$

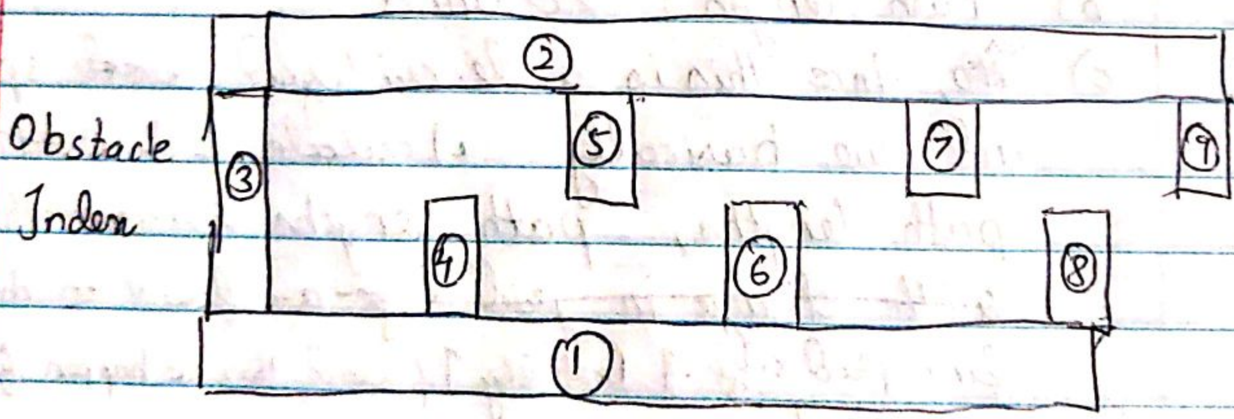
Scanned with CamScanner

Using the additional heuristic as explained above ~~gave~~ leads to following parameters

$$\eta = [4, 4, 7, 7, 7, 7, 7, 7, 7]$$
$$Q^* = [4, 4, 6, 0.1, 0.1, 0.1, 0.1, 0.1, 0.9]$$
$$\text{Centroid Gain} = [40, 60, 20, 50, 50, 50, 50, 50, 50]$$
$$\text{Obs Radius} = [5, 5, 7.5, 3.5, 3.5, 3.5, 3.5, 3.5, 3.5]$$

Each element of the above parameters ~~cor~~ represent the parameters for the obstacles, i.e, the 1st element of η, Q^*, Centroid Gain, Obstacle Radius are used for the 1st obstacle,

Namely



(ii) Path length generated : Attached

(iii) Lengths of the path generated : 64.32 units

(iv) Absolutely NOT, I was not able to get the gradient descent to converge with classic potential fields and had to use additional heuristics, thus changing d^* & Q_i leads to different path length.

Q.3

Ans $\boxed{\text{Workspace - 1}}$

a) Attached

b) Path length : 20 units

c) Yes ~~No~~, since this is a distritized ~~world~~ space, and we basically calculate L1 norm as path lengths, path lengths ~~remain the same with change in grid size~~ are found to change for very fine grid size. ~~I~~ ~~Probably~~ I found this to happen for grid size of 0.05 and less

$\boxed{\text{Workspace - 2}}$

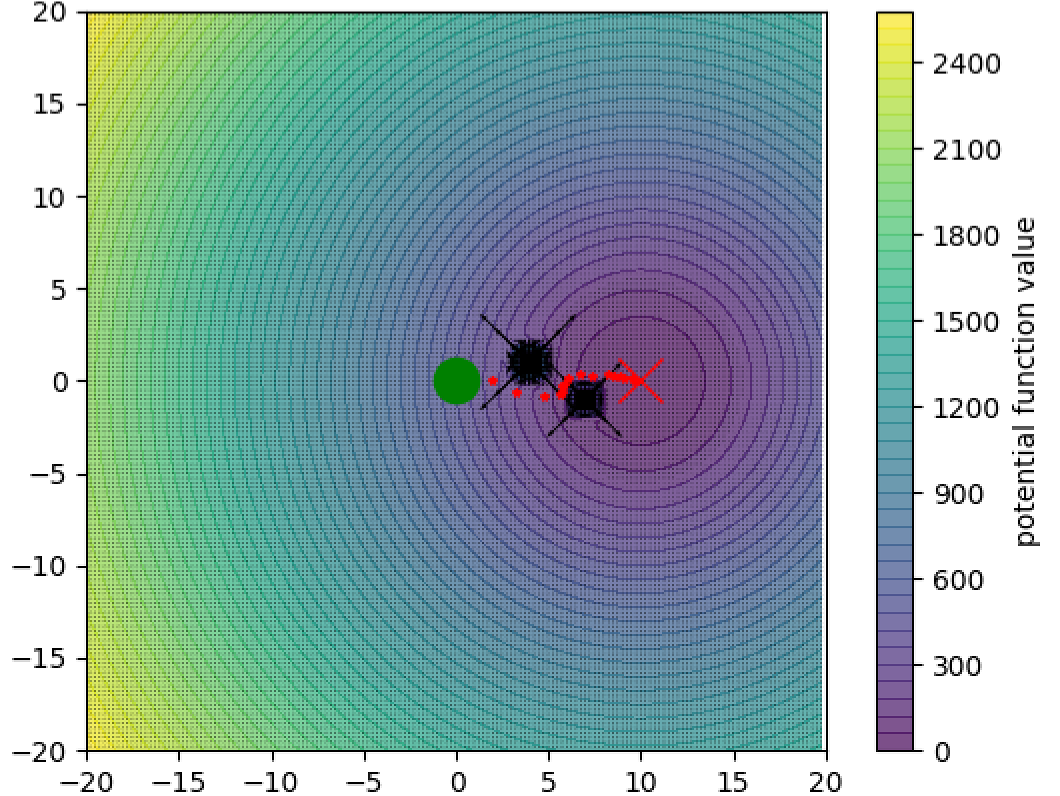
a) Attached

b) Path length : 44 units

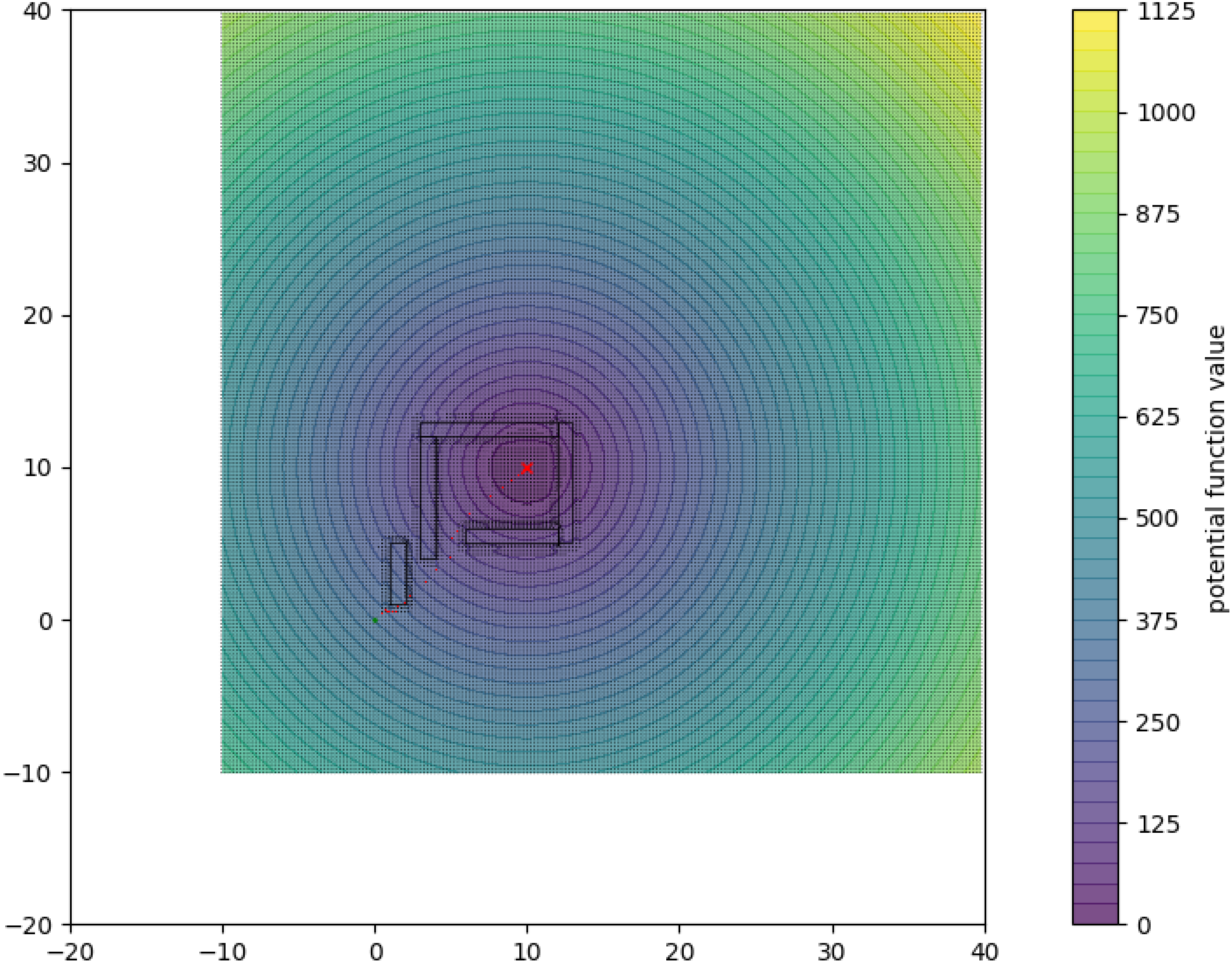
c) Path lengths, ~~remain~~ are found to change the ~~same~~ ~~irrespective~~ of grid size because of L1 norm used for path length calculation and very fine grid size causes the change in results, I found that results change for grid size less than 0.05 units and less

Q.3 Q) The wavefront planner does a better job navigating along the obstacles to reach goal when in comparison with gradient descent.

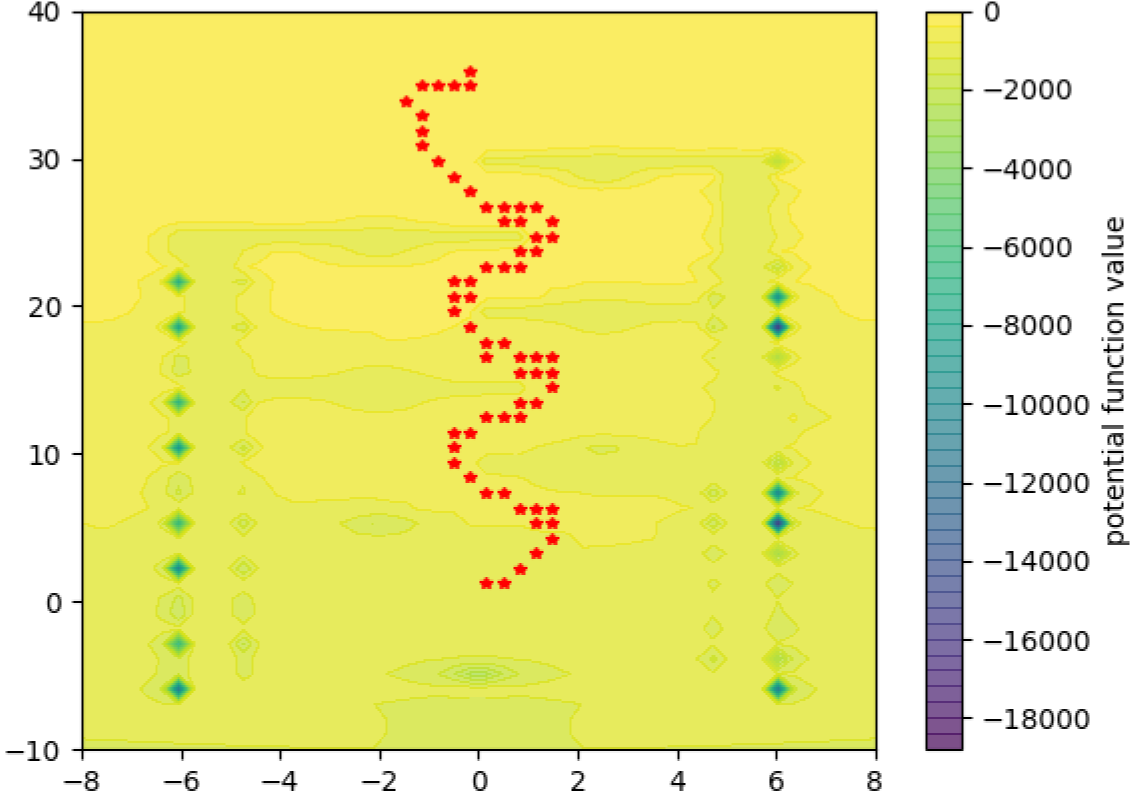
Q.4
Ans Attached

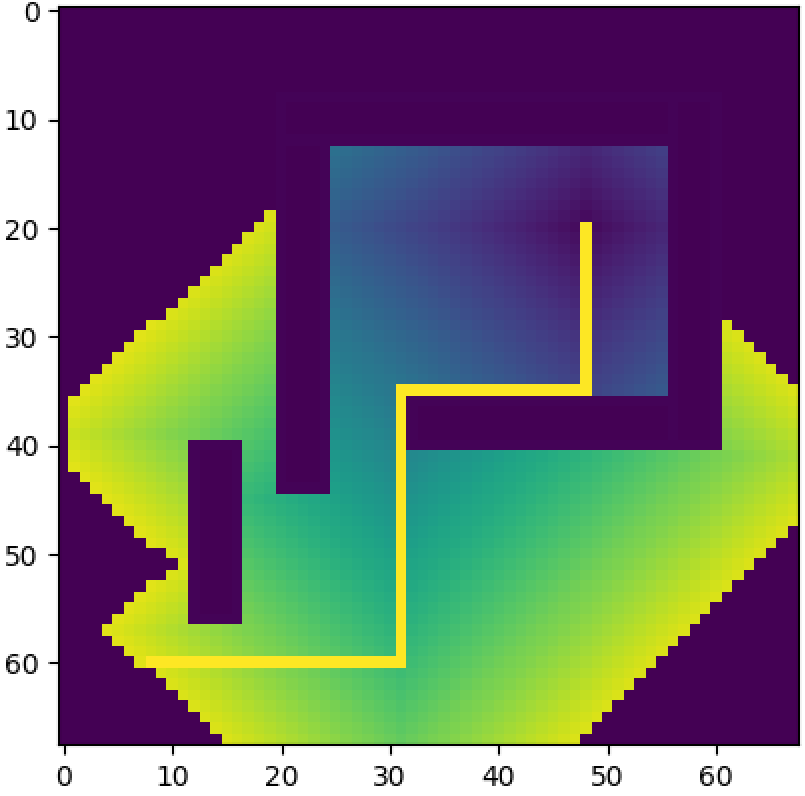


20
15
10
5
0
−5
−10
−15
−20
−20
−15
−10
−5
0
5
10
15
20
2400
2100
1800
1500
1200
900
600
300
0
potential function value

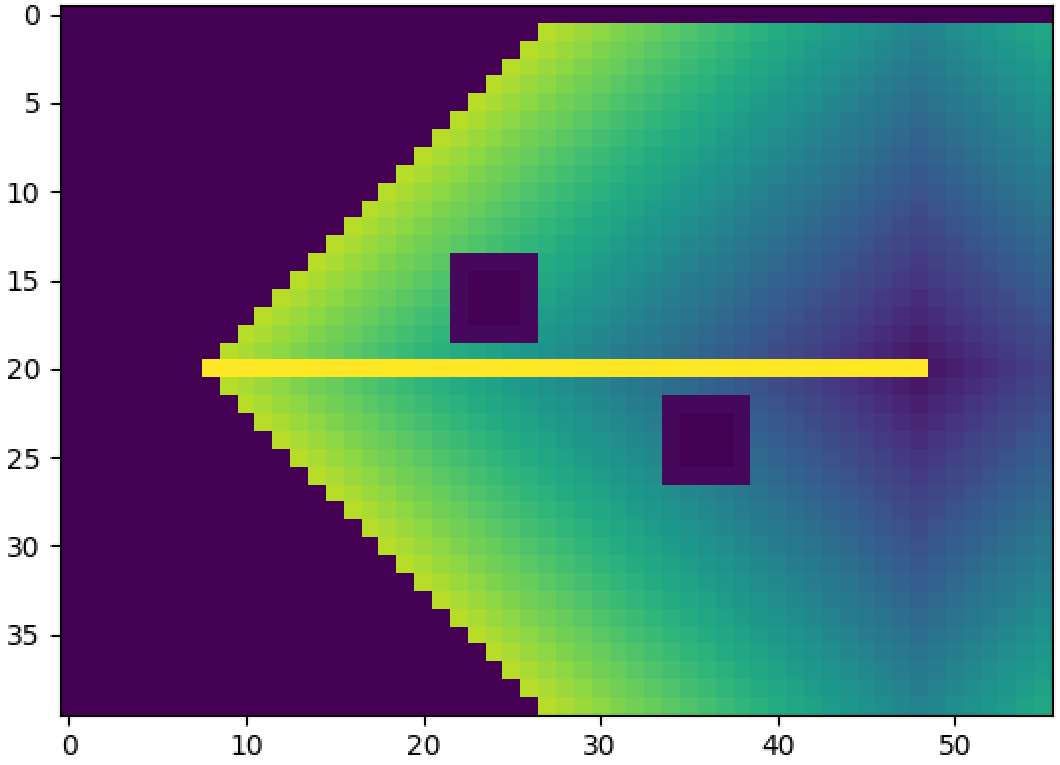


40
30
20
10
0
−10
−20
−20
−10
0
10
20
30
40
1125
1000
875
750
625
500
375
250
125
0
potential function value


40
30
20
10
0
−10
−8
−6
−4
−2
0
2
4
6
8
0
−2000
−4000
−6000
−8000
−10000
−12000
−14000
−16000
−18000
potential function value



0
10
20
30
40
50
60
0
10
20
30
40
50
60


0
5
10
15
20
25
30
35
0
10
20
30
40
50

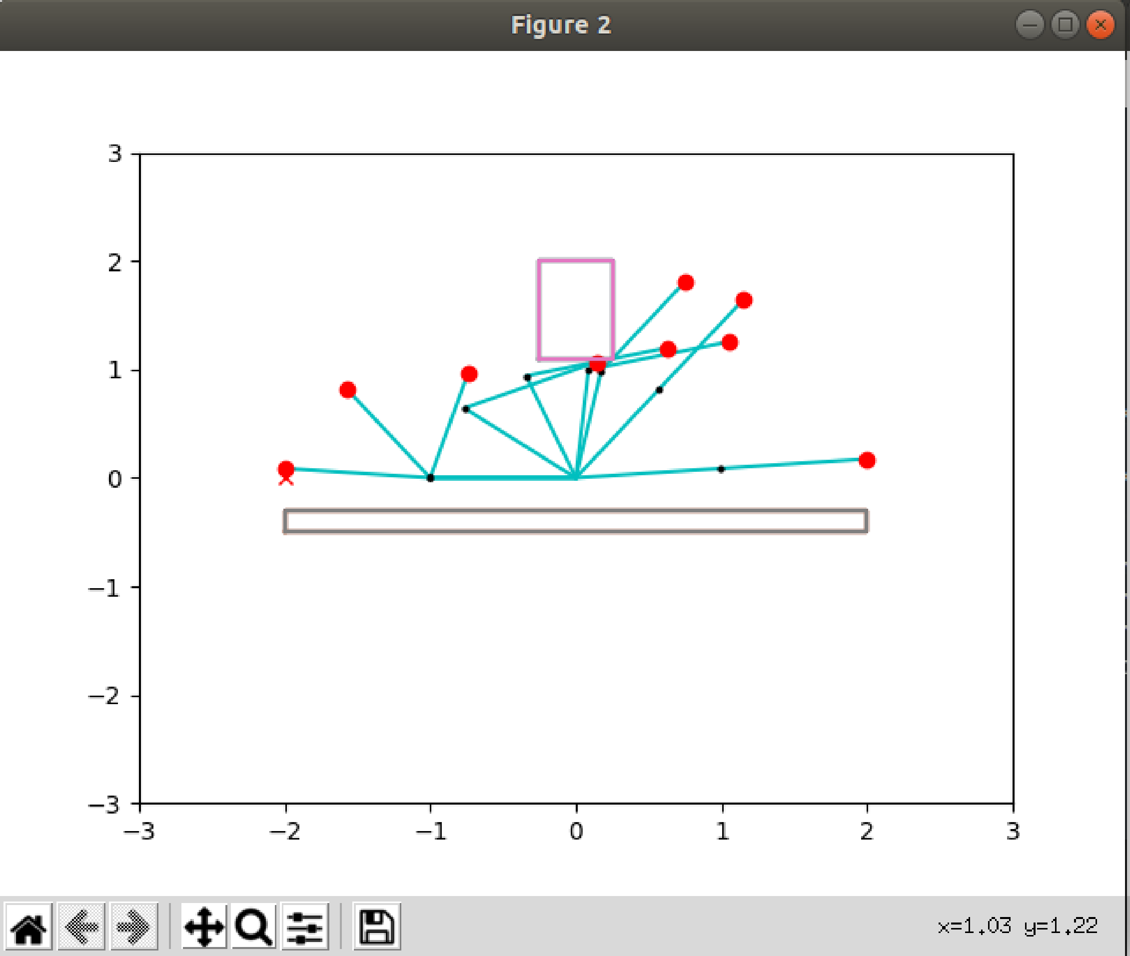


Figure 2
x=1.03 y=1.22

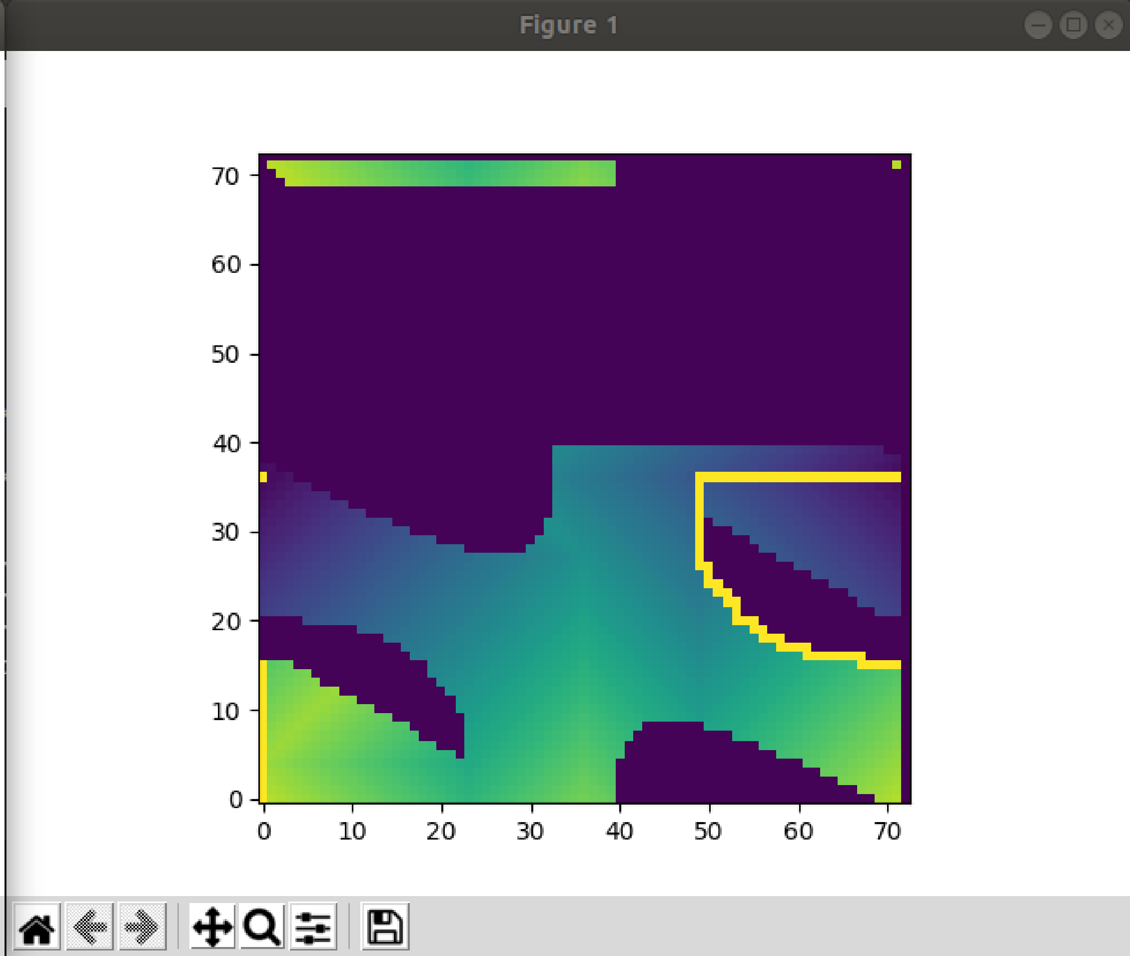
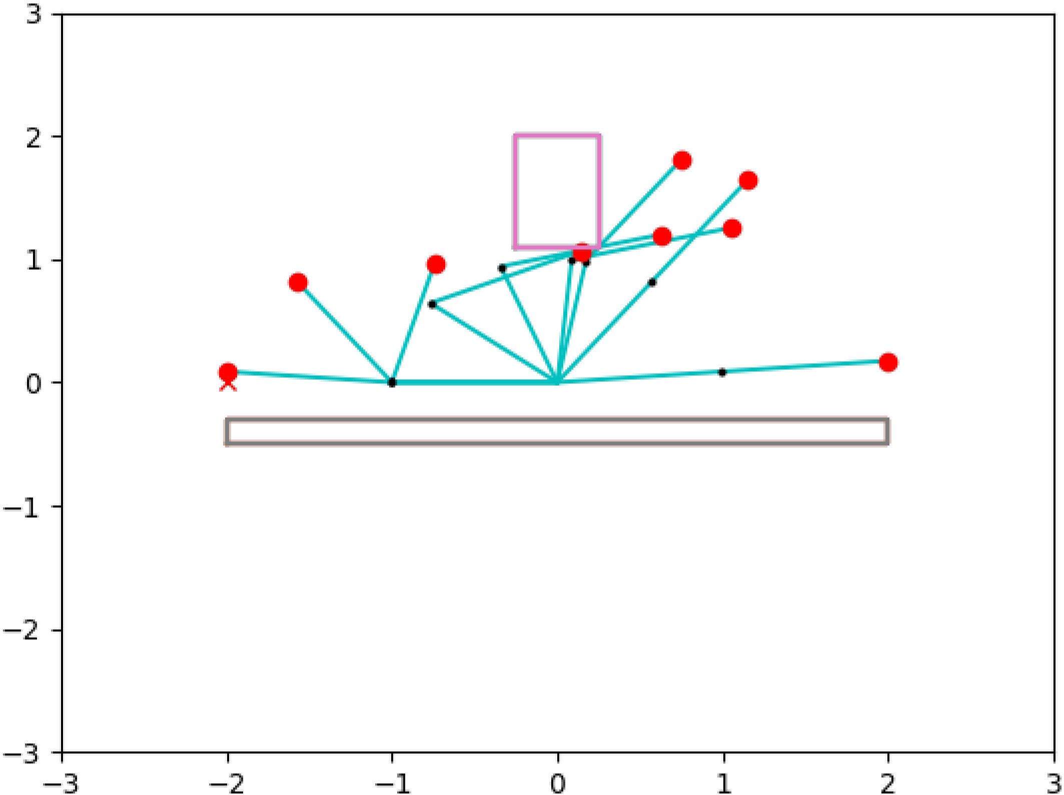
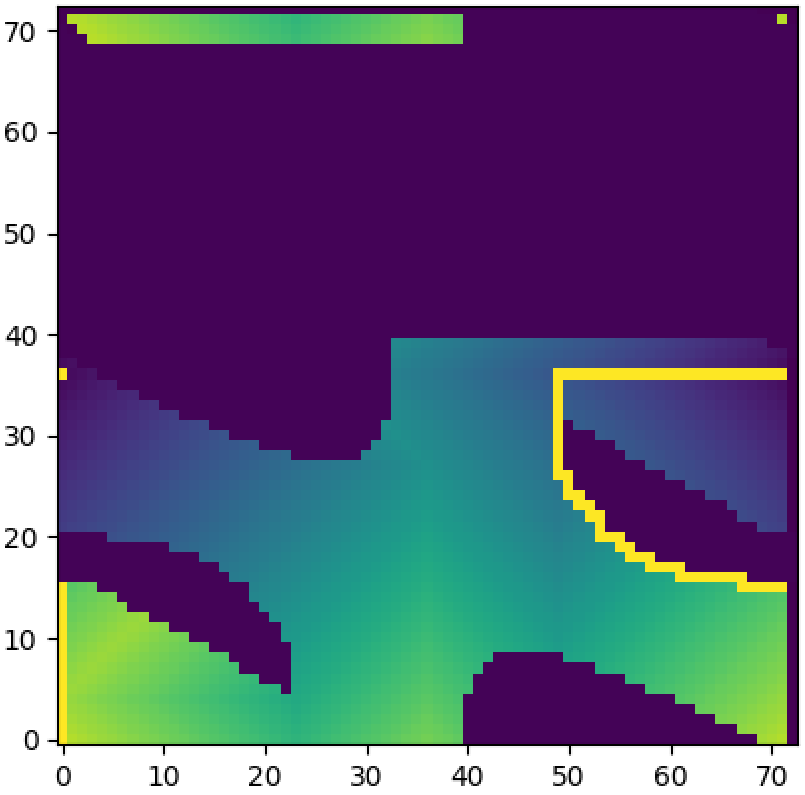


Figure 1


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


0
10
20
30
40
50
60
70
0
10
20
30
40
50
60
70

==
===========================
Name : Arpit Savarkar

==
===========================

===
potential_field.py
===
```
"""
Implementation of the Potential Field

I would like to thank Professor Morteza Lahijanian and the fellow course mate Kedar More for
discussion during the Implementation of this code

"""



import numpy as np
from shapely.geometry import Point, Polygon
import matplotlib.pyplot as plt
from scipy.interpolate import griddata
import polytope as pc
from workspaces import config

WORKSPACE_CONFIG = config()


# Global Flags
XLIMIN = -10
XLIMAX = 40
NUMS = 200

CENTROID_GAIN = [40, 60, 20, 300 ,60, 50, 50, 50, 50, 50]
OBS_RADIUS =    [5, 5, 7.5, 8 ,3.5, 5.5, 5.5, 5.5, 5.5, 5.5]


def attractivePotential(state, ATTRACT_GAIN, DSTARGOAL, q_goal):
    """
    Calculates the Attraction Potential based on the hyper parameters
    for each state
    PARAMETERS
    ----------
    state : numpy array
    ATTRACT_GAIN : Attraction Gain
    DSTARGOAL : Attraction distance over which the function is non-linear
    q_goal : Goal
    """

    distToGoal = np.hypot(state[0] - q_goal[0], state[1] - q_goal[1])

    # unroll for speed
    gain = ATTRACT_GAIN
```

```
    dStarGoal = DSTARGOAL

    if distToGoal <= dStarGoal:
        # Linear
        U_att = 0.5 * gain * distToGoal ** 2
    else:
        # Non - Linear
        U_att = dStarGoal * gain * distToGoal - 0.5 * gain * dStarGoal ** 2

    return U_att

def repulsivePotential(state, obs, REPULSIVE_GAIN, Q_STAR):
    """
    Repulsive Function
    PARAMETERS
    ----------
    state : Current state of the point robot
    obs : List of Shapely.Geometry.MultiPolygon
    REPULSIVE_GAIN : List of Gain for each obstacle
    Q_STAR : List of repulsive distance gain, from obstacle
    """
    obstacles = obs
    GAIN = REPULSIVE_GAIN

    # To prevent shattering, sum of all obstacles
    U_rep = 0

    for (obstacle, qStar, gn) in zip(obstacles, Q_STAR, GAIN):
        # Distance to the obstacle
        distToObst = abs(obstacle.exterior.distance(Point(state)))

        # return NAN in case of collision with the obstacle
        if distToObst == 0:
            distToObst = np.nan
            U_rep = 10

        elif Point(state).within(obstacle):
            # Inside Obstacle
            distToObst = np.nan
            U_rep = 10
        else:
            # Inside Q_Star
            if distToObst <= qStar:
                U_rep += 0.5 * gn * (1 / distToObst - 1 / qStar) ** 2
            else:
                U_rep += 0

    return U_rep

def repulsivePotential2(state, obs):
    """
    PARAMETERS
    ----------
```

```
    state : Current state of the point of the robot
    obs : List of Shapely.Geometry.MultiPolygon
    """
    global OBS_RADIUS
    global CENTROID_GAIN
    obstacles = obs
    RAD = OBS_RADIUS
    GAIN = CENTROID_GAIN
    dist = []
    U_rep = 0

    for (obstacle, gn, r) in zip(obstacles, GAIN, RAD):
        # return NAN, when on obstacle boundary
        distToObst = abs(obstacle.exterior.distance(Point(state)))
        if distToObst == 0:
            distToObst = np.nan
            U_rep = 10

        elif Point(state).within(obstacle):
            # If inside obstacle
            distToObst = np.nan
            U_rep = 10
        else:
            # Center of Mass/Gravity based
            cg = obstacle.centroid
            p = Point(state)

            # Elliptical Distance
            dist = np.hypot(p.x - cg.x, p.y - cg.y)
            if dist < r:
                U_rep = gn/(dist**2)
                return U_rep
            else:
                U_rep = 0

    return U_rep

def potential(state, obs, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR, q_goal):
    """
    Total Potential for a state
    """
    return (attractivePotential(state, ATTRACT_GAIN, DSTARGOAL, q_goal) +\
        repulsivePotential(state, obs, REPULSIVE_GAIN, Q_STAR))

def potential_large_workspace(state, obs, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR):
    """
    Total Potential for a state including elliptical distance
    """
    return (attractivePotential(state, ATTRACT_GAIN, DSTARGOAL, q_goal) +\
        repulsivePotential(state, obs, REPULSIVE_GAIN, Q_STAR) +\
            repulsivePotential2(state, obs))

def isCloseTo(state, q_goal, epsilon=0.25):
    """
```

```
    Returns true with within goal boundary
    PARAMETERS
    ----------
    state: numpy array - current state of the point robot
    q_goal: Goal

    """

    dist = np.linalg.norm(state-q_goal)

    return (dist <= epsilon)

def isAtGoal(state, q_goal):
    """
    Helper Function to check if the current state of the robot is at Goal
    """

    closeToGoal = isCloseTo(state, q_goal, epsilon=0.25)

    return closeToGoal


def calc_potential_field(q_start, q_goal, obs, NUMS, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_
STAR):
    """
    Calculates the potential of the all the states in the grid
    PARAMETERS
    ----------
    q_start : Start position of the point robot
    q_goal : Goal Position
    obs : List of Shapely.Geometry.MultiPolygon
    NUMS: Grid Size
    DSTARGOAL : Hyper parameter
    ATTRACT_GAIN : Hyper Parameter
    REPULSIVE_GAIN : Hyper Parameter
    Q_STAR : Hyper parameter
    """

    x_coor = np.arange(XLIMIN, XLIMAX, 0.25)
    y_coor = np.arange(XLIMIN, XLIMAX, 0.25)
    Y, X = np.meshgrid(x_coor, y_coor)
    nCoordsX = x_coor.shape[0]
    nCoordsY = y_coor.shape[0]

    U = np.zeros((nCoordsX, nCoordsY))
    points = []

    for i_x, x in enumerate(x_coor):
        for i_y, y in enumerate(y_coor):
            state = np.array([x, y])
            ptentl = potential(state, obs, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR, q_goal)
            U[i_y, i_x] = ptentl
            points.append((x, y))
```

```
    # fig = plt.figure()
    # ax = fig.gca(projection='3d')
    # x_len = x_coor.shape[0]
    # y_len = y_coor.shape[0]
    # ax.plot_surface(X, Y, U)
    # plt.show()

    return U, points

def gradientDescent(x, y, obs, q_start, q_goal, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR):
    """
    Gradient Descent using the Potential Field generated at the top
    x: Numpy array of the possible x direction of the point robot
    y: Numpy array of the possible y direction of the point robot
    """

    # Tolerance
    minimaTol = 1e-4

    # max iterations
    num_iterations = 3000000

    # Calculates the Potential Field
    U, points = calc_potential_field(q_start, q_goal, obs, 0.25, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GA
IN, Q_STAR)
    print("Entering Into Gradient Descent ")
    routes = []
    state = q_start
    currX, currY = 0,0

    # Appends the Route, state of the robot
    routes.append([currX, currY])

    # calculate the gradient on the CSpace grid
    dy, dx = np.gradient(U)

    # Part of gradient descent implementation
    dx_values = dx.flatten(order='F')
    dy_values = dy.flatten(order='F')

    # Gradeinte Desent Parameters
    iterCount = 0
    updateRate = 20

    # Gradeint Descent Implementation includes the
    while not isAtGoal(np.array([currX, currY]), q_goal):


        interp_gradient = np.zeros((2, 1))
        currX, currY = state

        # For Smooth Gradient Descent
        interp_dx = griddata(points, dx_values, (currX, currY), method='cubic')
        interp_dy = griddata(points, dy_values, (currX, currY), method='cubic')
```

```
        interp_gradient[0] = 0.005 * interp_dx
        interp_gradient[1] = 0.005 * interp_dy

        print('state: ', state, 'dx: ', interp_dx, 'dy:', interp_dy)

        # Data Logging
        shouldPrint = (iterCount % updateRate == 0)
        if shouldPrint:
            routes.append([round(currX[0], 1), round(currY[0], 1)])

        iterCount += 1

        # Updates the Status based on the gradeint
        state -= interp_gradient
        currX, currY = state
        routes.append([currX[0], currY[0]])

        # Failure Conditions
        hitObstacle = any([np.isnan(stateCoord[0])
                    for stateCoord in state])
        atLocalMinima = ((np.linalg.norm(interp_gradient) < minimaTol) and
                    not isAtGoal(np.array([currX, currY])))
        outOfIterations = iterCount >= num_iterations

        if hitObstacle or atLocalMinima or outOfIterations:
            if(hitObstacle):
                print("Hit Obstacle")
            if(atLocalMinima):
                print("atLocalMinima")
            if(outOfIterations):
                print("outOfIterations")
            return False

    return routes

# Separate Functions had to be setup to implement used for the larger workspace
def calc_potential_field_large_workspace(q_start, q_goal, obs, NUMS, DSTARGOAL, ATTRACT_GAIN, REPUL
SIVE_GAIN, Q_STAR):
    """
    Calculates the potential of the all the states in the grid
    PARAMETERS
    ----------
    q_start : Start position of the point robot
    q_goal : Goal Position
    obs : List of Shapely.Geometry.MultiPolygon
    NUMS: Grid Size
    DSTARGOAL : Hyper parameter
    ATTRACT_GAIN : Hyper Parameter
    REPULSIVE_GAIN : Hyper Parameter
    Q_STAR : Hyper parameter
    """
    x_coor = np.arange(XLIMIN, XLIMAX, 0.25)
    y_coor = np.arange(XLIMIN, XLIMAX, 0.25)
    Y, X = np.meshgrid(x_coor, y_coor)
```

```
    # potential = 0
    nCoordsX = len(x_coor)
    nCoordsY = len(y_coor)

    U = np.zeros((nCoordsX, nCoordsY))
    points = []

    for i_x, x in enumerate(x_coor):
        for i_y, y in enumerate(y_coor):
            state = np.array([x, y])
            ptentl = potential(state, obs, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR, q_goal)
            U[i_y, i_x] = ptentl
            points.append((x, y))

    fig = plt.figure()
    ax = fig.gca(projection='3d')
    x_len = x_coor.shape[0]
    y_len = y_coor.shape[0]
    ax.plot_surface(X, Y, U)
    plt.show()

    return U, points

def gradientDescent_field_large_workspace(x, y, obs, q_start, q_goal, DSTARGOAL, ATTRACT_GAIN, REPULS
IVE_GAIN, Q_STAR):
    """
    Gradient Descent using the Potential Field generated at the top
    x: Numpy array of the possible x direction of the point robot
    y: Numpy array of the possible y direction of the point robot
    """
    # Tolerance
    minimaTol = 1e-4

    # max iterations
    num_iterations = 3000000

    # robot = self.robot
    U, points = calc_potential_field_large_workspace(q_start, q_goal, obs, 0.25,\
        DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR)
    print("Entering Into Gradient Descent ")
    routes = []
    state = q_start
    currX, currY = 0,0

    # Appends the Route, state of the robot
    routes.append([currX, currY])

    # calculate the gradient on the CSpace grid
    dy, dx = np.gradient(U)

    dx_values = dx.flatten(order='F')
    dy_values = dy.flatten(order='F')

    # Initiation
```

```
    iterCount = 0
    updateRate = 20
    while not isAtGoal(np.array([currX, currY]), q_goal):

        interp_gradient = np.zeros((2, 1))
        currX, currY = state

        # For Smooth Gradient Descent
        interp_dx = griddata(points, dx_values, (currX, currY), method='cubic')
        interp_dy = griddata(points, dy_values, (currX, currY), method='cubic')
        interp_gradient[0] = 0.002 * interp_dx
        interp_gradient[1] = 0.002 * interp_dy

        print('state: ', state, 'dx: ', interp_dx, 'dy:', interp_dy)

        # Data Logging
        shouldPrint = (iterCount % updateRate == 0)
        if shouldPrint:
            routes.append([round(currX[0], 1), round(currY[0], 1)])

        # Updates State
        iterCount += 1

        state -= interp_gradient
        currX, currY = state
        routes.append([currX[0], currY[0]])

        # Failure Conditon
        hitObstacle = any([np.isnan(stateCoord[0])
                          for stateCoord in state])
        atLocalMinima = ((np.linalg.norm(interp_gradient) < minimaTol) and
                          not isAtGoal(np.array([currX, currY])))
        outOfIterations = iterCount >= num_iterations

        if hitObstacle or atLocalMinima or outOfIterations:
            if(hitObstacle):
                print("Hit Obstacle")
            if(atLocalMinima):
                print("atLocalMinima")
            if(outOfIterations):
                print("outOfIterations")
            return False

    return routes

def plotPotentialField():

    c = input("Enter '0' for 2-obstacle(config 1) , '1' for 5-obstacle(config 2) , '2' for 9-obstacle(config 3): ")
    c = int(c)

    if c==0:
        # routes = [[0, 0], [1.98959, -0.00197], [3.22019, -0.57861], [4.74161, -0.84644], [5.6343, -0.7471], \
        # [5.74897,-0.60972 ], [5.70093, -0.33053], [5.81752,-0.13003], [6.07758, 0.17838], [6.65898, 0.32788], \
        #     [7.35296, 0.29694], [8.19067, 0.38579], [8.5821, 0.29594], [8.89755, 0.23071], [9.14425,0.17859 ],\
```

```
#        [9.48265, 0.10806], [9.59782, 0.08396], [9.68742, 0.0652]]
    DSTARGOAL  = 8
    ATTRACT_GAIN = 10
    REPULSIVE_GAIN = [100,100]
    Q_STAR = [2, 1]
    XLIMIN = -10
    XLIMAX = 40
    NUMS = 50
    obs = WORKSPACE_CONFIG['WO3']
    q_start = WORKSPACE_CONFIG['start_pos']
    q_goal = WORKSPACE_CONFIG['WO3_goal']
    q_start = q_start.tolist()
    q_goal = q_goal.tolist()

    U, points = calc_potential_field(q_start, q_goal, obs, NUMS, DSTARGOAL, ATTRACT_GAIN, REPULSIVE
_GAIN, Q_STAR)
    x = np.arange(XLIMIN, XLIMAX, 0.25)
    y = np.arange(XLIMIN, XLIMAX, 0.25)
    routes = gradientDescent(x, y, obs, q_start, q_goal, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN,
Q_STAR)
    print("ROUTES")
    print(routes)
    potField = U

    fig = plt.figure()
    ax = fig.add_subplot(111)

    xGrid = yGrid = NUMS

    x = np.arange(XLIMIN, XLIMAX, 0.25)
    y = np.arange(XLIMIN, XLIMAX, 0.25)
    dy, dx = np.gradient(potField)

    N = 50

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_axisbelow(True)

    cs = ax.contourf(x, y, potField, N, alpha=0.7)
    plt.quiver(x, y, dx, dy, units='width')
    cbar = fig.colorbar(cs, ax=ax, orientation="vertical")
    cbar.ax.set_ylabel('potential function value')

    for obst in obs:
        x,y = obst.exterior.xy
        plt.plot(x,y, color='black', linewidth=0.5)


    val_x = [x[0] for x in routes]
    val_y = [y[1] for y in routes]
    plt.plot(val_x, val_y, color='red', marker='*', linestyle='none', linewidth=1, markersize=0.2, label='Robot path')

    # plotting the start / end location of the robot
```

```
        plt.plot(q_start[0], q_start[1],
                 color='green', marker='o', linestyle='none',
                 linewidth=2, markersize=1,
                 label='Starting State')

        plt.plot(q_goal[0], q_goal[1],
                 color='red', marker='x', linestyle='none',
                 linewidth=4, markersize=4,
                 label='Goal State')


        ax.set_aspect('equal')
        ax.set_xlim(-20, 40)
        ax.set_ylim(-20, 40)

        fig = plt.gcf()
        fig.show()
        plt.show()

    elif c==1:
        # routes = [[0, 0], [0.47, 0.47], [0.65, 0.65], [0.71, 0.64], [0.82, 0.62], [1.35, 0.56] ,[0.89, 0.61],[1.09, 0.59], [1.4
3, 0.88], [1.93, 1.2] ,[2.27, 1.63],[3.26, 2.5], [4.04, 3.35], [4.88, 4.11],[4.98, 5.39], [5.42, 5.81], [6.17, 7.04], [7.56, 8.
12],[8.35, 8.73] ,[8.88, 9.14], [9.38, 9.52], [9.58, 9.67], [9.71, 9.78], [9.76, 9.82]]
        DSTARGOAL = 3
        ATTRACT_GAIN = 9.2
        REPULSIVE_GAIN = [2, 2, 2, 2, 2]
        Q_STAR = [2.7, 0.5, 1.2, 1.2, 5]
        XLIMIN = -10
        XLIMAX = 40
        NUMS = 50
        obs = WORKSPACE_CONFIG['WO1']
        q_start = WORKSPACE_CONFIG['start_pos']
        q_goal = WORKSPACE_CONFIG['WO1_goal']
        q_start = q_start.tolist()
        q_goal = q_goal.tolist()

        U, points = calc_potential_field(q_start, q_goal, obs, NUMS, DSTARGOAL, ATTRACT_GAIN, REPULSIVE
_GAIN, Q_STAR)
        x = np.arange(XLIMIN, XLIMAX, 0.25)
        y = np.arange(XLIMIN, XLIMAX, 0.25)
        routes = gradientDescent(x, y, obs, q_start, q_goal, DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN,
Q_STAR)
        print("ROUTES")
        print(routes)
        potField = U

        fig = plt.figure()
        ax = fig.add_subplot(111)

        xGrid = yGrid = NUMS

        x = np.arange(XLIMIN, XLIMAX, 0.25)
        y = np.arange(XLIMIN, XLIMAX, 0.25)
        dy, dx = np.gradient(potField)
```

```
    N = 50

    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_axisbelow(True)

    cs = ax.contourf(x, y, potField, N, alpha=0.7)
    plt.quiver(x, y, dx, dy, units='width')
    cbar = fig.colorbar(cs, ax=ax, orientation="vertical")
    cbar.ax.set_ylabel('potential function value')

    for obst in obs:
        x,y = obst.exterior.xy
        plt.plot(x,y, color='black', linewidth=0.5)


    val_x = [x[0] for x in routes]
    val_y = [y[1] for y in routes]
    plt.plot(val_x, val_y, color='red', marker='*', linestyle='none', linewidth=1, markersize=0.2, label='Robot path')

    # plotting the start / end location of the robot
    plt.plot(q_start[0], q_start[1],
             color='green', marker='o', linestyle='none',
             linewidth=2, markersize=1,
             label='Starting State')

    plt.plot(q_goal[0], q_goal[1],
             color='red', marker='x', linestyle='none',
             linewidth=4, markersize=4,
             label='Goal State')


    ax.set_aspect('equal')
    ax.set_xlim(-20, 40)
    ax.set_ylim(-20, 40)

    fig = plt.gcf()
    fig.show()
    plt.show()

elif c==2:
    DSTARGOAL  = 10 #6
    ATTRACT_GAIN = 0.1 #5
    REPULSIVE_GAIN = [2, 2, 7, 100 ,7, 7, 7, 7, 5, 5]
    Q_STAR = [4, 4, 6, 10 ,0.1, 0.1, 0.1, 0.1, 0.1, 0.1]
    XLIMIN = -10
    XLIMAX = 40
    NUMS = 40
    obs = WORKSPACE_CONFIG['WO2']
    q_start = WORKSPACE_CONFIG['start_pos']
    q_goal = WORKSPACE_CONFIG['WO2_goal']
    q_start = q_start.tolist()
    q_goal = q_goal.tolist()
```

```
U, points = calc_potential_field_large_workspace(q_start, q_goal, obs, NUMS, DSTARGOAL,\
    ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR)
x = np.arange(XLIMIN, XLIMAX, 0.25)
y = np.arange(XLIMIN, XLIMAX, 0.25)
routes = gradientDescent_field_large_workspace(x, y, obs, q_start, q_goal,\
    DSTARGOAL, ATTRACT_GAIN, REPULSIVE_GAIN, Q_STAR)
print("ROUTES")
print(routes)
potField = U

fig = plt.figure()
ax = fig.add_subplot(111)

xGrid = yGrid = NUMS

x = np.arange(XLIMIN, XLIMAX, 0.25)
y = np.arange(XLIMIN, XLIMAX, 0.25)
dy, dx = np.gradient(potField)

N = 50

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_axisbelow(True)

cs = ax.contourf(x, y, potField, N, alpha=0.7)
plt.quiver(x, y, dx, dy, units='width')
cbar = fig.colorbar(cs, ax=ax, orientation="vertical")
cbar.ax.set_ylabel('potential function value')

for obst in obs:
   x,y = obst.exterior.xy
   plt.plot(x,y, color='black', linewidth=0.5)


val_x = [x[0] for x in routes]
val_y = [y[1] for y in routes]
plt.plot(val_x, val_y, color='red', marker='*', linestyle='none', linewidth=1, markersize=0.2, label='Robot path')

# plotting the start / end location of the robot
plt.plot(q_start[0], q_start[1],
         color='green', marker='o', linestyle='none',
         linewidth=2, markersize=1,
         label='Starting State')

plt.plot(q_goal[0], q_goal[1],
         color='red', marker='x', linestyle='none',
         linewidth=4, markersize=4,
         label='Goal State')


ax.set_aspect('equal')
ax.set_xlim(-20, 40)
```

```
        ax.set_ylim(-20, 40)

        fig = plt.gcf()
        fig.show()
        plt.show()


def main():
    """
    Plots the Path and Vector of the Potential Gradient as a Color Plot
    # Fair Warning
    # Takes a ridiculous amount of time for gradient descent, additonally the gradient descent update had to be kept
    # Really low (alpha) for optimum results but at the expense of time (a lot of time, in hours)
    """

    plotPotentialField()


if __name__ == '__main__':
    main()
```

==
wavefront.py
==

```
"""
Implementation of the Wave Front , brush fire Algorithm

I would like to thank Professor Morteza Lahijanian and the fellow course mate Kedar More for
discussion during the Implementation of this code

"""

import numpy as np
from math import pi
from shapely.geometry import Point, Polygon, MultiPolygon, LineString
import matplotlib.pyplot as plt
from scipy.interpolate import griddata
from workspaces import config


WORKSPACE_CONFIG = config()

# Simulation parameters
limit = 200

class Robot(object):
    """
    This Class is helper class for plotting and mainipulating which keeps track the end points.
    @param - arm_lengths : Array of the Lengths of each arm
    @param - motor_angles : Current Angle of the Each Revolute Joint in the Global Frame
    """
    def __init__(self, arm_lengths, motor_angles):
        # Initialization with a specific parameter
        self.arm_lengths = np.array(arm_lengths)
```

```
        self.motor_angles = np.array(motor_angles)
        self.link_end_pts = [[0, 0], [0, 0], [0, 0]]
        # Find the Location of End Points of each Link
        for i in range(1, 3):
            # Follows Forward Kinematic Update Steps Analysis
            self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
                np.cos(np.sum(self.motor_angles[:i]))
            self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
                np.sin(np.sum(self.motor_angles[:i]))

        self.end_effector = np.array(self.link_end_pts[2]).T

    def update_joints(self, motor_angles):
        """
        Update the Location of the end points of the link, Based on Updates of the End points
        """
        self.motor_angles = motor_angles
        # Forward Kinematic Update and storage of link_length data
        for i in range(1, 3):
            self.link_end_pts[i][0] = self.link_end_pts[i-1][0] + self.arm_lengths[i-1] * \
                np.cos(np.sum(self.motor_angles[:i]))
            self.link_end_pts[i][1] = self.link_end_pts[i-1][1] + self.arm_lengths[i-1] * \
                np.sin(np.sum(self.motor_angles[:i]))

        self.end_effector = np.array(self.link_end_pts[2]).T

def make_grid(C_xSpace, C_ySpace, q_goal, obs):
    """
    Initialization of the Grid of specified Grid Size
    PARAMETERS
    ----------
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    q_goal : Goal Location
    obs : List of Shapely.Geometry.MultiPolygon
    """
    gridX = len(C_xSpace)
    gridY = len(C_ySpace)

    # Initializing the Grid to 0
    grid=np.zeros((gridX,gridY))

    for i in range(gridX):
        for j in range(gridY):
            if (C_xSpace[i] == q_goal[0]) and (C_ySpace[j] == q_goal[1]):
                grid[i][j]=2

            for obstacle in obs:
                distToObst = abs(obstacle.exterior.distance(Point((C_xSpace[i], C_ySpace[j]))))
                if distToObst == 0:
                    grid[i][j]=1

    return grid
```

```
def boundary_condition_helper(grid, curr_status_num, i, j, gridX, gridY):
    """
    Helper Function to update grid values based on the curr_states of the grid value
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_status_num : Current Grid Value
    i : X location
    j : Y location
    gridX : Length of the GRid X direction
    gridY : Length of the GRid Y direction
    """

    # Helper Variables
    rows = [i+1, i-1]
    cols = [j+1, j-1]

    # Keeps it within bounds
    temp1 = rows[0] % gridX
    temp2 = cols[0] % gridY

    # For sanity check
    temp3 = max(rows[1],0)
    temp4 = max(cols[1],0)
    temp5 = curr_status_num + 1

    # Horizontal Facets
    if grid[temp1][j]==0 and rows[0] < gridX:
        grid[rows[0]][j] = temp5
    if grid[temp3][j]==0 and rows[1]:
        grid[rows[1]][j] = temp5

    # Vertical Facets
    if grid[i][temp2] == 0 and cols[0] < gridY:
        grid[i][cols[0]] = temp5
    if grid[i][temp4] == 0 and cols[1] :
        grid[i][cols[1]] = temp5

    return grid

def grid_update(grid,curr_grid_number, C_xSpace, C_ySpace, q_start):
    """
    Updates the grid Based on the current status of the grid
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_grid_number: Current Value of the grid value
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    q_start : Start Location
    """

    # Initialization
    gridX = len(C_xSpace)
```

```
    gridY = len(C_ySpace)


    # O(n2) implementation to dig through all the grid values
    for i in range(gridX):
        for j in range(gridY):
            if grid[i][j]==curr_grid_number:
                # Success Condition
                if C_xSpace[i] == q_start[0][0] and C_ySpace[j] == q_start[1][0]:
                    return grid, curr_grid_number
                else:
                    # Continue Updating The Grid
                    boundary_condition_helper(grid, curr_grid_number, i, j, gridX, gridY)
    return grid, None

def planning_path(grid,curr_grid_number, C_xSpace, C_ySpace):
    """
    Finds the Route Post the Grid Update
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_grid_number : Current Grid Number
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    """
    # Initial Flag Setup
    flag=curr_grid_number+10

    # Finds the Start Location
    i=np.argwhere(C_xSpace==0)[0][0]
    j=np.argwhere(C_ySpace==0)[0][0]

    # Updates the Start for approprate flag
    # Also necessary to showcase output as
    # plt.imshow() is used
    grid[(i)][j]=flag

    # Begin Calculation
    distance=0

    # While not Goal is reached
    while curr_grid_number>2:
        status = False
        # Right
        if grid[(i+1)][j]==curr_grid_number-1:
            i+=1
            status = True

        # Left
        elif grid[(i-1)][j]==curr_grid_number-1:
            i-=1
            status = True

        # Top
```

```
        elif grid[(i)][j+1]==curr_grid_number-1:
            j+=1
            status = True

        # Bottom
        elif grid[(i)][j-1]==curr_grid_number-1:
            j-=1
            status = True

        # Update Appropriate Flags
        if status:
            # Grid size is 0.5
            distance+=0.25
            grid[(i)][j]=flag
            # Update the grid number
            curr_grid_number-=1

    return distance


# Manipulator
def manipulator_cspace(C_xSpace, C_ySpace, q_goal, obs):
    """
    Gives the Grid of C Space Generated for a 2 link manipulator
    PARAMETERS
    ----------
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    q_goal : Goal Location
    obs : List of Shapely.Geometry.MultiPolygon
    """
    # Initialization
    gridX = len(C_xSpace)
    gridY = len(C_ySpace)

    # Initialize with 0's
    grid=np.zeros((int(gridX),int(gridY)))

    # Setup for link length as 1
    arm_lengths = [float(1.0), float(1.0)]
    # Begin Setup with motor angles to [0,0]
    motor_angles = np.array([0] * 2)

    obstacles = obs
    temp = []
    temp_status = False
    plt.ion()
    plt.show(block=False)
    arm = Robot(arm_lengths, motor_angles)

    # Rotate through all the possible angles of the Link
    theta_np_array = np.radians(np.arange(0, 365, 5))
    for i in range(gridX):
        for j in range(gridY):
```

```
        # Updates the Motor Joints for every 5 degree update
        arm.update_joints([theta_np_array[i], theta_np_array[j]])
        link_end_pts = arm.link_end_pts

        collision_detected = False

        # Checks if it intersects the obstacle
        for k in range(len(link_end_pts) - 1):
            for obstacle in obstacles:
                # Create a line segment
                line_seg = [link_end_pts[k], link_end_pts[k + 1]]
                line = LineString([link_end_pts[k], link_end_pts[k + 1]])
                collision_detected = line.intersects(obstacle)
                if collision_detected:
                    break
            if collision_detected:
                break
        # Updates it 1 if it intersects
        grid[i][j] = int(collision_detected)

    # Hard Coding the goal location
    grid[36][0] = 2

    return grid

def boundary_condition_helper_manipulator(grid, curr_grid_number, i, j, gridX, gridY):
    """
    Helper Function to update grid values based on the curr_states of the grid value
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_status_num : Current Grid Value
    i : X location
    j : Y location
    gridX : Length of the GRid X direction
    gridY : Length of the GRid Y direction
    """

    # Helper Variables
    rows = [i+1, i-1]
    cols = [j+1, j-1]

    # Keeps it within bounds of 0 and 360 degrees
    # Resets 360 to 0 degrees
    temp1 = (rows[0]) % (gridX-1)
    temp2 = (rows[1]) % (gridX-1)

    # For sanity check
    temp3 = (cols[0]) % (gridY-1)
    temp4 = (cols[1]) % (gridY-1)
    temp5 = curr_grid_number+1

    # Horizontal Facets
    if grid[temp1][j] == 0:
```

```
        grid[temp1][j] = temp5
    if grid[temp2][j] == 0:
        grid[temp2][j] = temp5

    # Vertical Facets
    if grid[i][temp3] == 0:
        grid[i][temp3] = temp5
    if grid[i][temp4] == 0:
        grid[i][temp4] = temp5

    return grid

def grid_update_manipulator(grid, curr_grid_number, C_xSpace, C_ySpace, q_start):
    """
    Updates the grid Based on the current status of the grid
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_grid_number: Current Value of the grid value
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    q_start : Start Location
    """

    # Initialization
    gridX = len(C_xSpace)
    gridY = len(C_ySpace)

    # O(n2) implementation to dig through all the grid values
    for i in range(gridX):
        for j in range(gridY):
            if grid[i][j]==curr_grid_number:
                # Goal Conditon
                if [round(C_xSpace[i],2), round(C_ySpace[j],2)] == [0.00,0.00] or \
                    [round(C_xSpace[i],2), round(C_ySpace[j],2)] == [6.28,6.28]:
                    return grid, curr_grid_number
                else:
                    # Boundary Conditon
                    grid = boundary_condition_helper_manipulator(grid, curr_grid_number, i, j, gridX, gridY)

    plt.imshow(grid, origin = 'lower')
    plt.show()
    return grid, None

def planning_path_manipulator(grid,curr_grid_number, C_xSpace, C_ySpace):
    """
    Finds the Route Post the Grid Update
    PARAMETERS
    ----------
    grid : np.meshgrid
    curr_grid_number : Current Grid Number
    C_xSpace : Numpy Array of the Possible X locations
    C_ySpace : Numpy Array of the Possible Y locations
    """
```

```
# Val stores the Location
val = []
gridX = len(C_xSpace)-1
gridY = len(C_ySpace)-1

# Initial Flag Setup
flag = curr_grid_number + 10

# Start Location
i = 0
j = 0

# Updates the Start for appropprate flag
# Also necessary to showcase output as
# plt.imshow() is used
grid[(i)][j]=flag

# Ready, Set, Go
distance=0

# While Goal is not reached
while curr_grid_number>2:
    status = False

    # Right
    if grid[(i+1) % gridX][j]==curr_grid_number-1:
        status = True
        i = (i+1) % gridX

    # Left
    elif grid[(i-1) % gridX][j]==curr_grid_number-1:
        status = True
        i = (i-1) % gridX

    # Top
    elif grid[(i)][(j+1) % gridY]==curr_grid_number-1:
        status = True
        j = (j+1) % gridY

    # Bottom
    elif grid[(i)][(j-1) % gridY]==curr_grid_number-1:
        status = True
        j = (j-1) % gridY

    # Update Flags
    if(status):
        # In degrees
        distance += 5
        val.append([i, j])
        grid[(i)][j]=flag
        plt.imshow(grid, origin = 'lower')
        plt.show()
        # Update the grid number
```

```
            curr_grid_number -= 1

    return distance, val

def forw_K(motor_angles):
    """
    Function to Calculate the forward kinematics.
    """
    pos_x_link1 = 0
    pos_y_link1 = 0
    pos_x = 0
    pos_y = 0
    # Simple logic gets the calculates the End Effector position
    # from the current motor angle and position
    for i in range(1, 3):
        pos_x += 1.0 * np.cos(np.sum(motor_angles[:i]))
        pos_y += 1.0 * np.sin(np.sum(motor_angles[:i]))

    pos_x_link1 += 1.0 * np.cos(np.sum(motor_angles[:1]))
    pos_y_link1 += 1.0 * np.sin(np.sum(motor_angles[:1]))


    # Transpose is necessary, for future updates
    return pos_x_link1, pos_y_link1, pos_x, pos_y

def manipulator_plotter(val, obs, q_goal):
    """
    Helper Function to Plot the 2 link manipulator
    """
    plt.figure(2)
    for idx, angle in enumerate(val):
        # Plots every 10 iteration
        if(idx % 10 == 0):
            angle = np.asarray(angle)*5
            mtr_angles = np.radians(angle)

            # Results from Forward Kinematics
            pos_x_link1, pos_y_link1, pos_x_link2, pos_y_link2 = forw_K(mtr_angles)

            plt.plot([0.0, pos_x_link1, pos_x_link2],\
                        [0.0, pos_y_link1, pos_y_link2], 'c-')

            plt.plot(pos_x_link1, pos_y_link1, 'ko', markersize=2)
            plt.plot(pos_x_link2, pos_y_link2, 'ro')

            for obstacle in obs:
                plt.plot(*obstacle.exterior.xy)

    # Mark the goal Position
    plt.plot(q_goal[0],q_goal[1], 'rx')
    plt.xlim([-3, 3])
    plt.ylim([-3, 3])
    plt.draw()
    plt.pause(100)
```

```
    plt.show()

def main():

    m1 = input("0 for non-manipulator , 1 or Manipulator : ")

    if(int(m1) == 0):
    # # Non- Manipulator
        cfg = input(" Enter 1 for config 1 , 2 for config 2: ")
        cfg = int(cfg)
        if cfg == 1:
            XLIMIN = -10
            XLIMAX = 40
            YLIMIN = -20
            YLIMAX = 20
            obs = WORKSPACE_CONFIG['WO1']
            q_start = WORKSPACE_CONFIG['start_pos']
            q_goal = WORKSPACE_CONFIG['WO1_goal']
            q_start = q_start.tolist()
            q_goal = q_goal.tolist()

            C_xSpace = np.arange(XLIMIN, XLIMAX, 0.25)
            C_ySpace = np.arange(XLIMIN, XLIMAX, 0.25)
            grid = make_grid(C_xSpace, C_ySpace, q_goal, obs)
            fig,ax=plt.subplots()
            curr_grid_number=2
            while True:
                updated_grid, val =grid_update(grid,curr_grid_number, C_xSpace, C_ySpace, q_start)
                if val is not None:
                    # Success Condition
                    grid,curr_grid_number=updated_grid, val
                    break
                else:
                    # Continue to Update GRid
                    grid=updated_grid
                    curr_grid_number=curr_grid_number+1
                    pass


            dist=planning_path(grid,curr_grid_number, C_xSpace, C_ySpace)
            print("Total distace of the path is:",dist)
            plt.imshow(grid, origin='lower')
            plt.show()

        elif cfg == 2:
            XLIMIN = -10
            XLIMAX = 40
            YLIMIN = -8
            YLIMAX = 8
            obs = WORKSPACE_CONFIG['WO2']
            q_start = WORKSPACE_CONFIG['start_pos']
            q_goal = WORKSPACE_CONFIG['WO2_goal']
            q_start = q_start.tolist()
            q_goal = q_goal.tolist()
```

```
        C_xSpace = np.arange(XLIMIN, XLIMAX, 0.25)
        C_ySpace = np.arange(XLIMIN, XLIMAX, 0.25)
        grid = make_grid(C_xSpace, C_ySpace, q_goal, obs)
        fig,ax=plt.subplots()
        curr_grid_number=2
        while True:
            updated_grid, val =grid_update(grid,curr_grid_number, C_xSpace, C_ySpace, q_start)
            if val is not None:
                # Success Condition
                grid,curr_grid_number=updated_grid, val
                break
            else:
                # Continue to Update GRid
                grid=updated_grid
                curr_grid_number=curr_grid_number+1
                pass


        dist=planning_path(grid,curr_grid_number, C_xSpace, C_ySpace)
        print("Total distace of the path is:",dist)
        plt.imshow(grid.T, origin='lower')
        plt.show()

else:
    # Manipulator
    obs = WORKSPACE_CONFIG['WO4']
    q_start = WORKSPACE_CONFIG['manip_start_pos']
    q_goal = WORKSPACE_CONFIG['manip_goal_pos']
    q_start = q_start.tolist()
    q_goal = q_goal.tolist()

    C_xSpace = np.arange(0, 365, 5)
    C_ySpace = np.arange(0, 365, 5)
    grid = manipulator_cspace(C_xSpace, C_ySpace, q_goal, obs)

    fig,ax = plt.subplots()
    curr_grid_number = 2
    while True:
        updated_grid, val = grid_update_manipulator(grid, curr_grid_number, C_xSpace, C_ySpace, q_start)
        if val is not None:
                # Success Condition
                grid,curr_grid_number=updated_grid, val
                break
        else:
            # Continue to Update GRid
            grid=updated_grid
            curr_grid_number=curr_grid_number+1
            pass

    dist, val = planning_path_manipulator(grid,curr_grid_number, C_xSpace, C_ySpace)
    # plt.pause(100)
    manipulator_plotter(val, obs, q_goal)
    print("Total distace of the path is:",dist)
```

```
        plt.imshow(grid, origin = 'lower')
        plt.show()


if __name__ == '__main__':
    main()
```