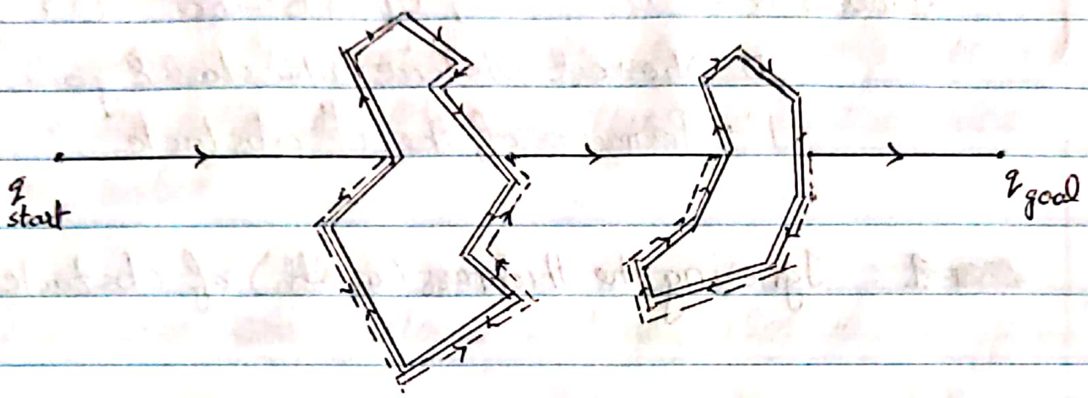
NAME: ARPIT SAVARKAR

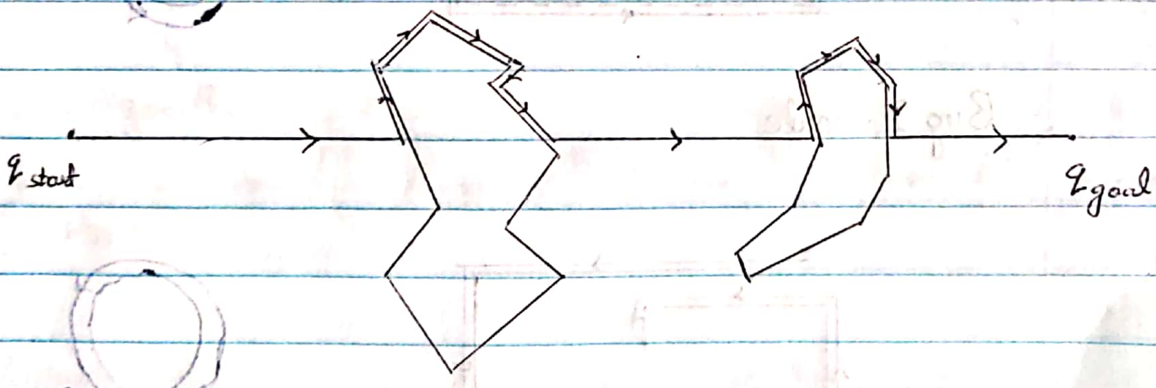
HW-1

Q.1

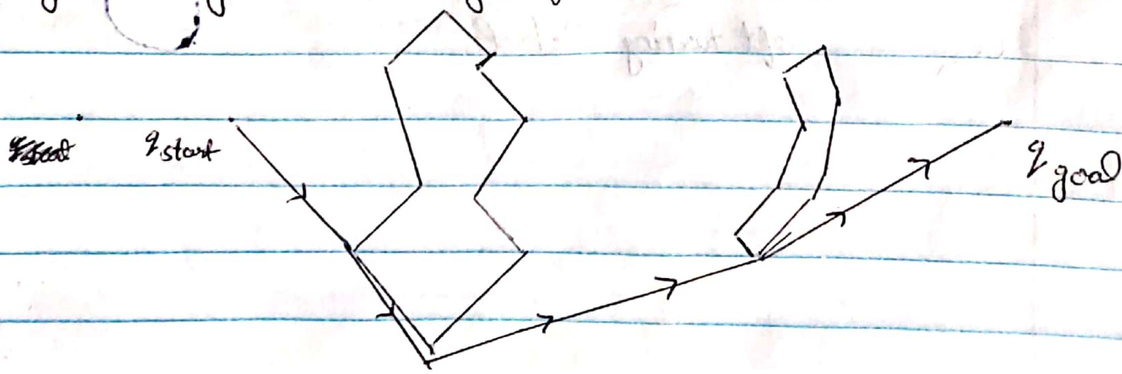
Ans Trajectories undertaken by Bug-1 (Assuming left turning Robot)



Trajectory undertaken by Bug-2



Tangent Bug (Assuming infinite radius)



Q.2

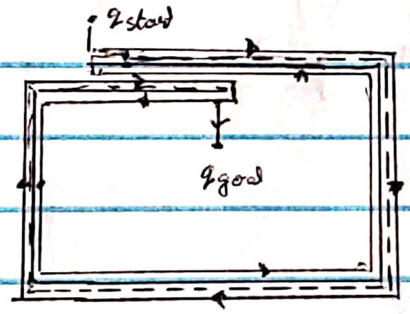
Ans The bounds on path length the point object takes

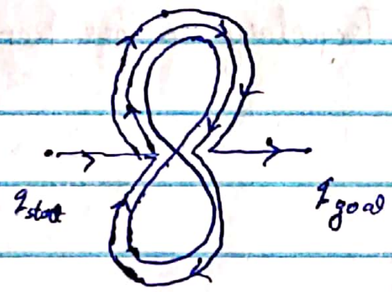
Lower Bound : D , Upper Bound : $D + \frac{3}{2} \sum P_i$

D: Shortest Distance b/w start & goal

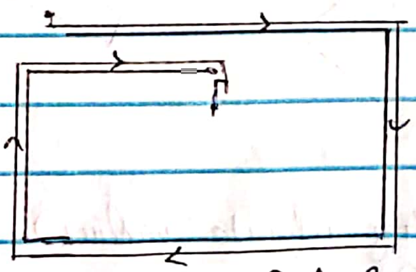
P_i: Perimeter of the i^{th} obstacle

: Ignoring the thickness (width) of obstacle

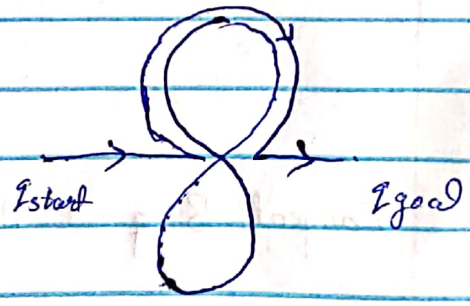




Bug 2 - route



Left turning Robot

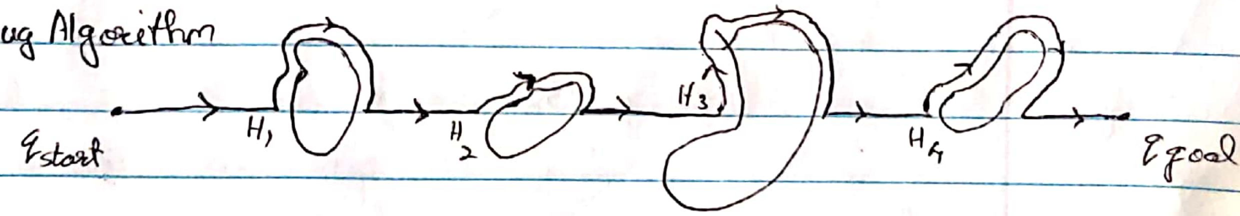


Q.3

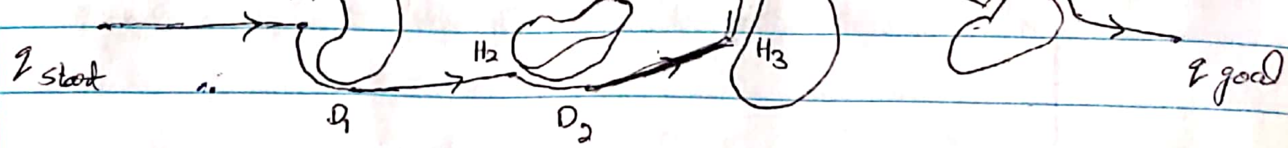
Ans Sensed obstacle by the robot is called followed obstacle. The value $d_{followed}$ is the shortest distance between the boundary which had been sensed and the goal. The value d_{reach} is the distance b/w the goal and the closest point on the followed obstacle that is within line of sight of the robot.

The robot ~~two~~ follows 2 stages "Motion-to-goal" and "Boundary Following". When sensor range is zero, then the point of local minima (closest point sensed by the robot on the obstacle to the goal) is the same as the hit point from the Bug 1 & Bug 2 algorithms.

Bug Algorithm



Tangent Bug

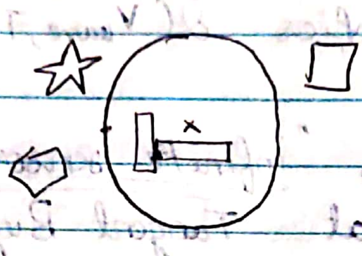


The robot would continues motion-to-goal until it hits a hit point. Assuming a left turning robot. ~~It~~ changes to ~~changes~~ from motion to goal to boundary following when it comes to Hit point. As it is a zero range sensor, d_{reach} is the distance b/w the robot & the goal & the process continues.

Scanned with CamScanner

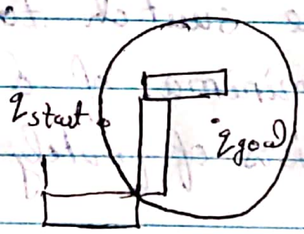
Q.4

Ans Case A: When "m" obstacles are completely inside the circle formed by q_{goal} & q_{start}



If this is the case then the max number of obstacles it will encounter is "m"

Case-B: If a condition exists such that an obstacle lies along the circumference



Then the maximum number of obstacles the robot can encouter can be "n"

The above cases assumes the boundary of the formed by q_{goal} & q_{start} does not form an obstacle

Q.5

Ans

Q.5

Ans let, $\delta_{min}(T)$, : min. Distance Recorded to the target
The robot leaves the obstacle boundary when there is a node V_{leave} that satisfies $\delta(V_{leave}, T) < \delta_{min}(T)$

- "If the bug has "infinite sensor range", then it can be proved that Tangent Bug Algorithm always terminates after following a finite - length path

⇒ Proof

Every switch point "P_i" is associated with a local minima point M_i of $\delta(w, T)$ which decreases at successive Local min points. Thus M_i is associated with at most one switch to boundary following. Thus no. of local minima of $\delta(w,T)$ is finite. Thus the path consists of finitely many boundary-following segments.
Secondly, path length of each motion-toward-the target is finite. Thirdly, the path length of each boundary following segment is finite.

TO PROVE : Using Unlimited Sensor Range, TangentBug always finds the target if it is reachable from the start point.

As there are only finitely many boundaries segments:
If "Target" is reachable from the starting point. It is for certain that leaving condition will cause the robot to terminate boundary following and thus leave the obstacles

Every trajectory is followed by a transition phase, & thus it will always have a last transition phase. Thus, the last termination phase is either followed by the last motion-toward-the target or is already at the goal. Thus it has an upper bound

~~If $\vec{v}$ is a vector~~

$$L_{max} = \left(1 + \frac{1}{\gamma}\right) \|S-T\| + \sum \#Minima_i \times (P_i + \rho)$$

where γ is a small +ve parameter,

$\|S-T\|$ = Disc ~~with~~ radius of the disc

$\#Minima_I$: No. of local minima $d(w,T)$ on a obstacle

ρ : Range sensor limitations

P_i : Perimeter of the i^{th} obstacle

Q.6

Ans Tangent Bug is a "Complete" algorithm if a path exists to the goal.

While true do

repeat

1) Continiously move towards the pt. $n \in \{T, O_i\}$ which minimizes $d(x, n) + d(n, q_{goal})$

Until

(i) The goal is encountered or

(ii) ~~If you hit the goal~~

(ii) If goal is ~~found~~ hit, then backoff few steps & change orientation by 180° & continue traversing the same path

(iii) If the heuristic distance tends to increase then move to boundary following & and continue

Repeat

2) Continiously update d_{reach}, $d_{followed}$, & $\{O_i\}$

3) Continiously moves towards $n \in \{O_i\}$ that is in the boundary direction

Until

4) The goal is reached

5) The robot ~~moves~~ completes a cycle & no path exists

6) $d_{reach} < d_{followed}$

End While

Q.7

a)

Ans Attached in the Zip file

b)

Ans Path lengths Generated

	Bug 1	Bug 2
Workspace - 1	102 units	19.8 units
Workspace - 2	315.6 units	41.2 units

c)

Ans No, Lengths ~~wo~~ generated would be different.

==
Name : Arpit Savarkar
Bug Algorithms
CODE : Q7
Ans :

===
<algo_bug.py>

```
    """
    RESULTS
    -------
    Workspace -1
        Bug 1 : Length : 102 uints
        Bug 2 : 19.8 units
    Workspace -2
        Bug 1 : Length : 315.6 uints
        Bug 2 : Length : 41.2 uints
    """
```

===

```
import numpy as np
import matplotlib.pyplot as plt
import sys
import time
import polytope as pc
import scipy
import types
from scipy.spatial import Delaunay as Del
from workspaces import config
from operator import eq, add, sub
from helper import *

WORKSPACE_CONFIG = config()
GLOBAL_UPDATE_TIME = 0.01


class bug:

    def __init__(self):
        pass

    def plot_plt(self, ws, start, goal, algo):
        """
        Plots the start, goal and the workspace configuration based
        on user Input
        PARAMETERS
        ----------
            WS = Workspace Configuration to be plotted
            Start = Global Location from where the robot starts
            Goal  = Final Destination of the robot
            Algo = Binary 1/2 to Implemente Bug1/Bug2
        RETURNS
        -------
            None
```

```
    """
    h_polytope = bug.drawobstacles(self, ws)
    if algo == '1':
        print("Implementing Bug 1 Algorithm")
        bug.algo_bug1(self, start, goal, h_polytope)
    elif algo == '2':
        print("Implementing Bug 2 Algorithm")
        bug.algo_bug2(self, start, goal, h_polytope)
    plt.show()

def drawobstacles(self, setup):
    """
    Plots the Obstacles based on the setup from config
    PARAMETERS
    ----------
        SETUP = Workspace Setup to use
            1: 1st Setup
            2: 2nd Setup
    RETURNS
    -------
        A plot of the Polytopes created from the pre-defined points
    """
    if setup == '1':
        print("Workspace 1 Configuration")
        W = WORKSPACE_CONFIG['WO1']
    elif setup == '2':
        print("Workspace 2 Configuration")
        W = WORKSPACE_CONFIG['WO2']

    p = pc.Region(list_poly=W)
    p.plot()

    return W

def collision_status(self, curr, active_polytopes):
    """
    # Delauncy Module for tessellation in N dimensions.
    # pc.extreme(polytope) - Extreme Points of a given polytope
    # find_simplex() - simplices containing the given points.
    Source : https://scipy.github.io/devdocs/generated/scipy.spatial.\
        Delaunay.find_simplex.html#scipy.spatial.Delaunay.find_simplex
    PARAMETERS
    ----------
    curr = Current Global (x,y) position of the point sized robot
    active_polytopes = Polytopes Enabled from the Config
    RETURNS
    -------
        A tuple of collision status and Polytope Hit
    """
    for index, p_tope in enumerate(active_polytopes):

        coll_status = (0,0)
        status = Del(pc.extreme(p_tope)).find_simplex(curr.T)
        if status != -1:
```

```python
            # Returns -1 for all points which do not lie inside the simplexes of polytopes
            coll_status =  (True, index)
            break

    return coll_status[0], coll_status[1]

def setup(self, goal, x, y, t, ActivatedPolytopes):
    """
    Initial Setup before the Algorithm starts implementing Conditional Checks
    PARAMETERS
    ----------
        SETUP = Workspace Setup to use
            1: 1st Setup
            2: 2nd Setup
    RETURNS
    -------
        Pose, PolyotopeHit and next orientation of the robot
    """

    # Orientation of the Robot
    angle = np.arctan2( (goal[1, 0] - y) , (goal[0, 0] - x))

    # Next Location of the robot
    x += t*np.cos(angle)*GLOBAL_UPDATE_TIME
    y += t*np.sin(angle)*GLOBAL_UPDATE_TIME

    # (Collison_Status, Collided_Polytope)
    collide, poly_number = bug.collision_status(self, np.array([[x], [y]]), ActivatedPolytopes)

    # The Index of Polytope which is under intersection
    EnabledPolytope = ActivatedPolytopes[poly_number]
    return x, y, collide, EnabledPolytope, angle

def collisionHelper(self, n, pose_x, pose_y, ActivatedPolytopes):
    """
    Helper Function to check for collison from the existing location
    PARAMETERS
    ----------
        n : TestBed Setup
        (pose_x, pose_y) : Current Pose of the Robot
        ActivatedPolytopes: List of Polytopes in configuration
    RETURNS
    -------
        Tuple (Collision_Status, PolyTope which may be hit from the \
            list of activated polytopes )
    """
    if n == 0:
        a, b = bug.collision_status(self,np.array([[pose_x], \
            [pose_y]]), ActivatedPolytopes)
    elif n == 1:
        a, b = bug.collision_status(self,np.array([[pose_x - 0.1 ],\
            [pose_y + 0.1]]), ActivatedPolytopes)
    elif n== 2:
        a, b = bug.collision_status(self,np.array([[pose_x + 0.1], \
```

```
            [pose_y - 0.1]]), ActivatedPolytopes)
    elif n == 3:
        a, b = bug.collision_status(self,np.array([[pose_x + 0.1],\
             [pose_y + 0.1 ]]), ActivatedPolytopes)
    elif n == 4:
        a, b = bug.collision_status(self,np.array([[pose_x - 0.1],\
             [pose_y - 0.1]]), ActivatedPolytopes)
    elif n == 5:
        a, b = bug.collision_status(self,np.array([[pose_x + 0.1],\
             [pose_y - 0.1]]), ActivatedPolytopes)
    elif n == 6:
        a, b = bug.collision_status(self,np.array([[pose_x + 0.1],\
             [pose_y]]), ActivatedPolytopes)
    elif n == 7:
        a, b = bug.collision_status(self,np.array([[pose_x], \
            [pose_y + 0.1]]), ActivatedPolytopes)
    return a, b

def bug_1_helper(self, EnabledPolytope, limit_x, limit_y, ActivatedPolytopes, flag,
            poseXYList, facet, PerimeterTraversalStatus, V1, V2,
            polytope_number_list, val, path_length):
    """
    Helper Function for Implementation of Bug1- Algorithm
    It Checks the following
    1) Facet Hit of the EnabledPolytope
    2) Next possible location based on the Facet Found
    PARAMETERS
    ----------
        EnabledPolytope: The Polytope which the robot has hit,
        limit_x : The Current Pose_x of the robot,
        limit_y : The Current Pose_x of the robot,
        ActivatedPolytopes : List of all Polytopes in the config,
        flag: Boolean Method to check for vectoriezed direction,
        poseXYList: The points where the robot has hit till now,
        facet: The side hit by the polytope,
        PerimeterTraversalStatus : Flag to check if it has traversed the entire polytope
        V1 : The Discontinuity Vertex over the Facet Hit by the Polytope
        V2: The endpoint Vertex of the facet hit by the polytope
        polytope_number_list : Helper Numbers of polytopes
        val: Temporary Helper Variable (Takes 1 or 2)
        path_lenght: Lenght Traversed till now
    """
    temp = [0,0]

    if type(V1) == type(None):
    # if isinstance(V1, type(None)):
        # Rounding Up is vital for convergence
        for l in [limit_x, limit_y]:
            l = round(l,1)
        # Polytope Hit
        V2 = EnabledPolytope.vertices[facet]

        if facet + 1 > EnabledPolytope.vertices.shape[0] - 1:
            V1 = EnabledPolytope.vertices[0] # Starting Vertex of Facet Hit
```

```
        else:
            V1 = EnabledPolytope.vertices[facet + 1] # End Vertex of Facet Hit

    # Traveral Around the Facet
    while limit_x != V2[0] or limit_y != V2[1]:
        if val == 0: # Flag configuration
            limit_x = V1[0]
            if V1[1] - V2[1] < 0:
                limit_y += 0.1
                direction = False
            else:
                limit_y -= 0.1
                direction = True
        elif val == 1:
            limit_y = V1[1]
            if V1[0] - V2[0] < 0:
                limit_x += 0.1
                direction = True
            else:
                limit_x -= 0.1
                direction = False

        limit_x, limit_y = round(limit_x, 2), round(limit_y, 2)

        #Checking for possibility to move Diagonally Upwards
        if not direction:
            if val == 0:
                coll_status, p_num = bug.collisionHelper(self, 1, \
                    limit_x, limit_y, ActivatedPolytopes)
            elif val == 1:
                coll_status, p_num = bug.collisionHelper(self, 4, \
                    limit_x, limit_y, ActivatedPolytopes)
        else:
        #Checking for possibilitty to move Diagonally Downwards
            if val == 0:
                coll_status, p_num = bug.collisionHelper(self, 2, \
                    limit_x, limit_y, ActivatedPolytopes)
            elif val == 1:
                coll_status, p_num = bug.collisionHelper(self, 3, \
                    limit_x, limit_y, ActivatedPolytopes)

        #If the Robot has collided, check for next possible location and check if that is collided as well
        if coll_status:
            ignore_val , prev_num = bug.collisionHelper(self, 0, \
                limit_x, limit_y, ActivatedPolytopes)
            if val == 0:
                if ActivatedPolytopes[p_num] != ActivatedPolytopes[prev_num] \
                    and (limit_y - 0.1) != V1[1]:
                    break
            elif val == 1:
                if ActivatedPolytopes[p_num] != ActivatedPolytopes[prev_num] \
                    and (limit_x - 0.1) != V1[0]:
                    break
```

```
        #Keep appending these positions
        poseXYList.append([limit_x, limit_y])
        plt.plot(limit_x, limit_y, 'b.')
        path_length+=0.1
        plt.pause(0.0001)

        # IF Completely Traversed the Facet
        if limit_x == poseXYList[0][0] and limit_y == poseXYList[0][1]:
            flag = True
            break

    # Managing the edge condition and next facet traveral is need be
    if val == 0:
        temp[0], polytope_number_list[0] = bug.collisionHelper(self, 1, \
            limit_x, limit_y, ActivatedPolytopes)
        temp[1], polytope_number_list[1] = bug.collisionHelper(self, 3, \
            limit_x, limit_y, ActivatedPolytopes)
    elif val == 1:
        temp[0], polytope_number_list[0] = bug.collisionHelper(self, 4, \
            limit_x, limit_y, ActivatedPolytopes)
        temp[1], polytope_number_list[1] = bug.collisionHelper(self, 1, \
            limit_x, limit_y, ActivatedPolytopes)

    if direction:
        if val == 0:
            temp[0], polytope_number_list[0] = bug.collisionHelper(self, 4, \
                limit_x, limit_y, ActivatedPolytopes)
            temp[1], polytope_number_list[1] = bug.collisionHelper(self, 5, \
                limit_x, limit_y, ActivatedPolytopes)
        elif val == 1:
            temp[0], polytope_number_list[0] = bug.collisionHelper(self, 2, \
                limit_x, limit_y, ActivatedPolytopes)
            temp[1], polytope_number_list[1] = bug.collisionHelper(self, 3, \
                limit_x, limit_y, ActivatedPolytopes)

    # If it Collided along the X or Y Direction
    if temp[0] or temp[1]:
        PerimeterTraversalStatus = 0
        if val == 0:
            # Find the next possible polytope and facet which can be
            # Travelled
            EnabledPolytope = ActivatedPolytopes[polytope_number_list[1]]
            if (temp[0]):
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[0]]

            if temp[0] & temp[1]:
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[0]]
        elif val == 1:
            if temp[1]:
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[1]]
            else:
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[0]]

            if temp[0] & temp[1]:
```

```
            if direction:
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[1]]
            else:
                EnabledPolytope = ActivatedPolytopes[polytope_number_list[0]]

        # Store these Vertexes and Check if M-Line Traversal is possible
        V1, V2 = NextLocationXY(EnabledPolytope, limit_x, limit_y)
        # if isinstance(V2, type(None)):
        if type(V2) == type(None):

            mline = np.dot(EnabledPolytope.A, np.array([[limit_x], [limit_y]])) \
                - np.transpose( EnabledPolytope.b[np.newaxis])
            facet = min_index(mline)

        else:
            index_for_facet = np.where((EnabledPolytope.vertices == \
                (V2[0], V2[1])).all(axis=1))
            facet = index_for_facet[0][0]

    else:
        # Update Flags
        PerimeterTraversalStatus += 1
        facet+= 1

    return EnabledPolytope, limit_x, limit_y, ActivatedPolytopes, flag, \
            poseXYList, facet, PerimeterTraversalStatus, \
                V1, V2, polytope_number_list, path_length

def algo_bug1(self, start, goal, ActivatedPolytopes):
    """
    Implementation of Bug1 Algorithm
    PARAMETERS
    ----------
        start : Start Pose of the robot
        Goal : Final Destination of the robot
        ActivatedPolytopes: List of All the Polytopes in the config
    """
    point_list = []
    x = start[0,0]
    y = start[1,0]
    path_length = 0.0
    # Total Euclidean Distance
    t = np.linalg.norm(start - goal)
    # Plot the start and goal destination
    plt.plot([x, goal[0]], [y, goal[1]], 'kx')
    point_list.append([x,y])
    polytope_number_list = [0,0]

    while (goal[0] - x != 0) or (goal[1] - y != 0):

        x, y, collide, EnabledPolytope, angle = bug.setup(self, goal, x, y, t, ActivatedPolytopes)
        print("Total Length : ", path_length)
        # Traversing the Polytope
        if collide:
```

```
if x <= round(round(x, 1)) + (GLOBAL_UPDATE_TIME*20):
    x = round(round(x, 1))
y = round(y, 1)

poseXYList = [[x, y]]

# Polytope closest from current robot pose
mline = np.dot(EnabledPolytope.A, np.array([[x], [y]])) - np.transpose(EnabledPolytope.b[np.newaxis])
# M-line tacticle sensor based knowledge
facet = min_index(mline)

# Setup
PerimeterTraversalStatus = 0
limit_x, limit_y = x, y
flag = False

while PerimeterTraversalStatus != 4: # Can travel a max of 4 sides of a polytope

    if limit_x == poseXYList[0][0] and \
        limit_y == poseXYList[0][1] and \
            PerimeterTraversalStatus != 0:
        flag = True
        break

    # If back to the first facet
    if facet > EnabledPolytope.A.shape[0] - 1:
        facet = 0

    #Logic for Next possible pose of the robot
    if EnabledPolytope.A[facet][1] == 0:
        V1, V2 = NextLocationXY(EnabledPolytope, limit_x, limit_y)
        if flag:
            break
        EnabledPolytope, limit_x, limit_y, ActivatedPolytopes, flag, \
            poseXYList, facet, PerimeterTraversalStatus, V1, V2,\
                polytope_number_list, path_length =  bug.bug_1_helper(self, EnabledPolytope, limit_x,\
                limit_y, ActivatedPolytopes, flag,\
                    poseXYList, facet, PerimeterTraversalStatus, V1, V2,
                    polytope_number_list, 0, path_length)

    else:
        V1, V2 = NextLocationXY(EnabledPolytope, limit_x, limit_y)
        if flag:
            break
        EnabledPolytope, limit_x, limit_y, ActivatedPolytopes, flag, \
            poseXYList, facet, PerimeterTraversalStatus, V1, V2,\
                polytope_number_list, path_length =  bug.bug_1_helper(self, EnabledPolytope, limit_x,\
                limit_y, ActivatedPolytopes, flag,\
                    poseXYList, facet, PerimeterTraversalStatus, V1, V2,
                    polytope_number_list, 1, path_length)

# Find the vertex on the polytope next to currently traversed polytope
if flag or PerimeterTraversalStatus == 4:
    closestPointIndex = polytopeEuclideanDistance(poseXYList, goal)
```

```
            # Plotter along the Polytopes Traversal
            for i in range(len(poseXYList) -1, closestPointIndex, -1):
                plt.plot(poseXYList[i][0], poseXYList[i][1], 'c.')
                path_length+=0.1
                plt.pause(0.001)

            # Rounding Up here is ectremely vital else leads to vague results
            limit_x, limit_y = round(poseXYList[closestPointIndex][0], 2), \
                round(poseXYList[closestPointIndex][1], 2)

            # Plotter Logic
            plt.plot(limit_x, limit_y, 'r.')
            plt.pause(0.001)
            print("Total Length : ", path_length)
            if  goal[0][0]-0.2 <=limit_x <= goal[0][0]+0.2 and \
                goal[1][0]-0.2 <= limit_y <= goal[1][0]+0.2:
                print("Total Length : ", path_length)
                sys.exit()

            x, y = limit_x, limit_y


        # When outside the Collision Zone
        x, y = round(x, 2), round(y, 2)
        plt.plot(x, y, 'b*')
        path_length +=0.1
        plt.pause(0.001)

    # Goal Sucessfully traversed
    if limit_x == goal[0][0] and limit_y == goal[1][0]:
        print("Total Length : ", path_length)
        # sys.exit()
        exit(0)

def bug_2_helper(self, previous_pose, temp2, val1, val2,
                 next_possible_pose, op, EnabledPolytope,
                 corr, b1, b2, ActivatedPolytopes):
    """
    Helper Function for Implementation of Bug2 - Algorithm
    It Checks the following
    1) Facet Hit of the EnabledPolytope
    2) Next possible location based on the Facet Found
    PARAMETERS
    ----------
        previous_pose = Existing Pose of the Robot
        temp2: X or Y depending on Pose Orientation
        val1, val2: Helper Variables for Collision Checking
        next_possible_pose : Next Pose of the Robot
        op : Helper Mathematical Operation
        EnabledPolytope: List of all Polytopes in configuration
        corr: Correlation Param
        b1, b2: Helper Variables
        ActivatedPolyopes: Currently Active Polytope about which the
```

```
                    robot is traversing
"""
#Helper Variables for Generic Implementation
temp1 = round(previous_pose[corr], 1)
temp2 = op(temp2, 0.1)
temp3 = op(temp2, 0.1)

# Collision Status Tracker
if corr == 0:
    collide, next_poly = bug.collisionHelper(self, val1, temp1,\
         temp2, ActivatedPolytopes)
    if ActivatedPolytopes[next_poly] == EnabledPolytope:
        collide, next_poly = bug.collisionHelper(self, val2, temp1,\
             temp2, ActivatedPolytopes)
elif corr == 1:
    collide, next_poly = bug.collisionHelper(self, val1, temp2, \
        temp1, ActivatedPolytopes)
    if ActivatedPolytopes[next_poly] == EnabledPolytope:
        collide, next_poly = bug.collisionHelper(self, val2, temp2,\
             temp1, ActivatedPolytopes)

# Locate the next possible Polytope / mline and move
if collide and ActivatedPolytopes[next_poly] != EnabledPolytope:
    EnabledPolytope = ActivatedPolytopes[next_poly]
    previous_pose = next_possible_pose

    previous_pose_index = NextPoseHelperFunc(EnabledPolytope, previous_pose)

    a = (round(temp3, 1) != next_possible_pose[0])
    b = (isinstance(previous_pose_index, type(None)))
    if a or b:
        if corr == 1:
            previous_pose, next_possible_pose = PoseFromPolytopeVertex(round(temp3, 1),\
                 round(temp1, 1), \
                     EnabledPolytope, b1, b2)
        if corr == 0:
            previous_pose, next_possible_pose = PoseFromPolytopeVertex(round(temp1, 1),\
                 round(temp3, 1),\
                      EnabledPolytope, b1, b2)

    else:
        # Previous Pose is Valid
        next_possible_pose = EnabledPolytope.vertices[previous_pose_index - 1]

# If not in collision find the next vertex
elif round(temp2, 1) == round(next_possible_pose[1-corr]):
    previous_pose = next_possible_pose
    for counter, vertex in enumerate(EnabledPolytope.vertices):
        if previous_pose[0] == vertex[0] and previous_pose[1] == vertex[1]:
            prev_index = counter
            break
    next_possible_pose = EnabledPolytope.vertices[prev_index - 1]

if corr == 0:
```

```
            return temp1, temp2, previous_pose, next_possible_pose, ActivatedPolytopes, collide, EnabledPolytope, nex
t_poly
        if corr == 1:
            return temp2, temp1, previous_pose, next_possible_pose, ActivatedPolytopes, collide, EnabledPolytope, nex
t_poly


    def algo_bug2(self, start, goal, ActivatedPolytopes):
        """
          Tracks the pose of the robot from the start pose and keeps updating two states of the robot
          previous state and the next, depending on the status of collision of the next the robot ose is updated
          either left, rigght, up or down
        """
        path_length = 0.0
        point_list = []
        x = start[0,0]
        y = start[1,0]
        visited = {}
        starting_pose = (0,0)
        ending_pose = (0,0)

        # Euclidean Distance of the goal
        t = np.linalg.norm(start - goal)
        plt.plot([x, goal[0]], [y, goal[1]], 'kx')
        point_list.append([x,y])
        polytope_number_list = [0,0]
        flag_plot_delay = 1

        # Move Logic
        while (goal[0] - x != 0) or (goal[1] - y != 0):
            print("Path length: ", path_length)

            # Initial Setup
            x, y, collide, EnabledPolytope, angle = bug.setup(self, goal, x, y, t, ActivatedPolytopes)

            # Counter Variables and Lists to keep track of existing poses of the robot
            on_facet_counter = 0
            polytopes_vertex_list = []

            # Checks at every instant is valid to move on mline post collision
            if collide:
                # Checks of next possible location
                mline = np.dot(EnabledPolytope.A, np.array([[x], [y]])) -\
                     np.transpose(EnabledPolytope.b[np.newaxis])
                faceOnPolytope = min_index(mline)
                previous_pose, next_possible_pose = NextLocationXY(EnabledPolytope, \
                    round(x, 1), round(y, 1))

                # If the Result is invalid
                # if isinstance(previous_pose, type(None)):
                if (type(previous_pose) ==  type(None)):
                    x, y = round(x, 1), round(y, 1)

                    # If on a facet, move towards the vertex end of the facet
```

```
    next_possible_pose = EnabledPolytope.vertices[faceOnPolytope]
    if faceOnPolytope + 1 > EnabledPolytope.vertices.shape[0] - 1:
        previous_pose = EnabledPolytope.vertices[0]
    else:
        previous_pose = EnabledPolytope.vertices[faceOnPolytope + 1]

# Rounding is Extremely Vital post Pose checks else lead to divergent solutions
x, y = round(x, 1), round(y, 1)

if previous_pose[0] == next_possible_pose[0]:
    # Since the Robot is in discrete space , it can take 4 possible move directions
    while x != next_possible_pose[0] or y != next_possible_pose[1]:
        # if round(y,1) == goal[1,0]:
        #     memory[round(x,1) + round(y+1)] = 0

        # Right
        if y < next_possible_pose[1] and round(x) == round(next_possible_pose[0]):
            x, y, previous_pose, next_possible_pose, ActivatedPolytopes, \
                collide, EnabledPolytope, next_poly= bug.bug_2_helper(self, previous_pose, y, 3, 1,
                                                    next_possible_pose, add, EnabledPolytope, 0, False,
                                                    True, ActivatedPolytopes)
        # Down
        elif y > next_possible_pose[1] and round(x) == round(next_possible_pose[0]):
            x, y, previous_pose, next_possible_pose, ActivatedPolytopes, \
                collide, EnabledPolytope, next_poly= bug.bug_2_helper(self, previous_pose, y, 4, 5,
                                                    next_possible_pose, sub, EnabledPolytope, 0, False,
                                                    False, ActivatedPolytopes)
        # Move Up
        elif x < next_possible_pose[0] and round(y) == round(next_possible_pose[1]):
            x, y, previous_pose, next_possible_pose, ActivatedPolytopes, \
                collide, EnabledPolytope, next_poly= bug.bug_2_helper(self, previous_pose, x, 5, 3,
                                                    next_possible_pose, add, EnabledPolytope, 1, True,
                                                    False, ActivatedPolytopes)
        # Left
        else:
            x, y, previous_pose, next_possible_pose, ActivatedPolytopes, \
                collide, EnabledPolytope, next_poly= bug.bug_2_helper(self, previous_pose, x, 1, 4,
                                                    next_possible_pose, sub, EnabledPolytope, 1, False,
                                                    False, ActivatedPolytopes)

        # Plots the next possible pose
        flag_plot_delay +=1
        if (flag_plot_delay % 5) == 0:
            plt.plot(x, y, 'b*')
            path_length += 0.1
            plt.pause(0.01)

        if goal[1][0] == 0:
            flag = round((y), 1)


        if flag == 0:
            # Chattering Condition over M-Line check
            if y <= start[0][0] or x <= start[1][0]:
```

```
            continue

        # Ability to move over the m-line
        mline_flag = False
        # if not isinstance(polytopes_vertex_list, type(None)):
        if not (type(polytopes_vertex_list) == type(None)):
            for vertex in polytopes_vertex_list:
                if round(x, 1) <= round(vertex[0], 1) and round(y, 1) <= round(vertex[1], 1):
                    mline_flag = True
                    break
        # Prevents Collision Check mline_flag is still valid
        if mline_flag:
            continue

        # If mline flag is not valid Continue polytope traversal
        on_facet_counter += 1
        polytopes_vertex_list.append([round(x, 1), round(y, 1)])
        if on_facet_counter != 1:
            # Goal Reach Check
            if round(x, 1) == round(polytopes_vertex_list[0][0], 1) and \
                round(y, 1) == round(polytopes_vertex_list[1][0] , 1):
                break

        # Helper Function check for Collision if not on mline
        if round(y, 1) <= round(goal[1][0], 1):
            collide, _ = bug.collisionHelper(self,3, x, y, ActivatedPolytopes)
        else:
            collide, _ = bug.collisionHelper(self,4, x, y, ActivatedPolytopes)

        # if(round(y, 1) == round(goal[1][0], 1)):
        #     memory[round(x, 1)] = 0
        #     memory[round(x, 1)]+=1
        mline_constraint = (round(y, 1) >= round(goal[1][0], 1) - 0.05) and ((round(y, 1) < round(goal[1][
0], 1) + 0.05))

        while not collide and mline_constraint:

            # If within Goal radius of 0.2 from goal reached succesfully, exact location is
            # Check is not satisfied due to accuracy being inconsistent during calculation.
            if ( round(goal[0][0]-0.2 , 1) <= round(x , 1) <= round(goal[0][0]+0.2, 1) ) and \
                round(goal[1][0]-1, 1) <= round(y, 1) <= round(goal[1][0]+1, 1):
                time.sleep(3)
                exit(0)

            # Else continue with Move logic
            if round(y,1) <= goal[1][0]:
                x += t * np.cos(angle) * GLOBAL_UPDATE_TIME
                y += t * np.sin(angle) * GLOBAL_UPDATE_TIME
            else:
                # If Goal is below move Diagonally Downwards
                x -= t * np.cos(angle) * GLOBAL_UPDATE_TIME
                y -= t * np.sin(angle) * GLOBAL_UPDATE_TIME

            # Collision Checks and movement to find the next possible traversal polytope
            collide, update_poly = bug.collisionHelper(self,6, x, y, ActivatedPolytopes)
```

```
                        EnabledPolytope = ActivatedPolytopes[update_poly]
                        mline = np.dot(EnabledPolytope.A, np.array([[x], [y]])) - \
                            np.transpose(EnabledPolytope.b[np.newaxis]) #Vertex Closest to robot pose
                        faceOnPolytope = min_index(mline)

                        # When the next possible facet of traversal is found locate their Vertexes
                        previous_pose, next_possible_pose = NextLocationXY(EnabledPolytope, \
                            round(x, 1), round(y, 1))

                        # When traversal is complete, Check for next possible polytope/ mline
                        if (type(previous_pose) == type(None)):
                            x, y = round(x, 1), round(y, 1)
                            next_possible_pose = EnabledPolytope.vertices[faceOnPolytope]

                            if faceOnPolytope + 1 > EnabledPolytope.vertices.shape[0] - 1:
                                previous_pose = EnabledPolytope.vertices[0]
                            else:
                                previous_pose = EnabledPolytope.vertices[faceOnPolytope + 1]

                        previous_pose = next_possible_pose
                        plt.plot(x, y, 'g*')
                        path_length+=0.1
                        print("Path length: ", path_length)
                        plt.pause(0.01)


def main():
    """
    RESULTS
    -------
    Workspace -1
        Bug 1 : Length : 102 uints
        Bug 2 : 19.8 units
    Workspace -2
        Bug 1 : Length : 315.6 uints
        Bug 2 : Length : 41.2 uints
    """
    PointSizedRobot = bug()
    ws = input(" Enter 1 for Workspace 1 or  for Workspace 2 \n")
    algorithmToBeUsed = input(" Enter 1 for using Bug-1 Algorithm or 2 for Bug-2 Algorithm \n")
    start = WORKSPACE_CONFIG['start_pos']
    if ws == '1':
        goal = WORKSPACE_CONFIG['WO1_goal']
    elif ws == '2':
        goal = WORKSPACE_CONFIG['WO2_goal']

    PointSizedRobot.plot_plt(ws, start, goal, algorithmToBeUsed)

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

==

<Helper.py>

```
import numpy as np
```

```
import matplotlib.pyplot as plt
import sys
import time
import polytope as pc
import scipy
from scipy.spatial import Delaunay as Del
from workspaces import config
from operator import eq, add, sub

WORKSPACE_CONFIG = config()

def min_index(pts):
    """
    PARAMETERS
    ----------
    Vertexes of Polytopes : List
    RETURNS
    -------
    Index of the Vertex
    """
    min = np.NINF
    status= None
    for idx, val in enumerate(pts):
        if val > min:
            min = val
            status = idx

    return status

def polytopeEuclideanDistance(pts, goal):
    """
    PARAMETERS
    ----------
    Vertexes of Polytopes : List
    RETURNS
    -------
    Smallest Distance from the the vertex of polytopes and the goal
    """
    min_val = np.Inf
    stat = None
    for idx, pt in enumerate(pts):
        dist = np.sqrt( (pt[0] - goal[0][0])**2 + (pt[1] - goal[1][0])**2 )
        if dist < min_val:
            min_val = dist
            stat = idx

    return stat

def NextLocationXY(EnabledPolytope, limit_x, limit_y):
    """
    Helper Function to Vertexes of Polytope
    """
    vertex1 = vertex2  = None
    vertex2 = None
```

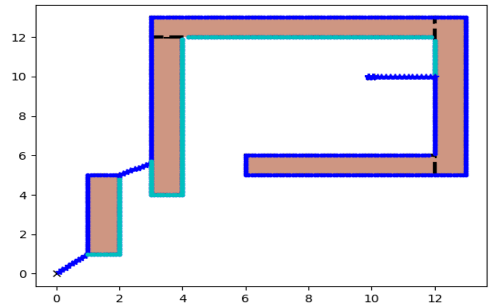
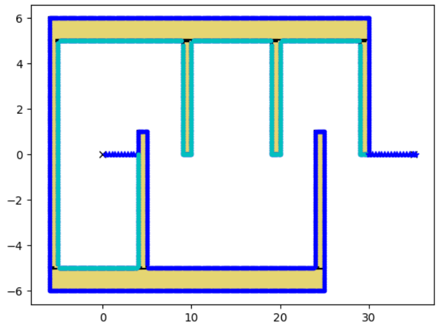
```python
    for idx, vertex in enumerate(EnabledPolytope.vertices):
        if limit_x == vertex[0] and limit_y == vertex[1]:

            vertex1 = vertex
            if idx != 0:
                vertex2 = EnabledPolytope.vertices[idx - 1]
            else:
                vertex2 = EnabledPolytope.vertices[-1]
    return vertex1, vertex2


def NextPoseHelperFunc(EnabledPolytope, facet_vertex):

    vx = None
    for idx, row in enumerate(EnabledPolytope.vertices):
        if facet_vertex[0] == row[0] and facet_vertex[1] == row[1]:
            vx = idx
            break
    return vx


def PoseFromPolytopeVertex(x, y, EnabledPolytope, vflag, hflg):
    """
    Helper Function to locate the next possible pose Vertex from the current
    pose of the robot for Bug 2 Algorithm
    PARAMETERS
    ----------
        (x, y) : Current Pose of the Robot
        EnabledPolytope: Currently active polytope
        vflag: Flag to Check if Vertical Traversal is preferred
        hflag: Flag to check is Horizontal Traversal is preferred
    RETURNS
    -------
        Next Pose of the Robot
    """
    x = round(x)
    for idx, vtx in enumerate(EnabledPolytope.vertices):
        if x == vtx[0] or y == vtx[1]:
            flag = False
            pose = idx
            if x == vtx[0]:
                flag = True

            if vflag:
                for idx, vtx in enumerate(EnabledPolytope.vertices):
                    if x == vtx[0]:
                        flag = True
                        pose = idx
                        break
            if hflg:
                for idx, vtx in enumerate(EnabledPolytope.vertices):
                    if y == vtx[1] and x == vtx[0]:
                        flag = False
```

```
                pose = idx
                break
        break

    possible_pose = pose + 1
    if flag:
        if pose != 0:
            if (EnabledPolytope.vertices[pose - 1][0] == x) or (EnabledPolytope.vertices[pose - 1][1] == y) :
                possible_pose = pose - 1
        else:
            if (EnabledPolytope.vertices[3][0] == x) or (EnabledPolytope.vertices[3][1] == y):
                possible_pose = 3

    if possible_pose == 4:
        possible_pose = 0

    v1 = EnabledPolytope.vertices[pose]
    v2 = EnabledPolytope.vertices[possible_pose]

    if not flag:
        if x <= round(v1[0], 1):
            v1, v2 = v2, v1
        if not hflg:
            v1, v2 = v2, v1
    else:
        if (y > round(v1[1], 1)):
            v1, v2 = v2, v1
        if not vflag:
            v1, v2 = v2, v1

    return v2, v1
```

```
==============================================================
<Workspaces.py>
==============================================================
# import os
# import os.path as osp
import polytope as pc
import numpy as np

def config():

    WO1_1 = pc.box2poly([[1,2],[1,5]])
    WO1_2 = pc.box2poly([[3,4],[4,12]])
    WO1_3 = pc.box2poly([[3,12],[12,13]])
    WO1_4 = pc.box2poly([[12,13],[5,13]])
    WO1_5 = pc.box2poly([[6,12],[5,6]])

    WO2_1 = pc.box2poly([[-6,25],[-6,-5]])
    WO2_2 = pc.box2poly([[-6,30],[5,6]])
    WO2_3 = pc.box2poly([[-6,-5],[-5,5]])
    WO2_4 = pc.box2poly([[4,5],[-5,1]])
    WO2_5 = pc.box2poly([[9,10],[0,5]])
    WO2_6 = pc.box2poly([[14,15],[-5,1]])
```

```
WO2_7 = pc.box2poly([[19,20],[0,5]])
WO2_8 = pc.box2poly([[24,25],[-5,1]])
WO2_9 = pc.box2poly([[29,30],[0,5]])

DEFAULT_TEST_CONFIG = {
        'WO1' : [WO1_1, WO1_2, WO1_3, WO1_4, WO1_5],
        'WO2': [WO2_1, WO2_2, WO2_3, WO2_4, WO2_5, WO2_6, WO2_7,WO2_8, WO2_9],
        'start_pos': np.array([[0], [0]]),
        'WO1_goal': np.array([[10], [10]]),
        'WO2_goal': np.array([[35], [0]])
}
return DEFAULT_TEST_CONFIG
```

Graphs

	Workspace -1	Workspace 2
Bug 1		
Bug 2