Do not write here!

Name: Vedad Bassari
Perm Number: 581996-6

1 pp1

ME179P W21

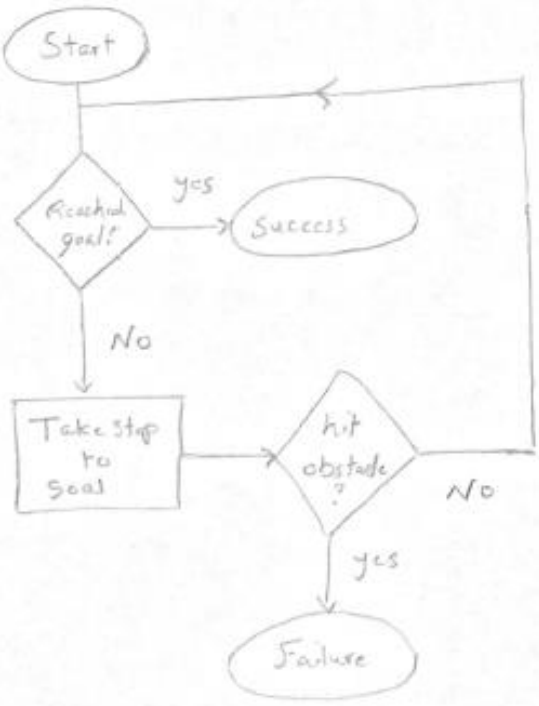
Programming Project #1

- Use a **DARK** pen or pencil, and write **INSIDE** the answer boxes provided.
- Write your name and perm number **CLEARLY** at the top of **EVERY** page, inside the boxes provided.
- Use the appropriate tools for submitting your code.

Exercise 1.8: The Bug1 algorithm

E1.8(i) Sketch a flowchart of the BugBase algorithm

Answer:



2 pp1

ME179P W21

Name: Vedad Bassari
Perm Number: 581996-6

Do not write here

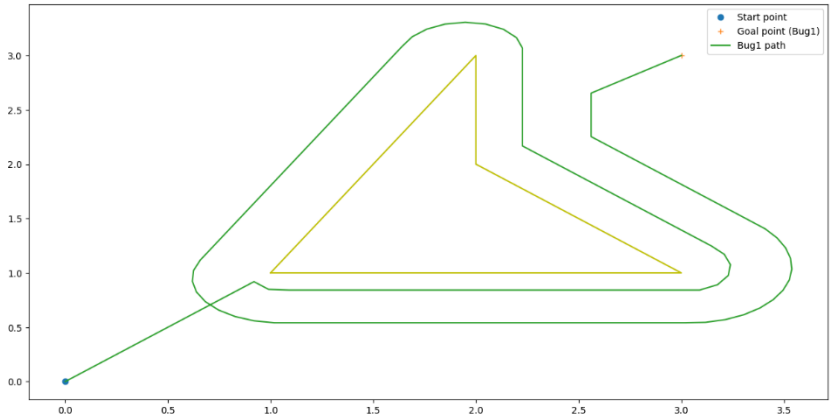
E1.8(ii)

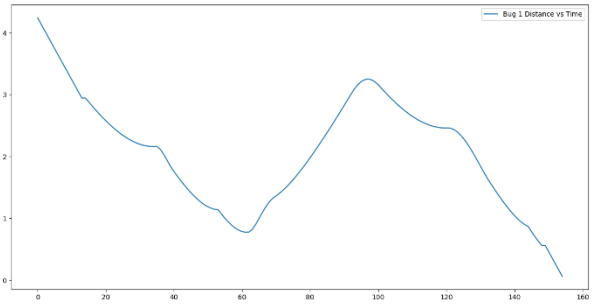
Answer:

To go from bug base to bug 1, we will alter the algorithms response when an obstacle is encountered. Instead of returning a failure, the algorithm will instead circumnavigate the obstacle and leave the obstacle at the point that is closest to the goal. The task of circumnavigating the obstacle use the tangent vectors to the polygon that are generated by the function from exercise 1.7. This function, in turn, uses the geometric functions of exercise 1.6 to define lines and form polygons from them.

E1.8(iv) plot 1

Answer:






```
Run Time:  0.012964963912963867
Total Path Length:  15.359293422248788
```

Do not write here!

Name: Vedad Barsari

Perm Number: 581996-6

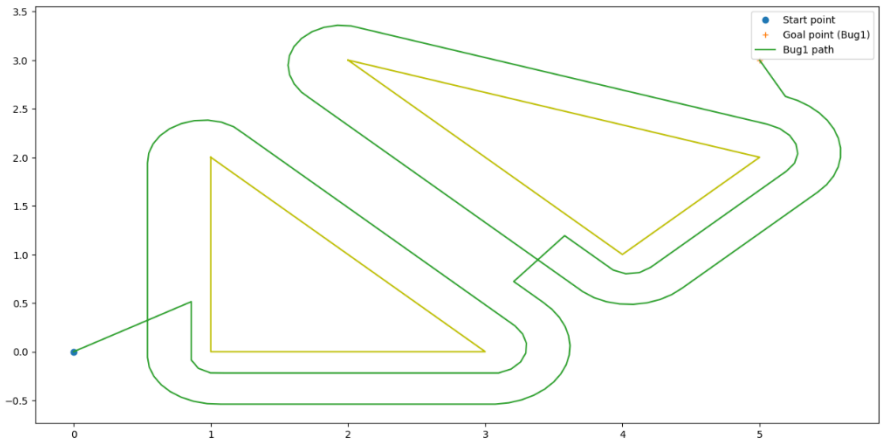
3

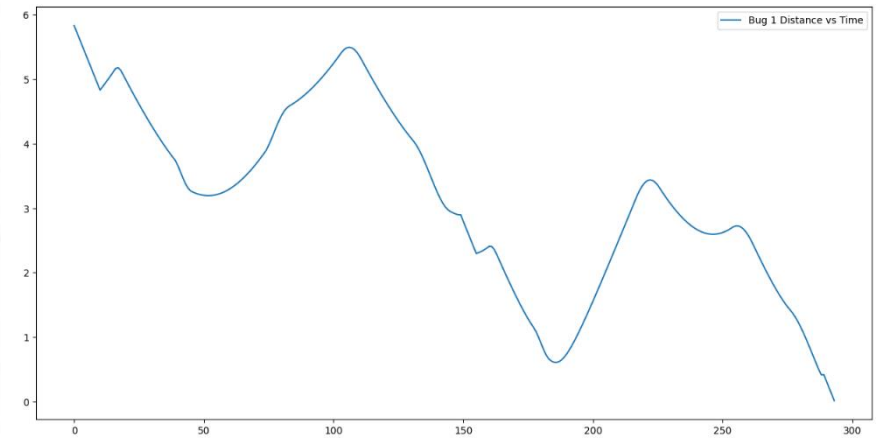
pp1

ME179P W21

E1.8(iv) plot 2

Answer:






```
Run Time:  1.1155030727386475
Total Path Length:  29.11804622370155
```

```python
# Copyright 2021 Francesco Seccamonte

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

# http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Template file for the assignment.
#
# Original author:  Patrick Therrien, 2015
# Refactor:         Francesco Seccamonte, 2021
#
# YOU MUST USE THIS FILE AND THE API HEREIN.
# DO NOT MODIFY HOW FUNCTIONS AND METHODS ARE DEFINED,
# SIMPLY FILL WHERE TODOs ARE.
#
# YOU NEED TO UPLOAD THIS FILE AS PART OF YOUR SUBMISSION


# Place your imports here as needed
import math

# Hint: the exercise asks you to give some kind of error
# in case of wrong inputs: you should do so by raising an Exception.

def computeLineThroughTwoPoints(p1, p2):
    """
    :param p1: a point in 2D specified as a numpy array
    :param p2: a point in 2D specified as a numpy array
    :return: a tuple (a,b,c) containing the coefficients of the line ax+ by = c
    passing through p1 and p2
    """

    # Write your code here
    x1 = p1[0] # Extract the coordinates from the array
    y1 = p1[1]
    x2 = p2[0]
    y2 = p2[1]
    distance = math.sqrt((abs(x2-x1))**2 + (abs(y2-y1))**2) # Calculate the distance between two
points
    tolerance = 10**-8 # Tolerance for detecting coincident points

    if distance<tolerance: # Exception: Points p1 and p2 cannot be the same
        raise Exception("Points cannot be coincident")

    if abs(y1-y2)<tolerance: # Exception: Equation for horizontal line
        a = 0
        b = 1
        c = -y1
    elif abs(x1-x2)<tolerance: # Exception: Equation for vertical line
        b = 0
        a = 1
        c = -x1
    else:
        a = (y1 - y2)/distance # Use formulas to calculate coefficients
        b = (x2 - x1)/distance
        c = (x1*y2 - x2*y1)/distance
```

```
    return a, b, c;


def computeDistancePointToLine(q, p1, p2):
    """
    :param q: the test point as a numpy array
    :param p1: the first point defining the line
    :param p2: the second point defining the line
    :return: the distance from the test point q to the line defined by p1 and p2
    """

    # Write your code here
    a,b,c = computeLineThroughTwoPoints(p1, p2); # Use the line defining function to get
coefficients
    x3 = q[0] # Extract coordinates of the third point
    y3 = q[1]
    d = abs(a*x3 + b*y3 + c) # Use orthogonal projection formula to find distance

    return d;


def computeDistancePointToSegment(q, p1, p2):
    """
    :param q: the test point as a numpy array
    :param p1: the first endpoint of the line segment
    :param p2: the second endpoint of the line segment
    :return: the distance d from the test point q to the line segment
            with endpoints p1 and p2
            w, with w=0 if the segment point closest to q is strictly
            inside the segment, w=1 if the closest point is p1, and
            w= 2 if the closest point is p2.
    """

    # Write your code here
    x1 = p1[0] # Extract the coordinates from all three points
    y1 = p1[1]
    x2 = p2[0]
    y2 = p2[1]
    x3 = q[0]
    y3 = q[1]
    d0 = computeDistancePointToLine(q, p1, p2) # Use the distance-finding function to find the
distance between q and the line corresponding to the segment
    d1 = math.sqrt((abs(x3-x1))**2 + (abs(y3-y1))**2) # Calculate distance between q and p1
    d2 = math.sqrt((abs(x3-x2))**2 + (abs(y3-y2))**2) # Calculate distance between q and p2
    o1 = math.sqrt(abs(d2**2-d0**2)) # Project the above distances onto the line
    o2 = math.sqrt(abs(d1**2-d0**2))
    l = math.sqrt((abs(x2-x1)**2+abs(y2-y1)**2)) # Find the length of the segment
    if o1>=l or o2>=l: # If the projected distances are greater than segment length, the orthogonal
projection does not fall onto the segment
        if d1<d2: # Pick the closest of points p1 and p2 as the distance
            w = 1
            d = d1
        else:
            w = 2
            d = d2
    else:
        w = 0 # Otherwise use orthogonal projection as the distance
        d = d0

    return d, w;
```

```python
# Copyright 2021 Francesco Seccamonte

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

# http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Template file for the assignment.
#
#
# Author:         Francesco Seccamonte, 2021
#
# YOU MUST USE THIS FILE AND THE API HEREIN.
# DO NOT MODIFY HOW FUNCTIONS AND METHODS ARE DEFINED,
# SIMPLY FILL WHERE TODOs ARE.
#
# YOU NEED TO UPLOAD THIS FILE AS PART OF YOUR SUBMISSION


# Place your imports here as needed
import numpy as np
import math
from programming_assignment1 import *

# Hint: the exercise asks you to give some kind of error
# in case of wrong inputs: you should do so by raising an Exception.

# Hint n.2: you may want to reuse the functions you wrote for
# programming assignment 1. You can import them by doing:
# from programming_assignment1 import *
# Remember to include that file as part of your submission!

def inpolygon(q, P):
    """
    :param q: a point in 2D specified as a numpy array
    :param P: a polygon with n vertices specified as an nx2 numpy array
    :return: 1 if the point q is inside the polygon P, 0 else.
    """

    from matplotlib import path

    p = path.Path(P);
    return p.contains_point(q);

def computeDistancePointToPolygon(q, P):
    """
    :param q: a point in 2D specified as a numpy array
    :param P: a polygon with n vertices specified as an nx2 numpy array
    :return: a tuple (d,v,idx) containing the distance from the point to
                the polygon, whether the distance is wrt a vertex (v=1) or not,
                the index idx within the polygonal list of
                either the vertex or segment which is closest to q.
                The first segment is considered to be the one
                between the first and second vertices.
    """

    # Please note: terms vertex and node are used interchangeably
```

```
    # It is assumed that all points are labelled in CCW convention, and that the
    # vertices are alligned in a CCW fashion within the input arrays.


    bool1 = inpolygon(q,P) # Define the boolean corresponding to point being in the polygon
    if bool1 == 1: # Raise an exception if the point is in the polygon
        return 0,0,0
    size = len(P) # Obtain the size of the array, or the number of nodes
    for i in range(size): # Loop through the nodes and form segments
        if i == size-1: # For the n-th node, the segment must connect to 0-th node
            p1 = P[i]
            p2 = P[0]
        else: # For all other node, the segment connects the i-th and the i+1-th nodes
            p1 = P[i]
            p2 = P[i+1]
        dT,wT = computeDistancePointToSegment(q,p1,p2) # Use previously defined function to find
distance to segment
        if i == 0: # For the first segment, store the distance to segment as the distance to the
polygon
            if wT == 0: # Depending on wheter the distance is to a vertex or to the segment, index
and assign v
                d = dT
                w = wT
                v = 0
                idx = np.array([0,1])
            elif wT == 1:
                d = dT
                w = wT
                v = 1
                idx = 0
            else:
                d = dT
                w = wT
                v = 1
                idx = 1
        else: # For all other segments:
            if d>=dT: # Write the distance to segment as the distance to the polygon if the
distance to segment is
                     # smaller than the previously stored distance to polygon
                if wT == 0:
                    d = dT
                    w = wT
                    v = 0
                    if i == size-1: # For the n-th node, the index must refer to the 0-th node as
the second segment-defining node
                        idx = np.array([i,0])
                    else:
                        idx = np.array([i,i+1])
                elif wT == 1: # Since each vertex is encountered twice, only the second encounter
is stored
                    d = dT
                    w = wT
                    v = 1
                    idx = i

    return d, v, idx;


def computeTangentVectorToPolygon(q, P):
    """
    :param q: a point in 2D specified as a numpy array
    :param P: a polygon with n vertices specified as an nx2 numpy array
    :return: the two-dimensional unit vector t (numpy array) tangent
                to the polygon P for a robot at q.
    """
```

```
    tolerance = 10**-8
    d,v,idx = computeDistancePointToPolygon(q,P) # Use the compute distance to polygon function to
    # find the distance and whether the closest point is a vertex or on a segment

    if v == 0: # If the closest point lies on the segment:
        r1 = idx[0] # Use the index to find the segment endpoints
        r2 = idx[1]
        p1 = P[r1] # Extract the x and y coordinates of the endpoints
        p2 = P[r2]
        x1 = p1[0]
        y1 = p1[1]
        x2 = p2[0]
        y2 = p2[1]
        x3 = q[0] # Extract coordinates of q
        y3 = q[1]

        # Find the point on the segment that is closes to q:
        if abs(x2-x1)<tolerance: # For the case where the line segment is vertical
            x4 = x2
            y4 = y3
        elif abs(y2-y1)<tolerance: # For the case where the line segment is horizontal
            y4 = y2
            x4 = x3
        else: # For all other line segments
            m1 = (y2-y1)/(x2-x1)
            m2 = -1/m1
            b1 = y2 - m1*x2
            b2 = y3-m2*x3 # Find the equation of the perpendicular line between the point and the
segment
            x4 = (b2-b1)/(m1-m2) # Find the point on the segment closest to the q
            y4 = b1 + m1*x4

        v1 = (x4 - x3)/d # Find the unit vector from q to the midpoint
        v2 = (y4 - y3)/d
        u1 = v2 # Use a 90-degree rotation transformation to get the unit tangent vector
        u2 = -v1

        t = np.array([u1,u2]) # Output the unit tangent vector

    else: # If the closest point is a vertex:
        p1 = P[idx] # Extract the vertext coordinates using the index
        x1 = p1[0]
        y1 = p1[1]
        x2 = q[0] # Extract coordinates of q
        y2 = q[1]

        v1 = (x1 - x2)/d # Find the unit vector from q to the vertex
        v2 = (y1 - y2)/d
        u1 = v2 # 90-degree Use a rotation transformation fo get the unit tangent vector
        u2 = -v1

        t = np.array([u1,u2]) # Output the unit tangent vector

    return t;

```

```python
# Copyright 2021 Francesco Seccamonte

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

# http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Template file for programming project 1.
#
#
# Author:         Francesco Seccamonte, 2021
#
# YOU MUST USE THIS FILE AND THE API HEREIN.
# DO NOT MODIFY HOW FUNCTIONS AND METHODS ARE DEFINED,
# SIMPLY FILL WHERE TODOs ARE.
#
# YOU NEED TO UPLOAD THIS FILE AS PART OF YOUR SUBMISSION


# Place your imports here as needed
import numpy as np
import math
from programming_assignment1 import *
from programming_assignment2 import *

# Hint: you may want to reuse the functions you wrote for
# programming assignments 1 and 2. You can import them by doing:
# from programming_assignment1 import *
# from programming_assignment2 import *
# Remember to include that file as part of your submission!

# Hint 2: it is strongly recommended to insert the functionalities
# needed in separate auxiliary functions, to improve debugging
# and readability, and to embrace the extremely useful DRY
# paradigm (=Don't Repeat Yourself)


def BugBase(start,goal,obstacleList,stepsize):
    """Implementation of the BugBase algorithm.

    :param start: a point in 2D specified as a numpy array
    :param goal: a point in 2D specified as a numpy array
    :param obstacleList: a list containing all the obstacles
                         (represented by polygons) in the environment.
                         # NOTE: our convention is that a polygon is
                         # represented by a list of vertices in
                         # counterclockwise order
    :param stepsize: a positive real number
    :return: a tuple (path, polygon index, success): path is a
             list containing the l 2D points
             that make up the path.
             polygon index is the index in the obstacleList
             corresponding to the polygon hit in case of returning a
             partial path (-1 if start-goal path returned).
             success is a boolean indicating whether the algorithm
             reached the goal or not.
    """
```

```
    tolerance = 10**-8 # Define tolerance for detecting a point
    x1 = start[0] # Set start point coordinates
    y1 = start[1]
    path = [[x1,y1]] # Begin the path list using start coordinates
    x2 = goal[0] # Set goal point coordinates
    y2 = goal[1]
    norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2) # Calculate the distance between start and end points
    l = len(obstacleList) # Obtain the number of polygons in the workspace

    while norm > stepsize: # While the goal point is not reached:
        dx = (x2 - x1)*(stepsize)/(norm) # Find the tangent vector of length stepsize that points towards the goal
        dy = (y2 - y1)*(stepsize)/(norm)
        cx1 = x1 + dx # Set the next step's coordinates as candidate
        cy1 = y1 + dy
        q = [cx1 , cy1]

        for n in range(l): # For all of the polygons in the workspace:
            P = obstacleList[n]
            d,v,idx = computeDistancePointToPolygon(q, P) # Obtain distance between candidate point and the polygon

            if d < tolerance: # If candidate point hits a polygon
                idx = n
                success = 0
                return path, idx, success # Terminate the function and return path, failure

        x1 = cx1 # If candidate point does not hit a polygon, move to it
        y1 = cy1
        path.append(q) # Append the path list with the new position
        norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2)

    path.append(goal) # Append the path list with the goal
    idx = -1
    success = 1 # Return success

    return path, idx, success;


def computeBug1(start,goal,obstacleList,stepsize):
    """Implementation of the Bug1 algorithm.

    :param start: a point in 2D specified as a numpy array
    :param goal: a point in 2D specified as a numpy array
    :param obstacleList: a list containing all the obstacles
                         (represented by polygons) in the environment.
                         # NOTE: our convention is that a polygon is
                         # represented by a list of vertices in
                         # counterclockwise order
    :param stepsize: a positive real number
    :return: a list containing the l 2D points
             that make up the path.
    """

    tolerance = 10**-8 # Define tolerance for detecting a point
    x1 = start[0] # Set start point coordinates
    y1 = start[1]
    path = [[x1,y1]] # Begin the path list using start coordinates
    x2 = goal[0] # Set goal point coordinates
    y2 = goal[1]
    norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2) # Calculate the distance between start and end points
    distanceList = [norm]
    l = len(obstacleList) # Obtain the number of polygons in the workspace
```

```
126
127     while norm > stepsize: # While the goal point is not reached:
128         dx = (x2 - x1)*(stepsize)/(norm) # Find the tangent vector of length stepsize that points
towards the goal
129         dy = (y2 - y1)*(stepsize)/(norm)
130         cx1 = x1 + dx
131         cy1 = y1 + dy
132         q = [cx1 , cy1] # Set the next step's coordinates as candidate
133         qb = [x1,y1]
134
135         for n in range(l): # For all of the polygons in the workspace:
136             P = obstacleList[n]
137             d,v,idx = computeDistancePointToPolygon(q, P) # Obtain distance between candidate point
and the polygon
138
139             if d < stepsize: # If candidate point hits a polygon
140                 t = computeTangentVectorToPolygon(qb,P) # Use computeTangentVectorToPolygon
function to find the tangent vector to the polygon
141                 dx = stepsize*t[0] # Circumnavigate the polygon using the tangent vector
142                 dy = stepsize*t[1]
143                 x1 = x1 + dx
144                 y1 = y1 + dy
145                 norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2) # Find the distance from the goal
146                 distanceList.append(norm)
147                 dmin = norm
148                 xmin = x1 # Store the first point as the current minimum distance from the goal
149                 ymin = y1
150                 q = [x1,y1]
151                 xstart = x1 # Store the starting point to identify end of circumnavigation
152                 ystart = y1
153                 path.append(q) # Append the path list with the new step
154
155                 for n in range(5): # Take five steps towards circumnavigation to ensure that the
robot is away from where it started circumnavigation
156                     t = computeTangentVectorToPolygon(q,P)
157                     dx = stepsize*t[0]
158                     dy = stepsize*t[1]
159                     x1 = x1 + dx
160                     y1 = y1 + dy
161                     norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2)
162                     distanceList.append(norm)
163                     q = [x1,y1]
164                     path.append(q)
165                     if norm < dmin: # If any point is closer to the goal than the previously stored
minimum distance, update the minimum distance point
166                         dmin = norm
167                         xmin = x1
168                         ymin = y1
169
170                 while (abs(xstart - x1)**2 + abs(ystart - y1)**2) > 0.15: # Continue
circumnavigation until the point of initial collision is reached
171                     # An arbitrary tolerance is imposed to ensure that the starting point is
detected
172                     print('in')
173                     t = computeTangentVectorToPolygon(q,P)
174                     dx = stepsize*t[0]
175                     dy = stepsize*t[1]
176                     x1 = x1 + dx
177                     y1 = y1 + dy
178                     norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2)
179                     distanceList.append(norm)
180                     q = [x1,y1]
181                     path.append(q)
182                     if norm < dmin: # Continue updating the point of minimum distance while
circumnavigating
```

```
                        dmin = norm
                        xmin = x1
                        ymin = y1

                while (abs(xmin - x1)**2 + abs(ymin - y1)**2) > 0.25: # Circumnavigate the polygon
until the point of minimum distacne is reached
                    # An arbitrary tolerance is imposed to ensure that the point of minimum
distance is detected
                    print('in here')
                    t = computeTangentVectorToPolygon(q,P)
                    dx = stepsize*t[0]
                    dy = stepsize*t[1]
                    x1 = x1 + dx
                    y1 = y1 + dy
                    cx1 = x1
                    cy1 = y1
                    q = [x1,y1]
                    path.append(q)
                    norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2)
                    distanceList.append(norm)
                    # Break out of the collision conditional when circumnavigation is successfully
completed

        x1 = cx1 # If candidate point does not hit a polygon, move to it
        y1 = cy1
        path.append(q) # Append the path list with the new position
        norm = ( (x2 - x1)**2 + (y2 - y1)**2 )**(1/2)
        distanceList.append(norm)

    path.append(goal) # Append the path list with the goal

    return path, distanceList;
```

```
# Copyright 2021 Francesco Seccamonte

# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at

# http://www.apache.org/licenses/LICENSE-2.0

# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

# Programming project 1 - Check the functions run

from programming_project1 import *
import numpy as np
import matplotlib.pyplot as plt
import math
import time
from programming_assignment1 import *
from programming_assignment2 import *

if __name__ == '__main__':

    start = np.array([0.0, 0.0]);
    goal = np.array([5.0, 3.0]);
    stepsize = 0.1;

    # NOTE: our convention is that a polygon is
    # represented by a list of vertices in
    # counterclockwise order
    P = np.array([[1, 2], # Obstacle 1
        [1, 0],
        [3, 0]]);

    Q = np.array([[2, 3], # Obstacle 2
        [4, 1],
        [5, 2]]);

    obstacleList = [P,Q]; # List of obstacles
    obstaclesList_plot = [P.tolist(),Q.tolist()];

    startTime = time.time() # Calculate run time using time module
    path, distanceList = computeBug1(start, goal, obstacleList, stepsize);
    endTime = time.time()
    length = len(path) # Obtain length of path
    d = 0
    for l in range(length): # Move accross each step of the path
        if l != 0:
            step = path[l]
            laststep = path[l-1]
            d = d + ( (step[0]-laststep[0])**2 + (step[1]-laststep[1])**2 )**(1/2) # Update total
path distance

    print('Run Time: ', endTime - startTime) # Print run time
    print('Total Path Length: ', d) # Print distance of path

    ###### PLOTTING #######
    plt.close('all');

    plt.plot(start[0],start[1],'o', label='Start point');
    plt.plot(goal[0],goal[1],'+', label='Goal point (Bug1)');
```

```
64
65      X,Y=map(list,zip(*path));
66      plt.plot(X,Y, label='Bug1 path');
67
68      for i in range(len(obstaclesList_plot)):
69          obstaclesList_plot[i].append(obstaclesList_plot[i][0]);
70          X,Y=map(list,zip(*obstaclesList_plot[i]));
71          plt.plot(X,Y, 'y');
72
73      plt.legend();
74      plt.show();
75
76      Y = distanceList; # Plot distance vs time
77      X = list(range(0, len(Y)))
78      plt.plot(X,Y, label='Bug 1 Distance vs Time');
79
80      plt.legend();
81      plt.show();
82
```