ECE452C_Fall-2018
Project-03_Team-7

Team member:

Yunjuan Wang

Yinbin Ma

Qiteng Wu

Video Link:

1. ECE452C Fall-2018 Proj-3 Team-7 Bug 0-Test 1:https://youtu.be/mMSnYxknxTM
2. ECE452C Fall-2018 Proj-3 Team-7 Bug 0-Test 2:https://youtu.be/QRDym5SDZfA
3. ECE452C Fall-2018 Proj-3 Team-7 Bug 1-Test 1: https://youtu.be/9P9fQtC19ZQ
4. ECE452C Fall-2018 Proj-3 Team-7 Bug 1-Test 2: https://youtu.be/Ukqe3wUIoSc
5. ECE452C Fall-2018 Proj-3 Team-7 Bug 2-Test 1: https://youtu.be/8wJCZy71AdQ
6. ECE452C Fall-2018 Proj-3 Team-7 Bug 2-Test 2: https://youtu.be/wLLAvK7shSc

Task allocation:

1. Configurate the camera: Yinbin Ma
2. Set the environment: Yunjuan Wang, Qiteng Wu
3. Bug 0 Algorithm: Yinbin Ma
4. Bug 1 Algorithm: Yunjuan Wang, Qiteng Wu
5. Bug 2 Algorithm: Yunjuan Wang
6. Debug the code and do the experiment: Yunjuan Wang, Yinbin Ma, Qiteng Wu

Difficulties & Solutions:

1. The map is imperfect, so it always leads to the failure of the line tracking. We fixed the map many times in order to make sure the line tracking could be done. Since the main point of this experiment is to implement bug algorithms, whenever the car fails to track the line, we manually fix it. You can also see this in the video.

2. In order to get the coordinates of the car easier, we buy a camera to detect the markers. We stick one marker on the car and another marker on the goal location. Thus, we can get the coordinates and the direction of the car in real time.

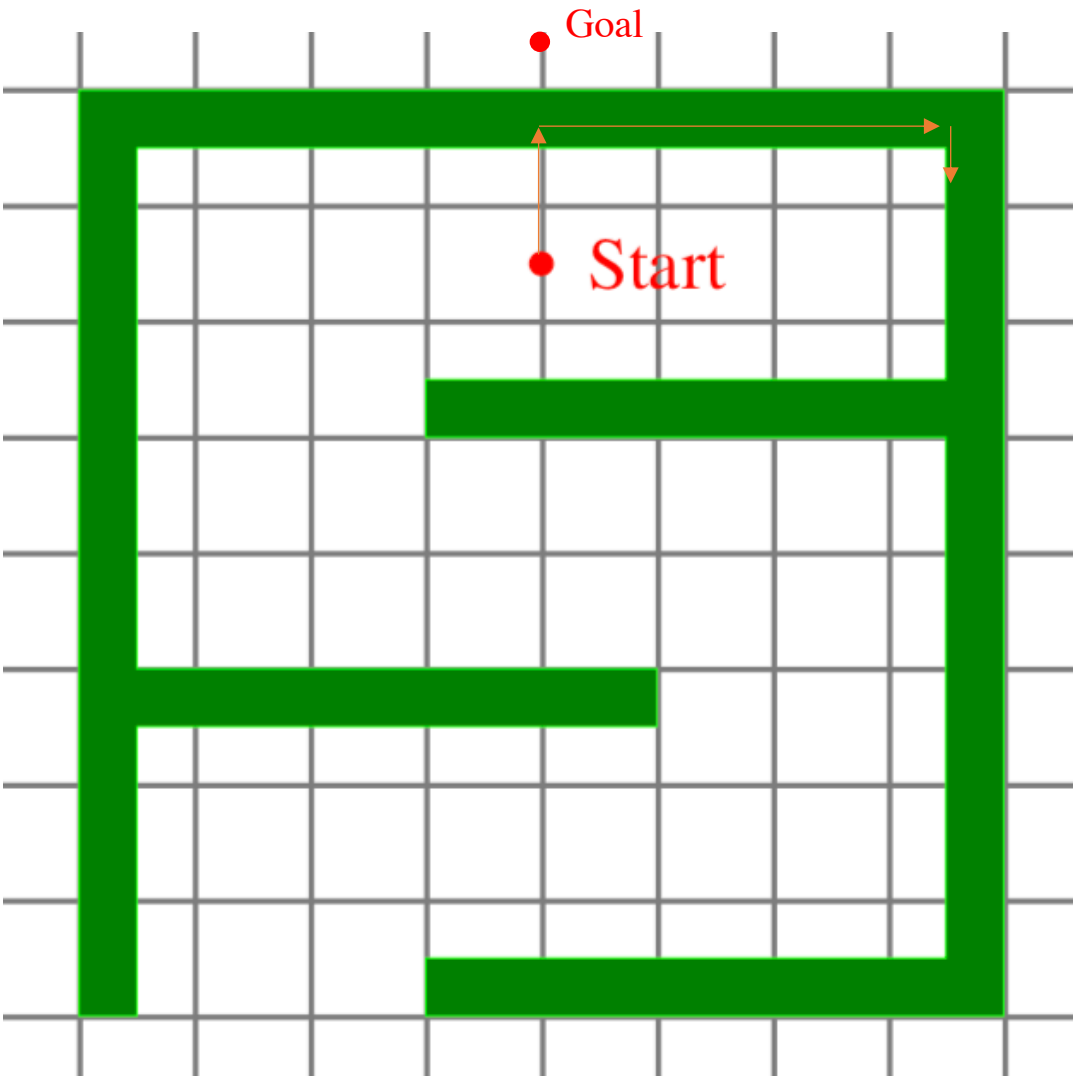
3. In Bug 0 algorithm, it almost impossible for us to detect whether there is obstacle between the car and the goal. We assume that the car has a chance to go to the goal only when it follows the line tracking and rotate 90 degree at the corner. Then we rotate the car and make its head towards the goal and detect whether there is black line under the sensor. If so, we assume that there is an obstacle between the car and the goal, then we keep tracking the line.

Discussion:

1. Is Bug 0 algorithm a good choice for this environment? Why? Under what conditions can Bug 0 find the goal?

Answer:

Bug 0 is not a good choice for this environment because the start point is in the obstacle and the goal is outside the obstacle. As we can see in the figure below, whenever the car reaches the top right corner and turn right, after line tracking for a small distance, we can find out there is no obstacle between the car and the goal, then the car will move towards the goal. It will hit the obstacle again and turn right, go ro the top right corner and repeat again.



2. Is Bug 1 algorithm guaranteed to find a path to the goal in this environment?

Answer:

Bug 1 is guaranteed to find a path as long as there is a path can lead the robot move from the start point to the goal.

3. Under what conditions can Bug 2 algorithm find the goal?

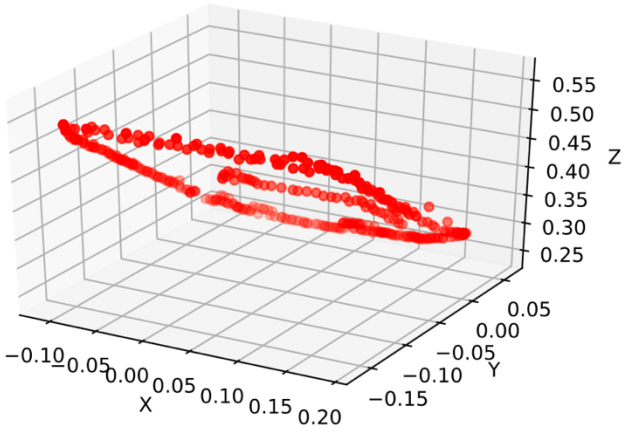
Answer:

As long as there is a path can lead the robot move from the start point to the goal, bug 2 algorithm will find it. Bug 2 algorithm is a simplWhen Bug 2 algorithm fails, there is no solution.

Notes:

1. How we localize the robot?

Instead of placing the markers around the map, we choose to attach the marker on the robot, and get the coordinate by external camera connected with a computer. The robot could communicate with computer to obtain its real-time coordinate during working.



We sampled the coordinates (translation coordinates which are relative to camera) when robot tracking black line, then scattered these coordinates in 3D space. The result is all coordinates data are mapping in a plane, so this method is reasonable.

2. How robot communist with computer?

We choose GRPC developed by Google. It is reliable and with high performance, and it already provide a testing webserver and client, so we don't need to pay too many attentions in communication coding.

In general, the camera connected with a computer, is capturing real-time images and sending to computer. The computer analyzes these images and update the coordinates. The robot obtains these coordinates and decide next moving. Thanks to GIL in Python, we don't need to worry about data race.

3. How to rotate the car's heading towards the goal.

We write a *rotation* function, where the rotate angle is the vector of (goal position-current position) and the vector of car's direction. Whether the rotation direction is left

or right can be decided by calculate the cross product of these two vectors defined above. In order to calculate the cross product of two vector, we project our vector from 3D to x-z plane. In this way the cross product is a number. Thus, we can easily find out whether the car need to turn left or turn right depend on the sign of this number. Another way to do this is to calculate the angle between these two vectors. If the angle is lower than a threshold, we assume the car's head is toward to the goal. To make the rotation more accuracy, we let one wheel move forward and another wheel move backward. The robot car will also move backward a lit bit after finish rotation. In this case, we can assume that the coordinates remain the same during rotation step.

4. How to find out three points on one line in Bug 2 algorithm
We write a *oneline* function. We calculate the distance between either two points as d1, d2, d3. If d1+d2-d3 or d1+d3-d2 or d2+d3-d1 lower than a certain threshold is true, these three points are on one line. The core idea is that if three points are not on one line, they can define a triangle.

Please see the following functions & steps and code comments for details on how to implement these algorithms

Code & Comments:

1. client.py

Function: Let robot car communicate the computer.

```
1.  from __future__ import print_function
2.  import grpc
3.  import aruco_pb2
4.  import aruco_pb2_grpc
5.  import cv2
6.  import numpy as np
7.
8.  server_ip = '10.107.208.86'
9.
10. channel = grpc.insecure_channel('{}:50051'.format(server_ip))
```

```
stub = aruco_pb2_grpc.GreeterStub(channel)

def check_connect():
    response = stub.SayHello(aruco_pb2.Empty())
    assert response is not None

def coordinate():
    c = stub.Ask(aruco_pb2.Empty())
    rvec = [c.r.x, c.r.y, c.r.z]
    car_front, _ = cv2.Rodrigues(np.array(rvec))
    tvec = [c.t.x, c.t.y, c.t.z]
    return np.array(tvec), car_front[:, 0]

def goal_location():
    c = stub.Goal(aruco_pb2.Empty())
    tvec = [c.t.x, c.t.y, c.t.z]
    return np.array(tvec)
```

2. Configure Environment

utils.py

Function: Robot configuration for the convenience of writing code calls later.

```
import RPi.GPIO as GPIO
from AlphaBot import AlphaBot
import numpy as np

CS = 5
Clock = 25
Address = 24
DataOut = 23

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)
GPIO.setup(Clock,GPIO.OUT)
GPIO.setup(Address,GPIO.OUT)
GPIO.setup(CS,GPIO.OUT)
GPIO.setup(DataOut,GPIO.IN,GPIO.PUD_UP)

class TRSensor(object):
    def __init__(self, numSensors=5):
        self.numSensors = numSensors
        self.calibratedMin = [967, 975, 972, 974, 984]
```

```
        self.calibratedMax = [190, 254, 243, 233, 461]
        self.last_value = 0

    """
    Reads the sensor values into an array. There *MUST* be space
    for as many values as there were sensors specified in the constructor.
    Example usage:
    unsigned int sensor_values[8];
    sensors.read(sensor_values);
    The values returned are a measure of the reflectance in abstract units,
    with higher values corresponding to lower reflectance (e.g. a black
    surface or a void).
    """

    def AnalogRead(self):
        value = [0, 0, 0, 0, 0, 0]
        # Read Channel0~channel4 AD value
        for j in range(0, 6):
            GPIO.output(CS, GPIO.LOW)
            for i in range(0, 4):
                # sent 4-bit Address
                if (((j) >> (3 - i)) & 0x01):
                    GPIO.output(Address, GPIO.HIGH)
                else:
                    GPIO.output(Address, GPIO.LOW)
                # read MSB 4-bit data
                value[j] <<= 1
                if (GPIO.input(DataOut)):
                    value[j] |= 0x01
                GPIO.output(Clock, GPIO.HIGH)
                GPIO.output(Clock, GPIO.LOW)
            for i in range(0, 6):
                # read LSB 8-bit data
                value[j] <<= 1
                if (GPIO.input(DataOut)):
                    value[j] |= 0x01
                GPIO.output(Clock, GPIO.HIGH)
                GPIO.output(Clock, GPIO.LOW)
            # no mean ,just delay
            for i in range(0, 6):
                GPIO.output(Clock, GPIO.HIGH)
                GPIO.output(Clock, GPIO.LOW)
            #           time.sleep(0.0001)
            GPIO.output(CS, GPIO.HIGH)
```

```
65.         return value[1:]
66.
67.     """
68.     Reads the sensors 10 times and uses the results for
69.     calibration.  The sensor values are not returned; instead, the
70.     maximum and minimum values found over time are stored internally
71.     and used for the readCalibrated() method.
72.     """
73.
74.     def calibrate(self):
75.         max_sensor_values = [0] * self.numSensors
76.         min_sensor_values = [0] * self.numSensors
77.         for j in range(0, 10):
78.
79.             sensor_values = self.AnalogRead();
80.
81.             for i in range(0, self.numSensors):
82.
83.                 # set the max we found THIS time
84.                 if ((j == 0) or max_sensor_values[i] < sensor_values[i]):
85.                     max_sensor_values[i] = sensor_values[i]
86.
87.                 # set the min we found THIS time
88.                 if ((j == 0) or min_sensor_values[i] > sensor_values[i]):
89.                     min_sensor_values[i] = sensor_values[i]
90.
91.         # record the min and max calibration values
92.         for i in range(0, self.numSensors):
93.             if (min_sensor_values[i] > self.calibratedMin[i]):
94.                 self.calibratedMin[i] = min_sensor_values[i]
95.             if (max_sensor_values[i] < self.calibratedMax[i]):
96.                 self.calibratedMax[i] = max_sensor_values[i]
97.
98.     """
99.     Returns values calibrated to a value between 0 and 1000, where
100.      0 corresponds to the minimum value read by calibrate() and 1000
101.      corresponds to the maximum value.  Calibration values are
102.      stored separately for each sensor, so that differences in the
103.      sensors are accounted for automatically.
104.      """
105.
106.      def readCalibrated(self):
107.          value = 0
108.          # read the needed values
```

```
        sensor_values = self.AnalogRead();

        for i in range(0, self.numSensors):

            denominator = self.calibratedMax[i] - self.calibratedMin[i]

            if (denominator != 0):
                value = (sensor_values[i] - self.calibratedMin[i]) * 1000 / denominator

            if (value < 0):
                value = 0
            elif (value > 1000):
                value = 1000

            sensor_values[i] = value

        # print("readCalibrated", sensor_values)
        return sensor_values

    """
    Operates the same as read calibrated, but also returns an
    estimated position of the robot with respect to a line. The
    estimate is made using a weighted average of the sensor indices
    multiplied by 1000, so that a return value of 0 indicates that
    the line is directly below sensor 0, a return value of 1000
    indicates that the line is directly below sensor 1, 2000
    indicates that it's below sensor 2000, etc.  Intermediate
    values indicate that the line is between two sensors.  The
    formula is:

       0*value0 + 1000*value1 + 2000*value2 + ...
       --------------------------------------------
            value0  +  value1  +  value2 + ...

    By default, this function assumes a dark line (high values)
    surrounded by white (low values).  If your line is light on
    black, set the optional second argument white_line to true.  In
    this case, each sensor value will be replaced by (1000-value)
    before the averaging.
    """

    def readLine(self, white_line=0):
        sensor_values = self.readCalibrated()
        avg = 0
```

```
153.         sum = 0
154.         on_line = 0
155.         for i in range(0, self.numSensors):
156.             value = sensor_values[i]
157.             if (white_line):
158.                 value = 1000 - value
159.             # keep track of whether we see the line at all
160.             if (value > 200):
161.                 on_line = 1
162.
163.             # only average in values that are above a noise threshold
164.             if (value > 50):
165.                 avg += value * (i * 1000);  # this is for the weighted total,
166.                 sum += value;  # this is for the denominator
167.
168.         if (on_line != 1):
169.             # If it last read to the left of center, return 0.
170.             if (self.last_value < (self.numSensors - 1) * 1000 / 2):
171.                 # print("left")
172.                 return 0;
173.
174.             # If it last read to the right of center, return the max.
175.             else:
176.                 # print("right")
177.                 return (self.numSensors - 1) * 1000
178.
179.         self.last_value = avg / sum
180.         return self.last_value
181.
182. Ab = AlphaBot()
183. Ab.stop()
184. TR = TRSensor()
```

3. bug_0.py

Step:

(1) Turn the car's direction towards the goal.

(2) Go straight to the goal under detect obstacle or reach the goal. If reach the goal, the robot car stops, return success. If detects obstacle, record this point's coordinate as hit coordinate.

(3) Turn left and follow line tracking.

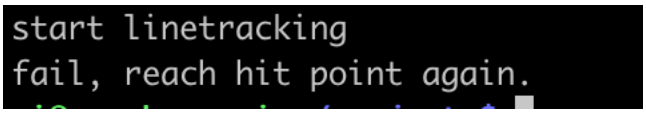
(4) During the line tracking, if the rotation angle is greater than a certain threshold (we assume the threshold is 70°), let the car follow the line tracking and move a small distance, then turn the car's direction towards the goal.

(5) Detect whether there is obstacle, if so, keep line tracking, else turn to step (2). Keep updating hit coordinate.

(6) If we meet the hit coordinate twice, return failure. The car will never reach the goal.

In this experiment, since we always put the car in the maze and put the goal out of the maze, the car will never reach the goal in this circumstance.

Result:




```python
import numpy as np
import math
import RPi.GPIO as GPIO
from utils import Ab, TR
import time

from client import coordinate, goal_location

cntl = 7
cntr = 8
EncL = 0
EncR = 0
speed = 30
last_proportional = 0
integral = 0

def updateEncoderL(channel):
    global EncL
    EncL += 1

def updateEncoderR(channel):
    global EncR
    EncR += 1

def get_angle(v1, v2): # calculate the angle between vector v1 and vector v2
```

```
    return math.acos(np.dot(v1, v2) / (norm2(v1, 0) * norm2(v2, 0))) * 360 / 2 / math.pi

def norm2(v1, v2):
    return np.linalg.norm(v1-v2, 2)

GPIO.setup(cntr, GPIO.IN)
GPIO.setup(cntl, GPIO.IN)
GPIO.add_event_detect(cntr, GPIO.BOTH, updateEncoderR)
GPIO.add_event_detect(cntl, GPIO.BOTH, updateEncoderL)

def rotation(angle, direction):
    global EncL, EncR
    ini_encl = EncL
    ini_encr = EncR

    d = 4.3  # the distance between two wheel
    if direction:  # turn left
        Ab.setMotor(-speed, -speed)
    else: # turn right
        Ab.setMotor(speed, speed)
    while True:
        distance_l = float(EncL - ini_encl) / 40 * 2 * math.pi * 1.5
        distance_r = float(EncR - ini_encr) / 40 * 2 * math.pi * 1.5
        if distance_l+distance_r >= d * angle * 2 * math.pi / 360:
            break
    # Ab.setMotor(-speed, speed)
    # time.sleep(0.3)
    Ab.stop()

def detect(): # if the sensor detects there is black line, return true, else return false
    sensor_values = TR.readCalibrated()
    on_line = 0
    for i in range(0, TR.numSensors):
        value = sensor_values[i]
        # keep track of whether we see the line at all
        if value > 200:
            on_line = 1
    if on_line == 1:
        return False
    else:
        return True

def linetracking(): # almost the same as project 1, we adjust the parameter of calculating
the power_difference
```

```
69.     global last_proportional
70.     global integral
71.     maximum = 35
72.     position = TR.readLine()
73.     Ab.backward()
74.     # The "proportional" term should be 0 when we are on the line.
75.     proportional = position - 2000
76.     # Compute the derivative (change) and integral (sum) of the position.
77.     derivative = proportional - last_proportional
78.     # integral += proportional
79.     last_proportional = proportional
80.     power_difference = proportional / 15 + derivative / 100  #+ integral/1000
81.     if power_difference > maximum:
82.         power_difference = maximum
83.     if power_difference < - maximum:
84.         power_difference = - maximum
85.     if power_difference < 0:
86.         Ab.setPWMB(maximum + power_difference)
87.         Ab.setPWMA(maximum)
88.     else:
89.         Ab.setPWMB(maximum)
90.         Ab.setPWMA(maximum - power_difference)
91.
92. def Bug_0(start, end, rvec):
93.     vec = end - start
94.     angle = get_angle(rvec, vec)
95.
96.     print angle
97.     time.sleep(1)
98.     direction = np.cross(rvec[[0, 2]], vec[[0, 2]]) > 0
99.     rotation(angle, direction)
100.
101.      last_hit_position = start
102.      while True:
103.          flag = 0
104.          while detect(): # if not detect obstacle, move straight to the goal
105.              Ab.setMotor(speed, -speed)
106.              current_position, current_direction = coordinate()
107.              distgoal = norm2(current_position, end) # calculate the distance between the
    current position and the goal
108.              if distgoal < 0.05:
109.                  flag = 1
110.                  break
111.
```

```
112.         Ab.stop()
113.         if flag == 1:
114.             print "reach goal"
115.             break
116.         rotation(120, 1)  # assume that the car turns left every time we hit the obstacle
117.
118.         while True:
119.             Ab.setMotor(speed, -speed - 2)
120.             if detect():
121.                 Ab.stop()
122.                 break
123.
124.         hit_position, hit_direction = coordinate()
125.
126.         if norm2(hit_position, last_hit_position) < 0.04:
127.             print "fail, reach hit point again."
128.             Ab.stop()
129.             return
130.
131.         last_hit_position = hit_position
132.
133.         current_position, current_direction = coordinate()
134.         goal_direction = end - current_position
135.
136.         af_encl = EncL
137.         af_encr = EncR
138.
139.         print "go straight"
140.         while True:
141.             linetracking()
142.             current_position, current_direction = coordinate()
143.             straight_distance = float(EncL - af_encl + EncR - af_encr) / 2 / 40
144.             if straight_distance >= 0.5 and get_angle(current_direction, hit_direction) >
    = 70:
145.                 break
146.
147.         af_encl = EncL
148.         af_encr = EncR
149.
150.         print "off straight"
151.         while True:
152.             linetracking()
153.             straight_distance = float(EncL - af_encl + EncR - af_encr) / 2 / 40
```

```
            print straight_distance
            if straight_distance >= 0.8:
                break

        Ab.stop()

        current_position, current_direction = coordinate()
        angle = get_angle(current_direction, goal_direction)
        # direction = np.cross(current_direction[[0, 2]], goal_direction[[0, 2]]) < 0
        direction = 1
        rotation(angle, direction)

if __name__ == "__main__":
    st, sr = coordinate()
    g = goal_location()
    Bug_0(st, g, sr)
```

4. bug_1.py

Step:

(1) Turn the car's direction towards the goal.

(2) Go straight to the goal under detect obstacle. Record this hit point's coordinate as hit coordinate.

(3) Start line tracking until go back to the hit point. During this process, keep calculating the distance between the current position and the goal and keep recording the minimum distance and its corresponding coordinates as target point (leave point).

(4) Keep line tracking until reach the target point.

(5) Turn the car's direction towards the goal.

(6) Go to the goal. If the distance between the current position and the goal is lower than a certain threshold, stop the car and return success. If detect obstacle, return failure.

```
import numpy as np
import AlphaBot
import math
import RPi.GPIO as GPIO
```

```
from utils import Ab, TR
import time

from client import coordinate, goal_location

cntl = 7
cntr = 8
EncL = 0
EncR = 0
speed = 30
last_proportional = 0
integral = 0

def updateEncoderL(channel):
    global EncL
    EncL += 1

def updateEncoderR(channel):
    global EncR
    EncR += 1

def get_angle(v1, v2):
    return math.acos(np.dot(v1, v2) / (norm2(v1, 0) * norm2(v2, 0))) * 360 / 2 / math.pi

def norm2(v1, v2):
    return np.linalg.norm(v1-v2, 2)

GPIO.setup(cntr, GPIO.IN)
GPIO.setup(cntl, GPIO.IN)
GPIO.add_event_detect(cntr, GPIO.BOTH, updateEncoderR)
GPIO.add_event_detect(cntl, GPIO.BOTH, updateEncoderL)

def rotation(angle, direction):
    global EncL, EncR
    ini_encl = EncL
    ini_encr = EncR
    # print ini_encl, ini_encr

    d = 4.75  # the distance between two wheel
    if direction:  # turn left
        Ab.setMotor(-speed, -speed)
    else:
        Ab.setMotor(speed, speed)
    while True:
```

```
49.         distance_l = float(EncL - ini_encl) / 40 * 2 * math.pi * 1.5
50.         distance_r = float(EncR - ini_encr) / 40 * 2 * math.pi * 1.5
51.         print distance_l, distance_r, angle
52.         if distance_l+distance_r >= d * angle * 2 * math.pi / 360:
53.             break
54.     Ab.setMotor(-speed, speed)
55.     time.sleep(0.3)
56.     Ab.stop()
57.
58. def auto_rotation():
59.     ct, cr = coordinate()
60.     g = goal_location()
61.     gr = g - ct
62.     direction = np.cross(cr[[0, 2]], gr[[0, 2]]) > 0
63.
64.     if direction:
65.         Ab.setMotor(-speed, -speed)
66.     else:
67.         Ab.setMotor(speed, speed)
68.
69.     while True:
70.         ct, cr = coordinate()
71.         gr = g - ct
72.         angle = get_angle(cr[[0,2]], gr[[0,2]])
73.         if angle < 10:
74.             break
75.     Ab.stop()
76.
77. def detect():
78.     sensor_values = TR.readCalibrated()
79.     on_line = 0
80.     for i in range(0, TR.numSensors):
81.         value = sensor_values[i]
82.         # keep track of whether we see the line at all
83.         if value > 200:
84.             on_line = 1
85.     if on_line == 1:
86.         return False
87.     else:
88.         return True
89.
90. def linetracking():
91.     global last_proportional
92.     global integral
```

```
    maximum = 30
    position = TR.readLine()
    Ab.backward()
    # The "proportional" term should be 0 when we are on the line.
    proportional = position - 2000
    # Compute the derivative (change) and integral (sum) of the position.
    derivative = proportional - last_proportional
     # integral += proportional
     last_proportional = proportional
     power_difference = proportional / 15 + derivative / 100  #+ integral/1000
     if power_difference > maximum:
         power_difference = maximum
     if power_difference < - maximum:
         power_difference = - maximum
     if power_difference < 0:
         Ab.setPWMB(maximum + power_difference)
         Ab.setPWMA(maximum)
     else:
         Ab.setPWMB(maximum)
         Ab.setPWMA(maximum - power_difference)

def Bug_1(start, end, rvec):
    # for i in range(0,400):
    # TR.calibrate()
    vec = end - start
    vec = vec[[0,2]]
    rvec = rvec[[0,2]]
    angle = get_angle(rvec, vec)

    print angle
    time.sleep(1)
    direction = np.cross(rvec, vec) > 0
    rotation(angle, direction)

    while detect(): # keep go towards the goal until detect obstacle
        Ab.setMotor(speed, -speed-2)

    Ab.stop()
    rotation(45, 0)
    print "start linetracking."
    time.sleep(2)

    af_encl = EncL
    af_encr = EncR
```

```
137.
138.     hit_position, _ = coordinate() # record the coordinate of hit position
139.     distmin = norm2(hit_position - end, 2)
140.     target = hit_position # initialize the target position as hit position
141.     straight_distance = 0
142.
143.     current_position = coordinate()
144.     disthit = 0
145.
146.     while straight_distance < 10 or disthit > 0.06:
147.         linetracking()
148.         current_position, _ = coordinate()
149.         current_position = current_position[:3]
150.
151.         dist = norm2(current_position, end)
152.         disthit = norm2(current_position, hit_position)
153.         if dist < distmin:
154.             distmin = dist
155.             target = current_position # update target position
156.         straight_distance = float(EncL - af_encl + EncR - af_encr) / 2 / 40
157.
158.     Ab.stop()
159.     distarget = norm2(current_position, target)
160.
161.     while distarget > 0.03: # go to target position
162.         linetracking()
163.         current_position, current_direction = coordinate()
164.         distarget = norm2(current_position, target)
165.     Ab.stop()
166.
167.     current_position, current_direction = coordinate()
168.     target_direction = goal_location() - current_position
169.
170.     goal_angle = get_angle(current_position, target_direction)
171.     goal_direction = np.cross(current_direction[[0, 2]], target_direction[[0, 2]]) > 0
172.
173.     Ab.stop()
174.     rotation(goal_angle, goal_direction)
175.
176.     count = 0
177.     Ab.setMotor(speed, -speed - 2)
178.     while True:
179.         current_position, _ = coordinate()
180.         distgoal = norm2(current_position, end)
```

```
        if distgoal <= 0.04:
            break

        if not detect():
            print("fail")
            return 0
    Ab.stop()

if __name__ == '__main__':
    g = goal_location()
    s, r = coordinate()
    Bug_1(s, g, r)
```

5. bug_2.py

Step:

(1) Turn the car's direction towards the goal.

(2) Go straight to the goal under detect obstacle or reach the goal. If reach the goal, the robot car stop, return success. If detect obstacle, record this hit point's coordinate as hit coordinate. Calculate the distance between the hit point and the goal as hit distance.

(3) Keep line tracking until the start point, goal and current position is on one line.

(4) Calculate the distance between this point and the goal as leave distance. If leave distance is lower than hit distance, turn to step (1), else keep line tracking.

(5) If we go to one hit point twice, return failure.

```
import numpy as np
import AlphaBot
import math
import RPi.GPIO as GPIO
from utils import Ab, TR
import time
from client import coordinate, goal_location

cntl = 7
cntr = 8
EncL = 0
```

```
EncR = 0
speed = 30
last_proportional = 0
integral = 0

def updateEncoderL(channel):
    global EncL
    EncL += 1

def updateEncoderR(channel):
    global EncR
    EncR += 1

def get_angle(v1, v2):
    return math.acos(np.dot(v1, v2) / (norm2(v1, 0) * norm2(v2, 0))) * 360 / 2 / math.pi

def norm2(v1, v2):
    return np.linalg.norm(v1-v2, 2)

GPIO.setup(cntr, GPIO.IN)
GPIO.setup(cntl, GPIO.IN)
GPIO.add_event_detect(cntr, GPIO.BOTH, updateEncoderR)
GPIO.add_event_detect(cntl, GPIO.BOTH, updateEncoderL)

def rotation(angle, direction):
    global EncL, EncR
    ini_encl = EncL
    ini_encr = EncR
    # print ini_encl, ini_encr

    d = 4.75  # the distance between two wheel

    if direction:  # turn left
        Ab.setMotor(-speed, -speed)
    else:
        Ab.setMotor(speed, speed)
    while True:
        distance_l = float(EncL - ini_encl) / 40 * 2 * math.pi * 1.5
        distance_r = float(EncR - ini_encr) / 40 * 2 * math.pi * 1.5
        print distance_l, distance_r, angle
        if distance_l+distance_r >= d * angle * 2 * math.pi / 360:
            break
    Ab.setMotor(-speed, speed)
    time.sleep(0.3)
```

```
    Ab.stop()

def detect():
    sensor_values = TR.readCalibrated()
    on_line = 0
    for i in range(0, TR.numSensors):
        value = sensor_values[i]
        # keep track of whether we see the line at all
        if value > 200:
            on_line = 1
    if on_line == 1:
        return False
    else:
        return True

def linetracking():
    global last_proportional
    global integral
    maximum = 30
    position = TR.readLine()
    Ab.backward()
    # The "proportional" term should be 0 when we are on the line.
    proportional = position - 2000
    # Compute the derivative (change) and integral (sum) of the position.
    derivative = proportional - last_proportional
    # integral += proportional
    last_proportional = proportional
    power_difference = proportional / 15 + derivative / 100  #+ integral/1000
    if power_difference > maximum:
        power_difference = maximum
    if power_difference < - maximum:
        power_difference = - maximum
    if power_difference < 0:
        Ab.setPWMB(maximum + power_difference)
        Ab.setPWMA(maximum)
    else:
        Ab.setPWMB(maximum)
        Ab.setPWMA(maximum - power_difference)

def oneline(pone, ptwo, pthree): # detect whether these three points are on one line
    thr = 0.03
    d1 = norm2(pone,ptwo)
    d2 = norm2(pone,pthree)
    d3 = norm2(ptwo,pthree)
```

```
100.     print "[[on line:]]", d1, d2, d3
101.     if abs (d1 + d2 - d3) > thr and abs (d1 + d3 - d2) > thr and abs (d2 + d3 - d1) > thr
    :
102.         return False
103.     else:
104.         return True
105.
106. def Bug_2(start, end, rvec):
107.     # for i in range(0,400):
108.     # TR.calibrate()
109.     vec = end - start
110.     vec = vec[[0,2]]
111.     rvec = rvec[[0,2]]
112.     angle = get_angle(rvec, vec)
113.
114.     print angle
115.     time.sleep(1)
116.     direction = np.cross(rvec, vec) > 0
117.     rotation(angle, direction)
118.
119.     flag = 0
120.     while True:
121.         while detect():
122.             Ab.setMotor(speed, -speed-2)
123.             current_position, current_direction = coordinate()
124.             distgoal = norm2(current_position, end)
125.             if distgoal < 0.05:
126.                 flag = 1
127.                 break
128.         Ab.stop()
129.         if flag == 1:
130.             print "reach goal"
131.             break
132.         rotation(45, 0) # assume that the car turns right every time we hit the obstacle

133.
134.         hit_position, _ = coordinate()
135.         hitdist = norm2(hit_position, end)
136.         current_position, current_direction = coordinate()
137.         goal_direction = end - current_position
138.
139.         while True:
140.             af_encl = EncL
141.             af_encr = EncR
```

```
142.
143.             while True:
144.                 current_position, current_direction = coordinate()
145.                 linetracking()
146.                 straight_distance = float(EncL - af_encl + EncR - af_encr) / 2 / 40
147.                 if straight_distance > 2 and oneline(start, end, current_position):
148.                     break
149.
150.             current_position, current_direction = coordinate()
151.             leavedist = norm2(current_position, end)
152.             Ab.stop()
153.
154.             if oneline(start, end, current_position):
155.                 print "on line"
156.                 angle = get_angle(current_direction, goal_direction)
157.                 if hitdist > leavedist:
158.                     turndir = np.cross(current_direction[[0, 2]], goal_direction[[0, 2]])
    > 0
159.                     rotation(angle, turndir)
160.
161.                     if not detect():
162.                         print "hit obstacle"
163.                         rotation(get_angle(current_direction, end - current_position), no
   t turndir)
164.                     break
165.
166.                 if abs(hitdist - leavedist) < 0.01:
167.                     print "fail"
168.                     break
169.
170.     Ab.stop()
171.
172. if __name__ == '__main__':
173.     g = goal_location()
174.     s, r = coordinate()
175.     Bug_2(s, g, r)
```