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:

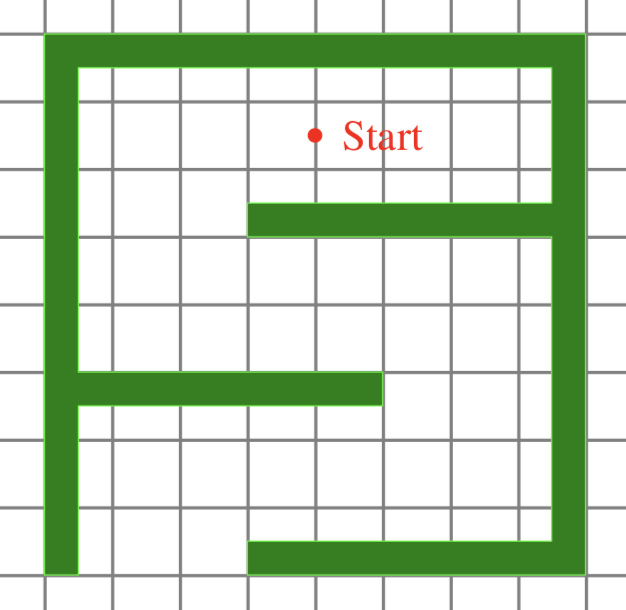
1. The map is imperfect, so it always leads to the failure of the line tracking. We fixed the map many times in order to make sure the line tracking could be done. Since the main point of this experiment is to implement bug algorithms, whenever the car fails to track the line, we manually fix it. You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



Goal

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.

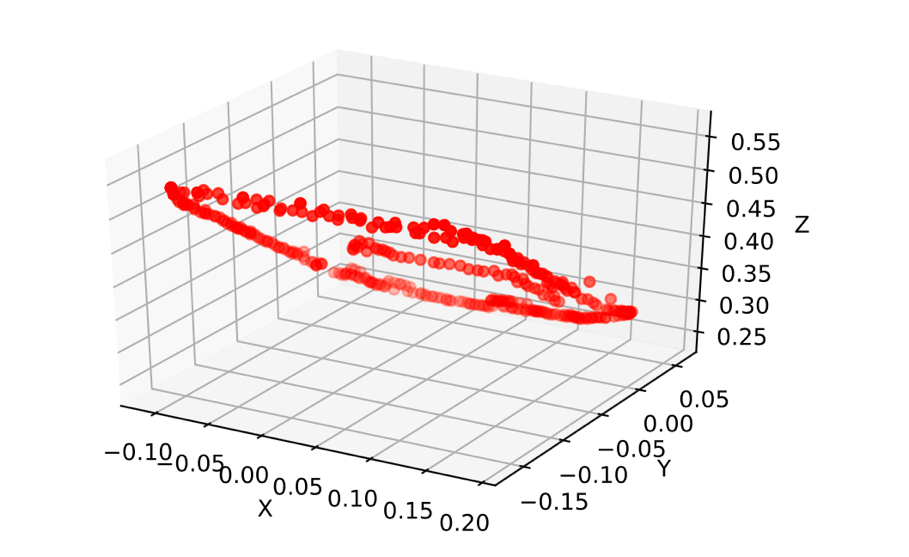
3. Under what conditions can Bug 2 algorithm find the goal?

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
8. server\_ip = '10.107.208.86'
10. channel = grpc.insecure\_channel('{}:50051'.format(server\_ip))
11. stub = aruco\_pb2\_grpc.GreeterStub(channel)
13. **def** check\_connect():
14. response = stub.SayHello(aruco\_pb2.Empty())
15. **assert** response **is** **not** None
17. **def** coordinate():
18. c = stub.Ask(aruco\_pb2.Empty())
19. rvec = [c.r.x, c.r.y, c.r.z]
20. car\_front, \_ = cv2.Rodrigues(np.array(rvec))
21. tvec = [c.t.x, c.t.y, c.t.z]
22. **return** np.array(tvec), car\_front[:, 0]
24. **def** goal\_location():
25. c = stub.Goal(aruco\_pb2.Empty())
26. tvec = [c.t.x, c.t.y, c.t.z]
27. **return** np.array(tvec)
28. Configure Environment

utils.py

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

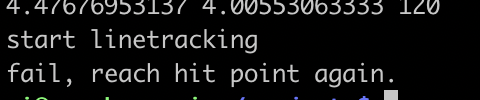
1. **import** RPi.GPIO as GPIO
2. **from** AlphaBot **import** AlphaBot
3. **import** numpy as np
5. CS = 5
6. Clock = 25
7. Address = 24
8. DataOut = 23
10. GPIO.setmode(GPIO.BCM)
11. GPIO.setwarnings(False)
12. GPIO.setup(Clock,GPIO.OUT)
13. GPIO.setup(Address,GPIO.OUT)
14. GPIO.setup(CS,GPIO.OUT)
15. GPIO.setup(DataOut,GPIO.IN,GPIO.PUD\_UP)
17. **class** TRSensor(object):
18. **def** \_\_init\_\_(self, numSensors=5):
19. self.numSensors = numSensors
20. self.calibratedMin = [967, 975, 972, 974, 984]
21. self.calibratedMax = [190, 254, 243, 233, 461]
22. self.last\_value = 0
24. """
25. Reads the sensor values into an array. There \*MUST\* be space
26. for as many values as there were sensors specified in the constructor.
27. Example usage:
28. unsigned int sensor\_values[8];
29. sensors.read(sensor\_values);
30. The values returned are a measure of the reflectance in abstract units,
31. with higher values corresponding to lower reflectance (e.g. a black
32. surface or a void).
33. """
35. **def** AnalogRead(self):
36. value = [0, 0, 0, 0, 0, 0]
37. # Read Channel0~channel4 AD value
38. **for** j **in** range(0, 6):
39. GPIO.output(CS, GPIO.LOW)
40. **for** i **in** range(0, 4):
41. # sent 4-bit Address
42. **if** (((j) >> (3 - i)) & 0x01):
43. GPIO.output(Address, GPIO.HIGH)
44. **else**:
45. GPIO.output(Address, GPIO.LOW)
46. # read MSB 4-bit data
47. value[j] <<= 1
48. **if** (GPIO.input(DataOut)):
49. value[j] |= 0x01
50. GPIO.output(Clock, GPIO.HIGH)
51. GPIO.output(Clock, GPIO.LOW)
52. **for** i **in** range(0, 6):
53. # read LSB 8-bit data
54. value[j] <<= 1
55. **if** (GPIO.input(DataOut)):
56. value[j] |= 0x01
57. GPIO.output(Clock, GPIO.HIGH)
58. GPIO.output(Clock, GPIO.LOW)
59. # no mean ,just delay
60. **for** i **in** range(0, 6):
61. GPIO.output(Clock, GPIO.HIGH)
62. GPIO.output(Clock, GPIO.LOW)
63. #           time.sleep(0.0001)
64. GPIO.output(CS, GPIO.HIGH)
65. **return** value[1:]
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. """
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):
79. sensor\_values = self.AnalogRead();
81. **for** i **in** range(0, self.numSensors):
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]
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]
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]
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. """
106. **def** readCalibrated(self):
107. value = 0
108. # read the needed values
109. sensor\_values = self.AnalogRead();
111. **for** i **in** range(0, self.numSensors):
113. denominator = self.calibratedMax[i] - self.calibratedMin[i]
115. **if** (denominator != 0):
116. value = (sensor\_values[i] - self.calibratedMin[i]) \* 1000 / denominator
118. **if** (value < 0):
119. value = 0
120. **elif** (value > 1000):
121. value = 1000
123. sensor\_values[i] = value
125. # print("readCalibrated", sensor\_values)
126. **return** sensor\_values
128. """
129. Operates the same as read calibrated, but also returns an
130. estimated position of the robot with respect to a line. The
131. estimate is made using a weighted average of the sensor indices
132. multiplied by 1000, so that a return value of 0 indicates that
133. the line is directly below sensor 0, a return value of 1000
134. indicates that the line is directly below sensor 1, 2000
135. indicates that it's below sensor 2000, etc.  Intermediate
136. values indicate that the line is between two sensors.  The
137. formula is:
139. 0\*value0 + 1000\*value1 + 2000\*value2 + ...
140. --------------------------------------------
141. value0  +  value1  +  value2 + ...
143. By default, this function assumes a dark line (high values)
144. surrounded by white (low values).  If your line is light on
145. black, set the optional second argument white\_line to true.  In
146. this case, each sensor value will be replaced by (1000-value)
147. before the averaging.
148. """
150. **def** readLine(self, white\_line=0):
151. sensor\_values = self.readCalibrated()
152. 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
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
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;
174. # If it last read to the right of center, return the max.
175. **else**:
176. # print("right")
177. **return** (self.numSensors - 1) \* 1000
179. self.last\_value = avg / sum
180. **return** self.last\_value
182. Ab = AlphaBot()
183. Ab.stop()
184. TR = TRSensor()
185. 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:



1. **import** numpy as np
2. **import** math
3. **import** RPi.GPIO as GPIO
4. **from** utils **import** Ab, TR
5. **import** time
7. **from** client **import** coordinate, goal\_location
9. cntl = 7
10. cntr = 8
11. EncL = 0
12. EncR = 0
13. speed = 30
14. last\_proportional = 0
15. integral = 0
17. **def** updateEncoderL(channel):
18. **global** EncL
19. EncL += 1
21. **def** updateEncoderR(channel):
22. **global** EncR
23. EncR += 1
25. **def** get\_angle(v1, v2): # calculate the angle between vector v1 and vector v2
26. **return** math.acos(np.dot(v1, v2) / (norm2(v1, 0) \* norm2(v2, 0))) \* 360 / 2 / math.pi
28. **def** norm2(v1, v2):
29. **return** np.linalg.norm(v1-v2, 2)
31. GPIO.setup(cntr, GPIO.IN)
32. GPIO.setup(cntl, GPIO.IN)
33. GPIO.add\_event\_detect(cntr, GPIO.BOTH, updateEncoderR)
34. GPIO.add\_event\_detect(cntl, GPIO.BOTH, updateEncoderL)
36. **def** rotation(angle, direction):
37. **global** EncL, EncR
38. ini\_encl = EncL
39. ini\_encr = EncR
41. d = 4.3  # the distance between two wheel
42. **if** direction:  # turn left
43. Ab.setMotor(-speed, -speed)
44. **else**: # turn right
45. Ab.setMotor(speed, speed)
46. **while** True:
47. distance\_l = float(EncL - ini\_encl) / 40 \* 2 \* math.pi \* 1.5
48. distance\_r = float(EncR - ini\_encr) / 40 \* 2 \* math.pi \* 1.5
49. **if** distance\_l+distance\_r >= d \* angle \* 2 \* math.pi / 360:
50. **break**
51. # Ab.setMotor(-speed, speed)
52. # time.sleep(0.3)
53. Ab.stop()
55. **def** detect(): # if the sensor detects there is black line, return true, else return false
56. sensor\_values = TR.readCalibrated()
57. on\_line = 0
58. **for** i **in** range(0, TR.numSensors):
59. value = sensor\_values[i]
60. # keep track of whether we see the line at all
61. **if** value > 200:
62. on\_line = 1
63. **if** on\_line == 1:
64. **return** False
65. **else**:
66. **return** True
68. **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)
92. **def** Bug\_0(start, end, rvec):
93. vec = end - start
94. angle = get\_angle(rvec, vec)
96. **print** angle
97. time.sleep(1)
98. direction = np.cross(rvec[[0, 2]], vec[[0, 2]]) > 0
99. rotation(angle, direction)
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**
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
118. **while** True:
119. Ab.setMotor(speed, -speed - 2)
120. **if** detect():
121. Ab.stop()
122. **break**
124. hit\_position, hit\_direction = coordinate()
126. **if** norm2(hit\_position, last\_hit\_position) < 0.04:
127. **print** "fail, reach hit point again."
128. Ab.stop()
129. **return**
131. last\_hit\_position = hit\_position
133. current\_position, current\_direction = coordinate()
134. goal\_direction = end - current\_position
136. af\_encl = EncL
137. af\_encr = EncR
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**
147. af\_encl = EncL
148. af\_encr = EncR
150. **print** "off straight"
151. **while** True:
152. linetracking()
153. straight\_distance = float(EncL - af\_encl + EncR - af\_encr) / 2 / 40
154. **print** straight\_distance
155. **if** straight\_distance >= 0.8:
156. **break**
158. Ab.stop()
160. current\_position, current\_direction = coordinate()
161. angle = get\_angle(current\_direction, goal\_direction)
162. # direction = np.cross(current\_direction[[0, 2]], goal\_direction[[0, 2]]) < 0
163. direction = 1
164. rotation(angle, direction)
166. **if** \_\_name\_\_ == "\_\_main\_\_":
167. st, sr = coordinate()
168. g = goal\_location()
169. Bug\_0(st, g, sr)
170. 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.
7. **import** numpy as np
8. **import** AlphaBot
9. **import** math
10. **import** RPi.GPIO as GPIO
11. **from** utils **import** Ab, TR
12. **import** time
14. **from** client **import** coordinate, goal\_location
16. cntl = 7
17. cntr = 8
18. EncL = 0
19. EncR = 0
20. speed = 30
21. last\_proportional = 0
22. integral = 0
24. **def** updateEncoderL(channel):
25. **global** EncL
26. EncL += 1
28. **def** updateEncoderR(channel):
29. **global** EncR
30. EncR += 1
32. **def** get\_angle(v1, v2):
33. **return** math.acos(np.dot(v1, v2) / (norm2(v1, 0) \* norm2(v2, 0))) \* 360 / 2 / math.pi
35. **def** norm2(v1, v2):
36. **return** np.linalg.norm(v1-v2, 2)
38. GPIO.setup(cntr, GPIO.IN)
39. GPIO.setup(cntl, GPIO.IN)
40. GPIO.add\_event\_detect(cntr, GPIO.BOTH, updateEncoderR)
41. GPIO.add\_event\_detect(cntl, GPIO.BOTH, updateEncoderL)
43. **def** rotation(angle, direction):
44. **global** EncL, EncR
45. ini\_encl = EncL
46. ini\_encr = EncR
47. # print ini\_encl, ini\_encr
49. d = 4.75  # the distance between two wheel
50. **if** direction:  # turn left
51. Ab.setMotor(-speed, -speed)
52. **else**:
53. Ab.setMotor(speed, speed)
54. **while** True:
55. distance\_l = float(EncL - ini\_encl) / 40 \* 2 \* math.pi \* 1.5
56. distance\_r = float(EncR - ini\_encr) / 40 \* 2 \* math.pi \* 1.5
57. **print** distance\_l, distance\_r, angle
58. **if** distance\_l+distance\_r >= d \* angle \* 2 \* math.pi / 360:
59. **break**
60. Ab.setMotor(-speed, speed)
61. time.sleep(0.3)
62. Ab.stop()
64. **def** auto\_rotation():
65. ct, cr = coordinate()
66. g = goal\_location()
67. gr = g - ct
68. direction = np.cross(cr[[0, 2]], gr[[0, 2]]) > 0
70. **if** direction:
71. Ab.setMotor(-speed, -speed)
72. **else**:
73. Ab.setMotor(speed, speed)
75. **while** True:
76. ct, cr = coordinate()
77. gr = g - ct
78. angle = get\_angle(cr[[0,2]], gr[[0,2]])
79. **if** angle < 10:
80. **break**
81. Ab.stop()
83. **def** detect():
84. sensor\_values = TR.readCalibrated()
85. on\_line = 0
86. **for** i **in** range(0, TR.numSensors):
87. value = sensor\_values[i]
88. # keep track of whether we see the line at all
89. **if** value > 200:
90. on\_line = 1
91. **if** on\_line == 1:
92. **return** False
93. **else**:
94. **return** True
96. **def** linetracking():
97. **global** last\_proportional
98. **global** integral
99. maximum = 30
100. position = TR.readLine()
101. Ab.backward()
102. # The "proportional" term should be 0 when we are on the line.
103. proportional = position - 2000
104. # Compute the derivative (change) and integral (sum) of the position.
105. derivative = proportional - last\_proportional
106. # integral += proportional
107. last\_proportional = proportional
108. power\_difference = proportional / 15 + derivative / 100  #+ integral/1000
109. **if** power\_difference > maximum:
110. power\_difference = maximum
111. **if** power\_difference < - maximum:
112. power\_difference = - maximum
113. **if** power\_difference < 0:
114. Ab.setPWMB(maximum + power\_difference)
115. Ab.setPWMA(maximum)
116. **else**:
117. Ab.setPWMB(maximum)
118. Ab.setPWMA(maximum - power\_difference)
120. **def** Bug\_1(start, end, rvec):
121. # for i in range(0,400):
122. # TR.calibrate()
123. vec = end - start
124. vec = vec[[0,2]]
125. rvec = rvec[[0,2]]
126. angle = get\_angle(rvec, vec)
128. **print** angle
129. time.sleep(1)
130. direction = np.cross(rvec, vec) > 0
131. rotation(angle, direction)
133. **while** detect(): # keep go towards the goal until detect obstacle
134. Ab.setMotor(speed, -speed-2)
136. Ab.stop()
137. rotation(45, 0)
138. **print** "start linetracking."
139. time.sleep(2)
141. af\_encl = EncL
142. af\_encr = EncR
144. hit\_position, \_ = coordinate() # record the coordinate of hit position
145. distmin = norm2(hit\_position - end, 2)
146. target = hit\_position # initialize the target position as hit position
147. straight\_distance = 0
149. current\_position = coordinate()
150. disthit = 0
152. **while** straight\_distance < 10 **or** disthit > 0.06:
153. linetracking()
154. current\_position, \_ = coordinate()
155. current\_position = current\_position[:3]
157. dist = norm2(current\_position, end)
158. disthit = norm2(current\_position, hit\_position)
159. **if** dist < distmin:
160. distmin = dist
161. target = current\_position # update target position
162. straight\_distance = float(EncL - af\_encl + EncR - af\_encr) / 2 / 40
164. Ab.stop()
165. distarget = norm2(current\_position, target)
167. **while** distarget > 0.03: # go to target position
168. linetracking()
169. current\_position, current\_direction = coordinate()
170. distarget = norm2(current\_position, target)
171. Ab.stop()
173. current\_position, current\_direction = coordinate()
174. target\_direction = goal\_location() - current\_position
176. goal\_angle = get\_angle(current\_position, target\_direction)
177. goal\_direction = np.cross(current\_direction[[0, 2]], target\_direction[[0, 2]]) > 0
179. Ab.stop()
180. rotation(goal\_angle, goal\_direction)
182. count = 0
183. Ab.setMotor(speed, -speed - 2)
184. **while** True:
185. current\_position, \_ = coordinate()
186. distgoal = norm2(current\_position, end)
187. **if** distgoal <= 0.04:
188. **break**
190. **if** **not** detect():
191. **print**("fail")
192. **return** 0
193. Ab.stop()
195. **if** \_\_name\_\_ == '\_\_main\_\_':
196. g = goal\_location()
197. s, r = coordinate()
198. Bug\_1(s, g, r)
199. 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.
6. **import** numpy as np
7. **import** AlphaBot
8. **import** math
9. **import** RPi.GPIO as GPIO
10. **from** utils **import** Ab, TR
11. **import** time
12. **from** client **import** coordinate, goal\_location
14. cntl = 7
15. cntr = 8
16. EncL = 0
17. EncR = 0
18. speed = 30
19. last\_proportional = 0
20. integral = 0
22. **def** updateEncoderL(channel):
23. **global** EncL
24. EncL += 1
26. **def** updateEncoderR(channel):
27. **global** EncR
28. EncR += 1
30. **def** get\_angle(v1, v2):
31. **return** math.acos(np.dot(v1, v2) / (norm2(v1, 0) \* norm2(v2, 0))) \* 360 / 2 / math.pi
33. **def** norm2(v1, v2):
34. **return** np.linalg.norm(v1-v2, 2)
36. GPIO.setup(cntr, GPIO.IN)
37. GPIO.setup(cntl, GPIO.IN)
38. GPIO.add\_event\_detect(cntr, GPIO.BOTH, updateEncoderR)
39. GPIO.add\_event\_detect(cntl, GPIO.BOTH, updateEncoderL)
41. **def** rotation(angle, direction):
42. **global** EncL, EncR
43. ini\_encl = EncL
44. ini\_encr = EncR
45. # print ini\_encl, ini\_encr
47. d = 4.75  # the distance between two wheel
49. **if** direction:  # turn left
50. Ab.setMotor(-speed, -speed)
51. **else**:
52. Ab.setMotor(speed, speed)
53. **while** True:
54. distance\_l = float(EncL - ini\_encl) / 40 \* 2 \* math.pi \* 1.5
55. distance\_r = float(EncR - ini\_encr) / 40 \* 2 \* math.pi \* 1.5
56. **print** distance\_l, distance\_r, angle
57. **if** distance\_l+distance\_r >= d \* angle \* 2 \* math.pi / 360:
58. **break**
59. Ab.setMotor(-speed, speed)
60. time.sleep(0.3)
61. Ab.stop()
63. **def** detect():
64. sensor\_values = TR.readCalibrated()
65. on\_line = 0
66. **for** i **in** range(0, TR.numSensors):
67. value = sensor\_values[i]
68. # keep track of whether we see the line at all
69. **if** value > 200:
70. on\_line = 1
71. **if** on\_line == 1:
72. **return** False
73. **else**:
74. **return** True
76. **def** linetracking():
77. **global** last\_proportional
78. **global** integral
79. maximum = 30
80. position = TR.readLine()
81. Ab.backward()
82. # The "proportional" term should be 0 when we are on the line.
83. proportional = position - 2000
84. # Compute the derivative (change) and integral (sum) of the position.
85. derivative = proportional - last\_proportional
86. # integral += proportional
87. last\_proportional = proportional
88. power\_difference = proportional / 15 + derivative / 100  #+ integral/1000
89. **if** power\_difference > maximum:
90. power\_difference = maximum
91. **if** power\_difference < - maximum:
92. power\_difference = - maximum
93. **if** power\_difference < 0:
94. Ab.setPWMB(maximum + power\_difference)
95. Ab.setPWMA(maximum)
96. **else**:
97. Ab.setPWMB(maximum)
98. Ab.setPWMA(maximum - power\_difference)
100. **def** oneline(pone, ptwo, pthree): # detect whether these three points are on one line
101. thr = 0.03
102. d1 = norm2(pone,ptwo)
103. d2 = norm2(pone,pthree)
104. d3 = norm2(ptwo,pthree)
105. **print** "[[on line:]]", d1, d2, d3
106. **if** abs (d1 + d2 - d3) > thr **and** abs (d1 + d3 - d2) > thr **and** abs (d2 + d3 - d1) > thr:
107. **return** False
108. **else**:
109. **return** True
111. **def** Bug\_2(start, end, rvec):
112. # for i in range(0,400):
113. # TR.calibrate()
114. vec = end - start
115. vec = vec[[0,2]]
116. rvec = rvec[[0,2]]
117. angle = get\_angle(rvec, vec)
119. **print** angle
120. time.sleep(1)
121. direction = np.cross(rvec, vec) > 0
122. rotation(angle, direction)
124. flag = 0
125. **while** True:
126. **while** detect():
127. Ab.setMotor(speed, -speed-2)
128. current\_position, current\_direction = coordinate()
129. distgoal = norm2(current\_position, end)
130. **if** distgoal < 0.05:
131. flag = 1
132. **break**
133. Ab.stop()
134. **if** flag == 1:
135. **print** "reach goal"
136. **break**
137. rotation(45, 0) # assume that the car turns right every time we hit the obstacle
139. hit\_position, \_ = coordinate()
140. hitdist = norm2(hit\_position, end)
141. current\_position, current\_direction = coordinate()
142. goal\_direction = end - current\_position
144. **while** True:
145. af\_encl = EncL
146. af\_encr = EncR
148. **while** True:
149. current\_position, current\_direction = coordinate()
150. linetracking()
151. straight\_distance = float(EncL - af\_encl + EncR - af\_encr) / 2 / 40
152. **if** straight\_distance > 2 **and** oneline(start, end, current\_position):
153. **break**
155. current\_position, current\_direction = coordinate()
156. leavedist = norm2(current\_position, end)
157. Ab.stop()
159. **if** oneline(start, end, current\_position):
160. **print** "on line"
161. angle = get\_angle(current\_direction, goal\_direction)
162. **if** hitdist > leavedist:
163. turndir = np.cross(current\_direction[[0, 2]], goal\_direction[[0, 2]]) > 0
164. rotation(angle, turndir)
166. **if** **not** detect():
167. **print** "hit obstacle"
168. rotation(get\_angle(current\_direction, end - current\_position), **not** turndir)
169. **break**
171. **if** abs(hitdist - leavedist) < 0.01:
172. **print** "fail"
173. **break**
175. Ab.stop()
177. **if** \_\_name\_\_ == '\_\_main\_\_':
178. g = goal\_location()
179. s, r = coordinate()
180. Bug\_2(s, g, r)