Lab 5 Report

Group 7: Yu GU (yg2466), Boyuan Chen (bc2699), Yueting Lu(yl3607)

Question 1:

Write a program to do 2-D collision-free motion planning.
Take as input a file which contains sets of ordered vertices of convex polygons which represents a set of obstacles in your world, along with a bounding box around the entire environment.
Code implementation found in roborace.py

Grow these obstacles by the size of the GoPiGo robot using the reflection algorithm and the convex hull algorithm discussed in class.
Code implementation found in Grow_Obstacle.py and ConvexHull.py

Then, given an arbitrary start and end point in this environment, create a Visibility graph on these grown obstacles, and search it using Dijkstra's algorithm to find a safe, collision free path.
Code implementation found in generate_visibility.py

Draw the graph. Code implementation found in roborace.py

Disclaimer:

We implemented the particle filter algorithm according to the instructions provided. Growing obstacle is based on the assumption that the cart is enclosed by a spherical capsule.

User Instruction:

1. As per request of the additional note, the configure file need to be specified in the commandline as below for the main python module roborace.py

```
python roborace.py configure.txt
```

2. After running the roborace.py, the program will generate path file named 'Route_For_race.txt'. Put both this file and 'Cart_Route.py' into gopigo. Then in gopigo:

```
Python Cart_Route.py
```

It should be noted custom name for path is not supported as its chained with roborace.py.

3. cart_offline.py is used to generate corresponding command for debugging purposes. Running it would generate a. the encoder count cart need to turn and L/R of the turn b. the distance the cart need to travel based on the path file generated by roborace.py

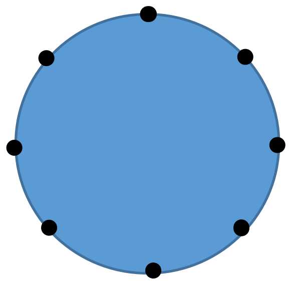
Video Link
Graph: https://www.youtube.com/watch?v=BcYaZNr_kxY
Navigation: https://www.youtube.com/watch?v=3C-Wb99ZQbM&feature=youtu.be

Implementation:

We implemented the algorithms provided in slides with some modifications and adjustments. Detailed implementation will be explained per module (rather than per function)

Modules:

1. Grow_Obstacle.py
   Obstacles was grown assuming the cart is a big spherical capsule. The diameter of the cart was 32cm (Diagonals), thus we calculated the radius to be 16cm. The capsule was denoted using a total of 8 points.



2. ConvexHull.py
   Same algorithm as discussed in class and in handout. mycomp is custom sorting used to sort points by angle and distance.

3. generate_visibility.py
   a. Path finding used Dij as discussed in class and in handout.
   b. Finding proper edges, meanwhile, was done in the following ways:
      Step 1. For each pair of nodes, divide the edge between them using binary divide (up to 2^6 - 1= 63 subpoints)
      Step 2. For each point, check to see if it's within an obstacle
         If yes, Return True – skip the line
         If not, Return False and plot the line on the graph

   *Above is the generic idea of the implementation, not explained according to the actual lines of codes. In fact, actual codes are broken into smaller functions.

4. roborace.py
   Read in file and plot animated graph, then generates output file for future usage.

5. cart_offline.py
   The route for part 2 can be generated using cart_offline.py. It creates the same movement specs as Cart_route.py. The only difference is the movement functions for gopigo were removed in the cart_offline.

Discussion:

1. The graph generation and path finding works perfectly. In the meantime, the binary_cnt in the generate_visibility.py can be adjusted to higher value (total subpoints = 2^ binary_cnt – 1) in order to obtain more accurate graphs.

2. Odometry error appears to be very significant. Even after trimming the movement is not stable. Meanwhile, movement per encoder count is not consistent. It could have been caused by the fact that the parts of the cart are becoming loose. Adjusting is quite difficult…

Extra

1. In Cart_route.py
   When the cart is moving, it would turn the servo to its left (map specific). If an obstacle is present within 10cms, it would turn to right by 2 encoder count (24 degrees on average).
   Then it would move forward by 2 encoder count and turn left by 2 encoder count.
   This is to ensure the cart can steer away from the obstacle with minimal influence.

2. cam_way_point.py
   cam_way_point.py would test to see if the center of the target blob is along the center of the image. If it is, it would return True. Otherwise, it kept spinning for the target.
   In Cart_route.py, the code would check for successful alignment, if the cam_way_point returned True, then the cart would move forward within 10cm of the real goal.