Robot Planning and its Applications a.y. 2020/2021

•••

Matteo Malacarne mat[203311]

GitHub Repo

Focus points (a.k.a. main assumptions & decisions)

- Camera calibration;
- Digit recognition;
- Collision detection & avoidance;
- Motion planning;
- Mission 1 & Mission 2;

Camera calibration

Design decision

Distortion coefficients are hard coded in *src/extrinsicCalib.cpp* (i.e. The project relies on the “Chessboard procedure” as seen in Lab. Sept, 30th 2020).

How & Where

- Run *camera_calibration.cpp* with the appropriate *calib_config.xml*;
- Copy distortion coefficients’ values *from intrinsic_calibration.xml*;
- Paste values in the appropriate variables in *src/extrinsicCalib.cpp*.

```
/*!
* Flag to determine dist_coeffs values. If def -> dist_coeffs = [0,0,0,0,0]
*/
#define DIST_COEFFS_DEFAULT
```

```
// Copied & pasted dist coeffs value from intrinsic_calibration.xml
double dc_k1 = -3.8003887070277098e-01;
double dc_k2 = 1.6491942975982371e-01;
double dc_k3 = -7.2969848408770512e-04;
double dc_p1 = -8.3850307681785957e-04;
double dc_p2 = 0.;

dist_coeffs = (cv::Mat1d(1,4) << dc_k1, dc_k2, dc_k3, dc_p1, dc_p2);
```

Digit recognition

Design assumption

Provided a known manipulation, all victims' digits underwent the same "deformation".

How & Where

The manipulation can be described by 3 parameters:

- Rotation angle (expressed in degrees) on the center of the digit.
- Digit flipped on the x-axis;
- Digit flipped on the y-axis;

Digit can then be recognized via template matching technique.

```
/*!
 * Rotation angle to apply to the ROI of a detected digit.
 */
#define ANGLE 90

/*!
 * True if the ROI has to be flipped on the x axis, false otherwise.
 */
#define X_FLIP false

/*!
 * True if the ROI has to be flipped on the y axis, false otherwise.
 */
#define Y_FLIP true
```

```
    getEditedROI(orig_ROI, ANGLE, X_FLIP, Y_FLIP, edited_ROI);
    getTMDigit(edited_ROI, template_images, victim_id);
```

Collision detection & avoidance

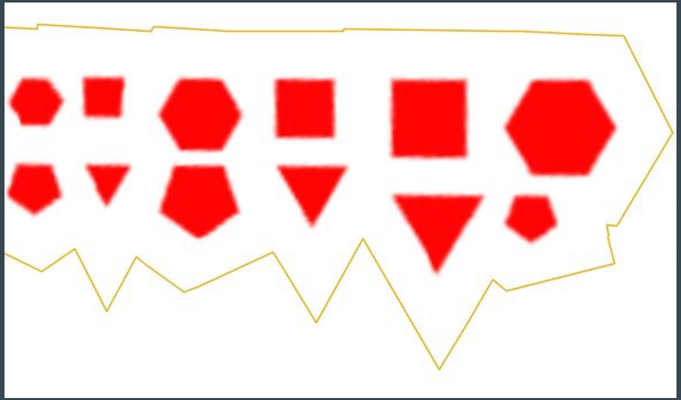
Design decision

Collision detection & avoidance = obstacles offset + (prm + *collisionDetectionModule.cpp*).

How & Where

- Determine the length of the radius that circumscribes the robot.
- Obstacles augmentation equals to the radius + 2cm of safe margin. (i.e. better be safe than sorry)

Obstacles offset is performed in *mapProcessing.cpp* via clipper library.



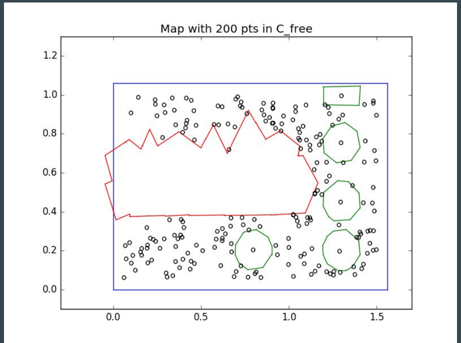
Motion Planning

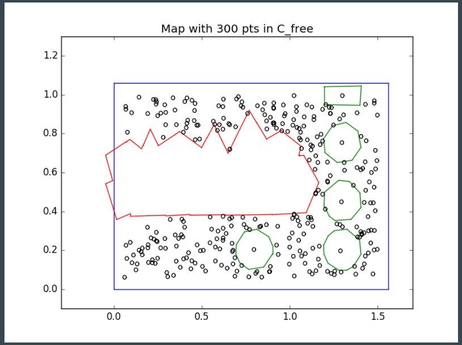
Design decision

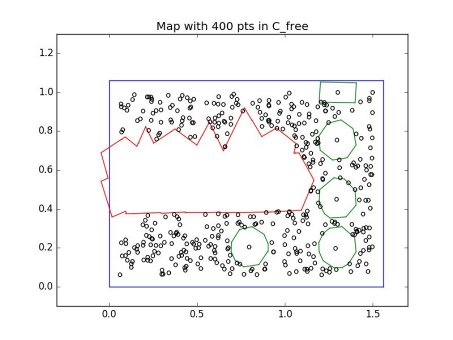
Points are sampled from the arena free space in order to create a graph (i.e. PRM).

How

- Points sampled from a uniform distribution.







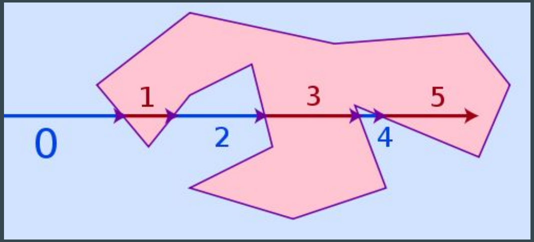
Motion Planning

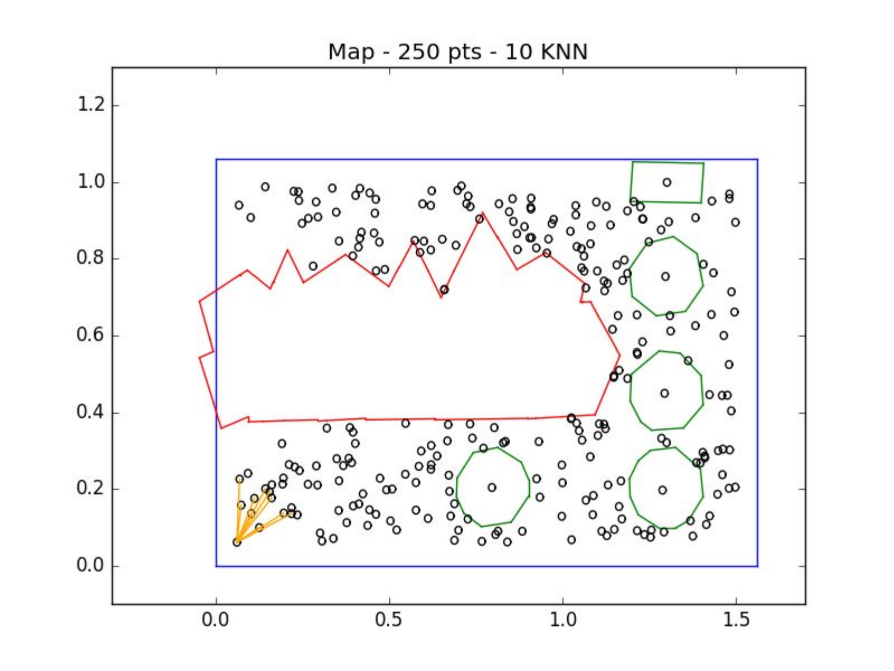
Design decision

Points are sampled from the arena free space in order to create a graph (i.e. PRM).

How

- Points sampled from a uniform distribution.
- Each point/node has K edges (i.e. KNN).
- Edges are valid iff they do not intersect with any obstacle (i.e. ray-casting algorithm).





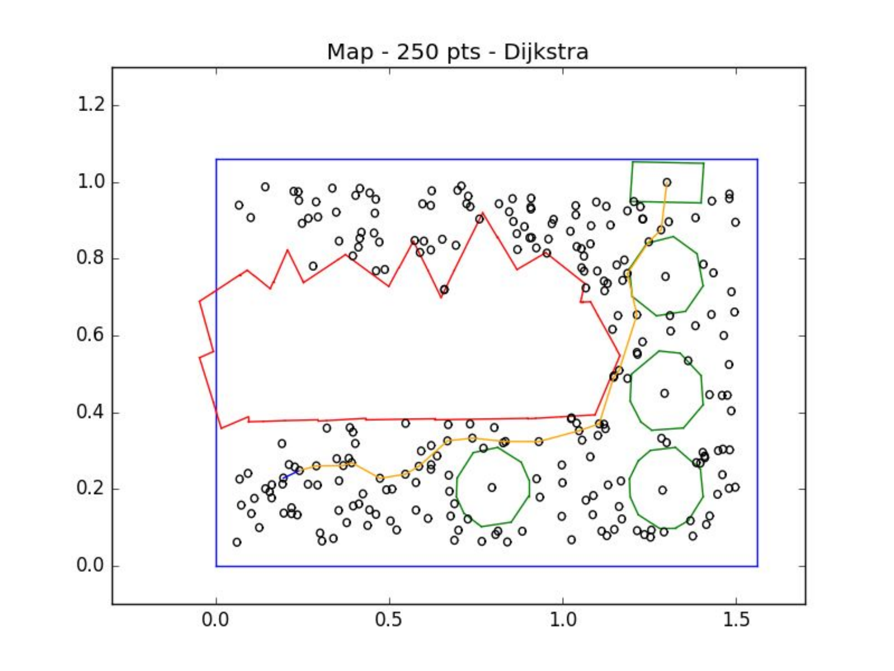
Motion Planning

Design decision

Points are sampled from the arena free space in order to create a graph (i.e. PRM).

How

- Points sampled from a uniform distribution.
- Each point/node has K edges (i.e. KNN).
- Edges are valid iff they do not intersect with any obstacle (i.e. ray-casting algorithm).
- Get shortest path via Dijkstra discrete search.



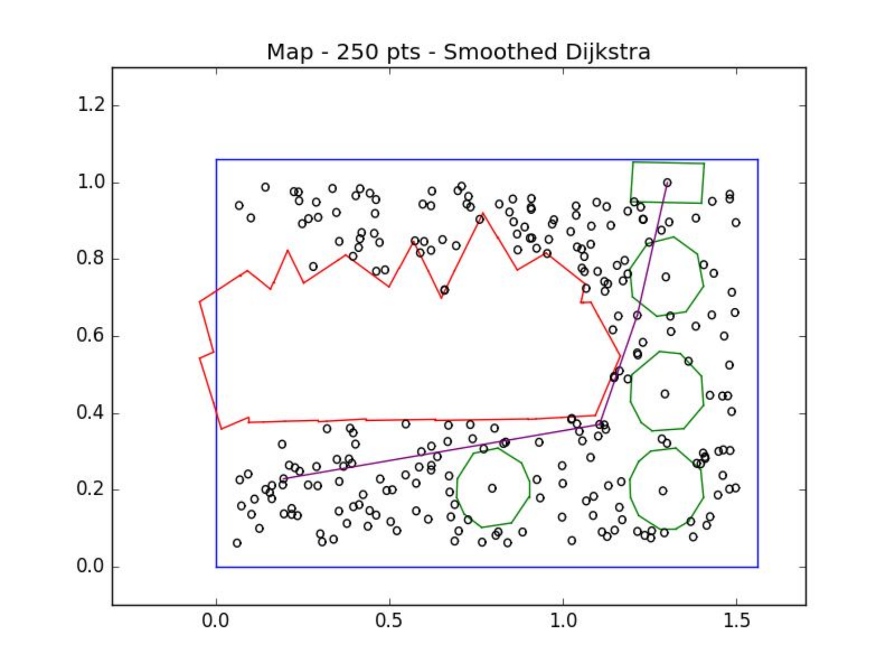
Motion Planning

Design decision

Points are sampled from the arena free space in order to create a graph (i.e. PRM).

How

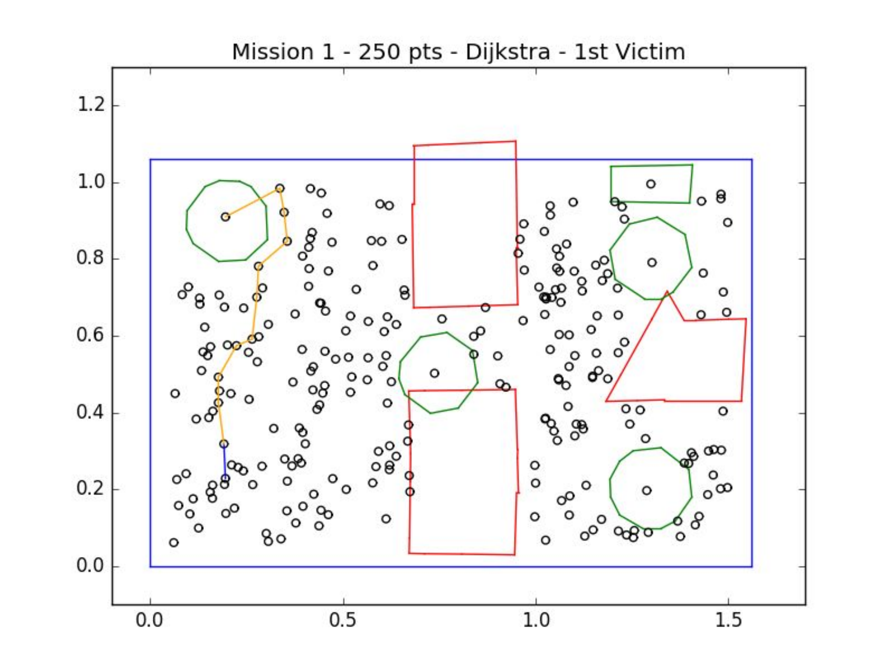
- Points sampled from a uniform distribution.
- Each point/node has K edges (i.e. KNN).
- Edges are valid iff they do not intersect with any obstacle (i.e. ray-casting algorithm).
- Get shortest path via Dijkstra discrete search.
- The smoother tries to shortcut all pts between the beginning and the end one by one.



Mission 1

How it works

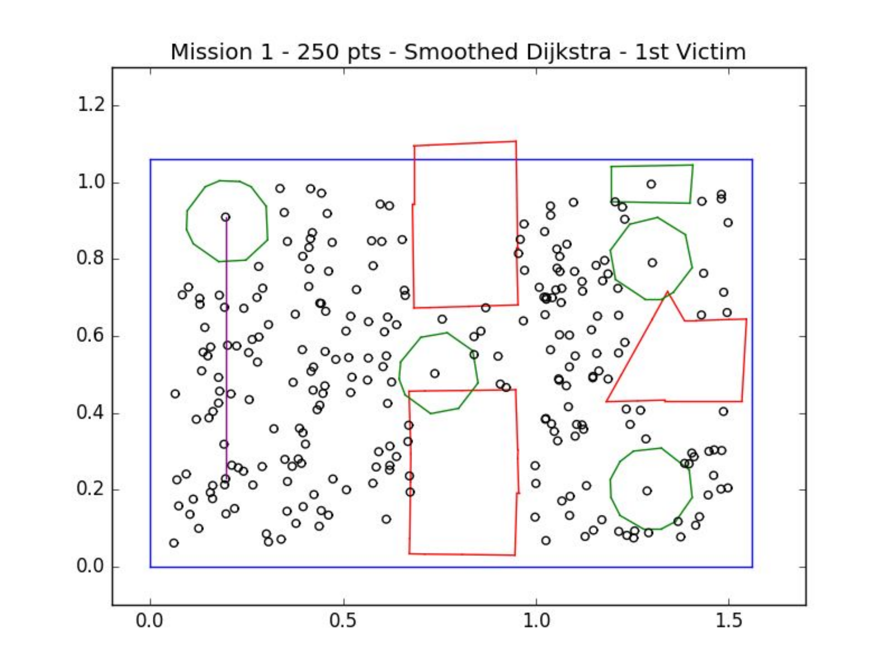
1. Sort the victims by ID;
2. Create a list of known path’s points;
3. For each two consecutive nodes of the list:
    a. Compute the Dijkstra shortest path;



Mission 1

How it works

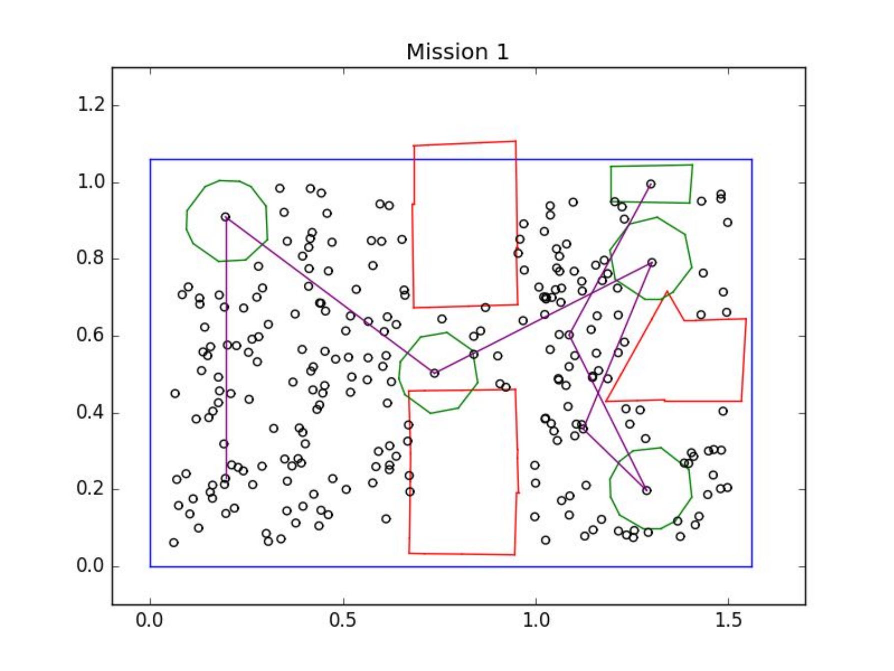
1. Sort the victims by ID;
2. Create a list of known path's points;
3. For each two consecutive nodes of the list:
   a. Compute the Dijkstra shortest path;
   b. Compute the smoothed path;
   c. Add the smoothed points to a new path list;



Mission 1

How it works

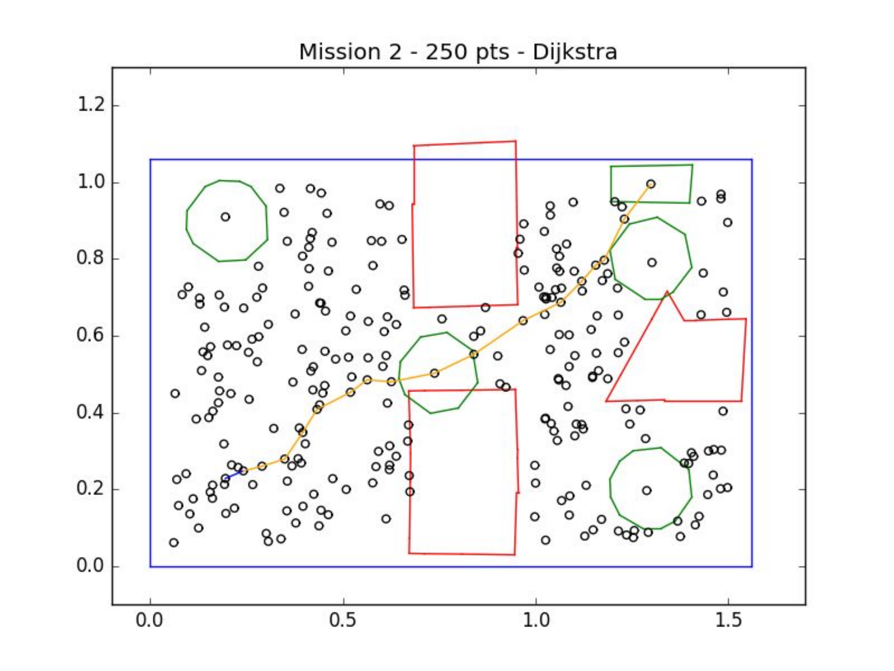
1. Sort the victims by ID;
2. Create a list of known path's points;
3. For each two consecutive nodes of the list:
    a. Compute the Dijkstra shortest path;
    b. Compute the smoothed path;
    c. Add the smoothed points to a new path list;
4. Use the points in the new list to solve the Dubins problem.



Mission 2

How it works

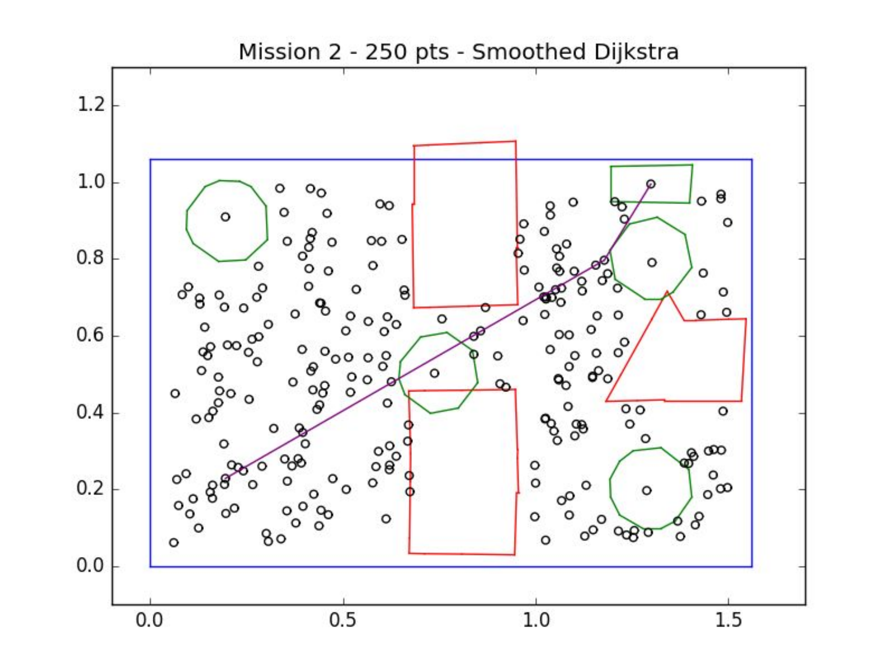
1. Compute the smoothed Dijkstra distance for the robot-gate path;



Mission 2

How it works

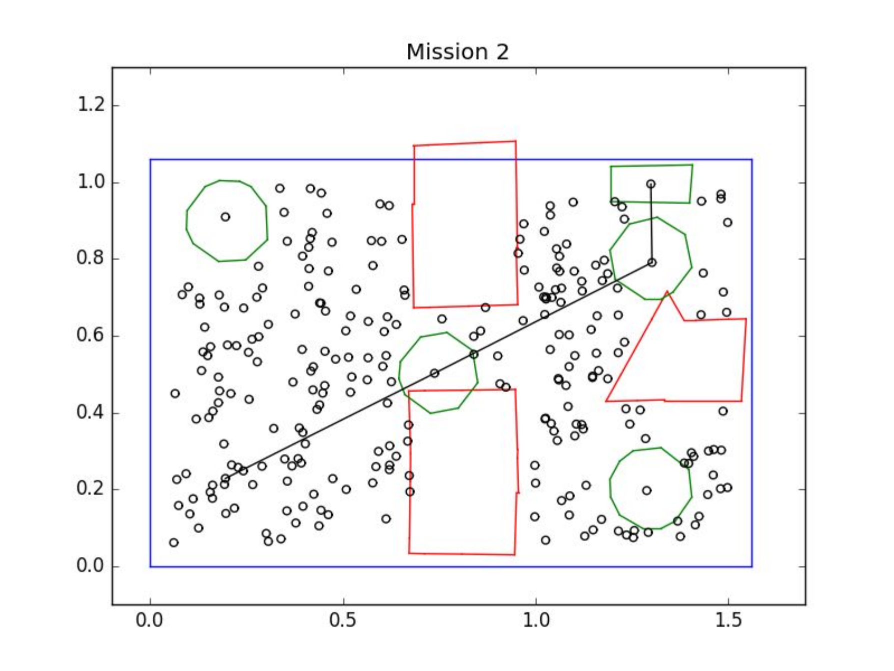
1. Compute the smoothed Dijkstra distance for the robot-gate path;
2. Set the max path length as 1. + 30%;



Mission 2

How it works

1. Compute the smoothed Dijkstra distance for the robot-gate path;
2. Set the max path length as 1. + 30%;
3. Create a list of known path's points;
4. Compute a matrix of Dijkstra distances for all the points of the list;
5. Get the nearest pt w.r.t. the starting pt;
6. A = len(starting pt - nearest pt path);
7. B = leng(nearest pt - gate path);
8. If A+B <= 2.
    a. Add the nearest pt to a new list;
    b. starting pt = nearest pt;
    c. Go to 5.
9. Use the points in the new list to solve the Dubins problem.



Thank You for the attention