Artificial Intelligence - Methods and Applications (5DV122)

Assignment 2

Map maker

Submitted by

Username	Group Members
ens15zni	Zheyi Ni
mrc15ymi	Yongcui Mi

Under the guidance of

Ola Ringdahl

Autumn Semester
2015

Contents

Chapter 1

Introduction

Here, we report our robot “Curiosity” is able to explore autonomously and efficiently in large-scale environment, as well as mapping accurately. “Curiosity” possesses a hybrid controller, including frontier based destination decider module, revised A* algorithm based path planner module, Bayes’s rule based cartographer module, “follow the carrot” based path follower module and vector field histogram based obstacle dodger module, who has tremendous application in space, ocean or radioactive environment exploration.

Our program has eleven files, where mapMake.py, getDst.py, pathPlan.py, pathFollow.py and vfh.py correspond to the cartographer module, destination decider module, path planner module, path follower module and obstacle dodger module respectively; MRDS.py interacts with Microsoft Robotics Developer Studio 4 (MRDS); basic.py includes some common functions; input.py reads arguments; main.py combines aforementioned files together; visualize.m visualizes maps in the Matlab and mapper supply an interface that our program can be called from Linux.

The program takes five arguments: url, x_1, y_1, x_2 and y_2. url specifies the address and port used to connect to MRDS. url is truncated to remove “http://”. x1, y1, x2 and y2 give the coordinates of the lower left and upper right corners of the area need to be explored. We assume that the area always includes the starting position (0, 0).

The program will interact with the robot by making http connections to the locally running MRDS server. A Robosoft Kompai robot is used in the simulation environment: Factory.

The achieve funtion save the current map as an image file every 5 seconds. matplotlib.pyplot is employed to save the image. Note that Putty doesn’t support the graphical output, hence a non-interactive backend directive matplotlib.use() is called before importing pyplot to prevent the window appearance of the generated images. Meanwhile, the program will also save the current map, robot position, path as text file for further analysis.

The metric map is a discrete, two-dimensional matrix. Each grid cell in a map is assigned a value that measures the subjective belief of the corresponding region of the world is occupied, coined mapProb in the program accordingly. Therefore the value is close to 1, if the grid cell is occupied, and 0 otherwise. The initial value of all grid cells is 0.5. Occupancy values are determined based on sensor readings. The resolution of a grid must be fine enough to capture important detail of the world and not too fine to prevent the complexity of time and space. The resolution of the grid is set to 0.5 in our program, larger than the length of the robot. Thus the metric map contains the belief as to whether or not the robot can be moved to that grid cell and represents the configuration space of the robot projected into the x-y plane.

“Curiosity” is equipped with an array of laser scanners which measures proximity of

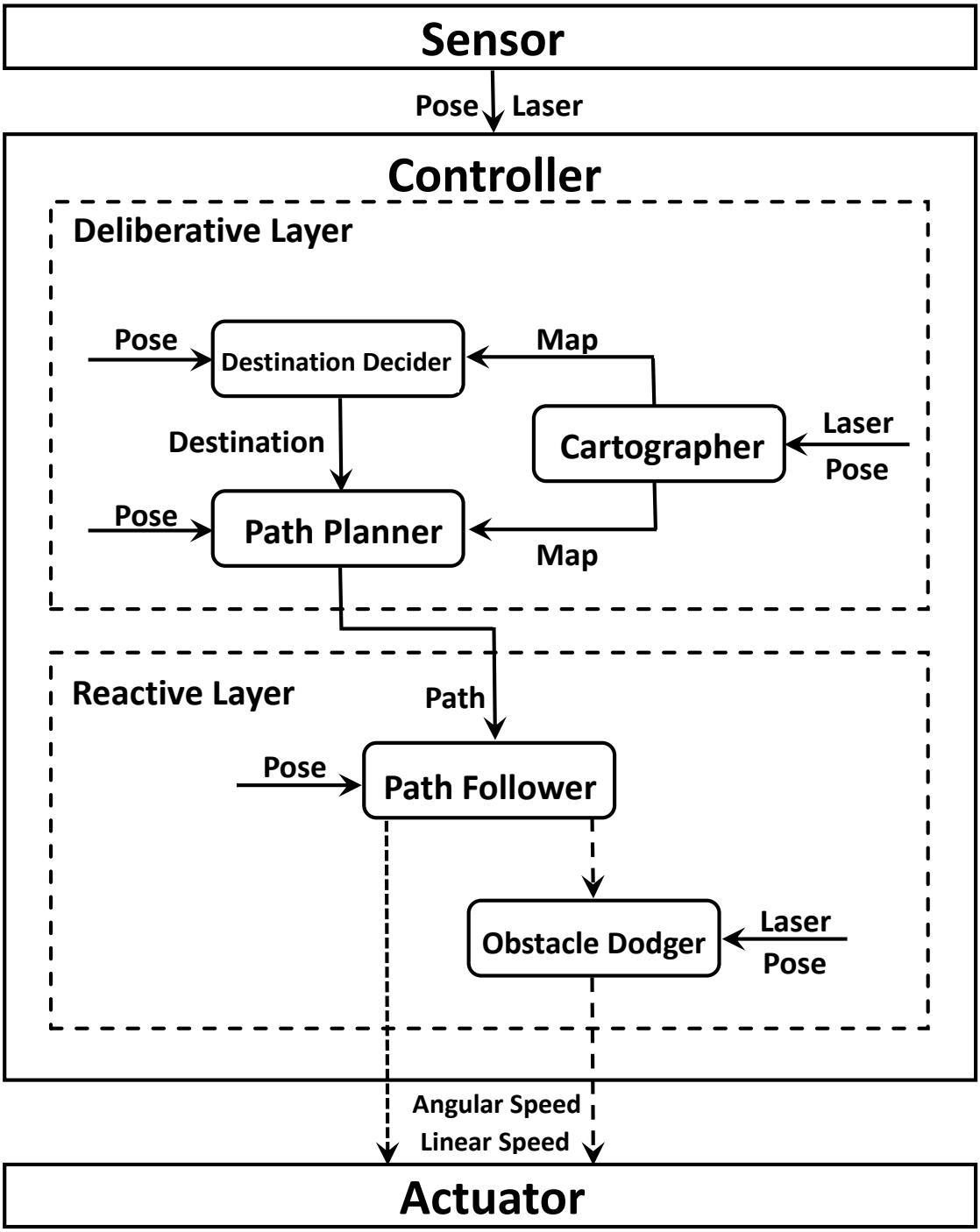

Figure 1. The architecture of "Curiosity"

nearby objects with high spatial resolution. The array contains 271 laser scanners evenly distributed from -135 degree to 135 degree with an interval of 1 degree. The maximum detection distance is 40 meters. If an obstacle is more than 40 meters away from the robot, the property 'echoes' gotten from laser is 40 and the property 'overflow' is 1. Besides, "Curiosity" is also equipped with a position sensor which provides location information of robot. Note that the heading of robot is represented in quaternion (w, x, y, z). Following formula (1) is used to convert the quaternion to the Euler angle.

$$\varphi = \tan^{-1}\frac{2(w*z+x*y)}{1-2(y^2+z^2)}$$

The laser scanner cannot see through walls. If the robot is going to build a complete map it has to explore the environment, using its sensors and moving to different locations, until all areas are covered [2]. The flowchart of "Curiosity" is illustrated in figure 1.

The sensor will export pose and laser data to the controller.

The controller is a hybrid of deliberative layer and reactive layer. The deliberative layer, which represents and maintains the knowledge of the world, involves cartographer module, path planner module and destination decider module. The cartographer module reads laser and pose and deliver map to destination decider module and path planner module. The destination decider takes pose and map as input and output the coordinate of the destination. Then destination, pose and map are imported to the path planner module and generate a list of coordinates of path.

The reactive layer, which has more concrete responses to the environment and less knowledge representation and planning, involves the path follower module and obstacle dodger module.

When there is no obstacles nearby, which is checked by the isVfh function, the path follower module takes pose and the path transferred from path planner module as input and sets the linear speed and angular speed. If the distance between robot and obstacle is less than 2 meters, then the goal direction, pose and laser will be sent to the obstacle dodger module and let the obstacle dodger algorithm determine a new goal direction [3].

In the current version of our program, obstacle dodger module is disabled. For the reason that in the most cases, "Curiosity" strikes the wall because of the unsatisfactory planned path and inaccurate sensor data. Even though the robot avoids the obstacle, path follow module will misguide the robot to the obstacle again. Hence, when an obstacle is near to the "Curiosity", it will turn back, stop and get a new destination.

A list of unknown grid cells that is going to be detected, called roster, is derived through getRoster function. Destination decider won't decide next destination until "Curiosity" has reached the destination or most of the roster is detected, which is checked by replan function. Considered the time that destination decider module and path planner module require, "Curiosity" will stop and wait for the following command.

Destination decider will return a list of destination sorted base on the ascending order of the cost function value. For the complexity of the topography, sometimes path planner fails to search a path to the destination. Then next destination will be popped out until the path is found.

This whole process won't stop until isEnd function gives a positive signal, who monitors the progress of the exploration and the number of frontiers.

The actuator executes movement according to linear speed and angular speed.

Exploration strategies has two general approaches: reactive and model based. The most widely used exploration strategy in reactive robotics is wall following. However, wall following is a local method and easily get trapped in loops. Model based strategies on basis of the same underlying idea: go to the least-explored region [2].

We take frontier based exploration, a kind of model based strategy. Frontiers are defined as the grid cell at the boundary of empty and unknown areas, which is an interesting place for exploration. Due to the great number of frontiers, nearly 4000 grid cells satisfy the criterion of frontier at beginning, a filter is applied to reduce the number of frontiers by a factor of 20. This reduction is reasonable since there is no need to keep massive adjacent grid cells for further calculation and the remaining frontiers could be seen as the representations of local unknown areas. Frontiers are scored depend on the cost function, formulated in equation 2, where k_d, k_s and k_a are the significance coefficients. d_i, s_i and a_i are normalized.

$$v_i = k_d * d_i + k_s * s_i + k_a * a_i \quad (2)$$

The cost function could be divided into three terms:

The first term is the cost for going to frontier i, where d_i is the distance between the robot and frontier i.

The second term is the cost for the repeating detection, where s_i is the negative number of unknown grid cells the robot can detect when it is at frontier i. Here we make three simplification. The first one is that we presume the heading equals to the orientation of line from the robot to the frontier i. Since that s_i is dependent on the heading of the robot, which will reach its minimum when the heading perpendicular to tangent line of frontier i. The second one is that we presume the laser scanner can see through walls here, the unknown grid behind an obstacle is also taken into account. The third one is that we presume a subset of the coverage of laser scanner array could reflect the real number of unknown grid cells. Since that computation of this term is the most time-consuming module in our program. We reduce 20000 covered grid cells by a factor of 20.

The third term is the cost for turning to frontier i, where a_i is the difference between the robot heading and the orientation of line from the robot to the frontier i. We expect the robot will prefer front frontiers more than back frontiers.

Revised A* algorithm is applied to the path planner module. A* algorithm is an informed search algorithm, which selects the node to expand next by using an evaluation function $f(n)$ shown in equation 3, where $g(n)$ is the cost of getting to one node and $h(n)$ is the heuristic evaluation of the node. Here, the heuristic is the straight line distance between the grid cell and the goal. A* algorithm will find the shortest path to the goal. However, the shortest path is not the optimal path. We hope that the path stay away from obstacles. Consequently, a topography term is taken into account. The evaluation function of the revised version of A* algorithm is the combination of the $g(n)$, $h(n)$ and the topography.

$$f(n) = g(n) + h(n) \quad (3)$$

topoEval function will evaluate the topography neighbor to one grid cell in terms of the parameter l. For example, if l equals to 2, then the value of 5x5 matrix center on that grid cell will be summed and normalized. As the value of one grid cell reporting the probability that the corresponding region is occupied, we try to minimize cost of topography, as well as the cost of $f(n)$. Nevertheless, topoEval function seems to has no effect: planned path tend to close to rather than away from the obstacles.

Thus the blur function is called to keep a safe distance to obstacles and minimize the

risk of collisions. The blur function will enlarge the size of obstacles in terms of the parameter l and return a blurred map. For example, if l equals to 2, then the 5x5 matrix center on occupied grid cell all will be regarded as occupied. A small l is useless while a large l will block proper path that can be gone through. l equals to 5 in our program. The blurred map will be imported to path planner module to obtain a safer path.

Path follower module takes "follow the carrot" algorithm, a straight forward algorithm for path tracking. The principle of the "follow the carrot" algorithm is that the robot steers towards a "carrot" at each time step. We keep the linear speed a constant and change the angular speed according to the "carrot" and the robot position. The angular speed ω is determined by equation 4, where k is the gain factor and orientation error ε is the angle between the robot heading and the line from the robot to the target. Schematic of the 'follow the carrot' algorithm is shown in figure 2. It should be mentioned that ε need to be converted to the range from $-\pi$ to π.

$$\omega = k\varepsilon \tag{4}$$

Carrot point is decided based on the look ahead distance. The robot will tend to oscillate around the path if the look ahead distance is too small, while ignore the situation around if the look ahead distance is too large. If the distance between the robot and the carrot point is below a certain threshold, the nextPoint function will find another carrot point ahead of the look ahead distance on the path to renew the current one. Linear speed is set to 3.0, look ahead distance is set to 1.6 and k is set to 1.1 in our program.

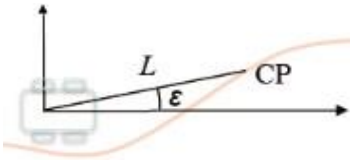

Figure 2. Schematic of the 'follow the carrot' algorithm

Obstacles dodger module takes vector field histogram (VFH) method. The first step of VFH is creating a one-dimensional polar histogram around the robot, resulting in a number of sectors with obstacles. Obstacles are allocated to sectors according to the angular resolution α and the value of one sector is the distance to the nearest obstacle in that sector. A sector is thought to be empty if whose value larger than a certain threshold.

The next step is selecting the steering angle. Candidate valley could be single empty sector or consecutive empty sectors, representing a free space where the robot may pass. The goal valley is chosen with the lowest value of the cost function formulated in equation 5, where k_s, k_a and k_t are the significance coefficients. s_i, a_i and t_i are normalized.

$$g_i = k_s * s_i + k_a * a_i + k_t * t_i \tag{5}$$

The cost function could be divided into three terms:

The first term is the cost for the risk going at the candidate direction, where s_i is the negative number of empty sectors in the candidate valley.

The second term is the cost for turning to the candidate direction, where a_i is the difference between the candidate direction and the orientation of the robot.

The last term is the cost for turning to the target direction, where t_i is the difference between the candidate direction and the target direction [3].

Cartographer module takes charge of building sensor model, which connects sensing and perception. The sensor model mainly focus on two problems: interpretation and integration of the sensor readings. Interpretation means mapping sensor readings to occupancy values and integration means integrating multiple sensor interpretations over time to yield a single combined estimation of occupancy.

The role of mapUpdate function is mapping the sensor readings from continuous world to the metric map and updating mapFreq. mapFreq records the times a grid cell is detected as empty or occupied. If the property 'overflow' of one laser is 1, then all the grid cells the laser passing through are empty. Otherwise, except for the grid cell located at the end of the laser beam is occupied, other grid cells are empty. getLine function will return a list of grid cells the laser passing through when informed of the start point and end point of the laser beam.

The role of integrate function is interpreting and integrating the sensor readings, namely calculating mapProb according to mapFreq by the evidential method Bayes's rule. The sensor reading at time t is denoted as $s^{(T)}$ and the probability that a grid cell (x, y) is occupied conditioned on the sensor reading $s^{(T)}$ is denoted as $Prob\left(occ_{x,y}\middle|s^{(T)}\right)$. Therefore $Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T)}\right)$, which is condition on all sensor readings, can be computed as equation 6.

$$Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T)}\right)=1-\left(\frac{1}{1+\prod_{\tau=1}^{T}\frac{Prob\left(occ_{x,y}\middle|s^{(\tau)}\right)}{1-Prob\left(occ_{x,y}\middle|s^{(\tau)}\right)}}\right)$$

For the ignorance of the accuracy of the laser scanners, we suppose that the probability that a grid cell (x, y) is occupied conditioned on the sensor reading is occupied is set to 0.9 and the probability that a grid cell (x, y) is empty conditioned on the sensor reading is empty is also set to 0.9.

The derivation of equation 6 follows directly from Bayes's rule and the conditional independence assumption. According to Bayes's rule,

$$\frac{Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T)}\right)}$$

$$=\frac{Prob\left(s^{(T)}\middle|occ_{x,y},s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}{Prob\left(s^{(T)}\middle|\neg occ_{x,y},s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}\frac{Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}$$

which can be simplified by virtue of the conditional independence assumption to

$$=\frac{Prob\left(s^{(T)}\middle|occ_{x,y}\right)}{Prob\left(s^{(T)}\middle|\neg occ_{x,y}\right)}\frac{Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}$$

Applying Bayes's rule to the first term leads to

$$=\frac{Prob\left(occ_{x,y}\middle|s^{(T)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(T)}\right)}\frac{Prob\left(\neg occ_{x,y}\right)}{Prob\left(occ_{x,y}\right)}\frac{Prob\left(occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}{Prob\left(\neg occ_{x,y}\middle|s^{(1)},s^{(2)},\cdots,s^{(T-1)}\right)}$$

Induction over T yields:

$$= \frac{Prob(occ_{x,y})}{1 - Prob(occ_{x,y})} \prod_{\tau=1}^{T} \frac{Prob(occ_{x,y}|s^{(\tau)})}{1 - Prob(occ_{x,y}|s^{(\tau)})} \frac{1 - Prob(occ_{x,y})}{Prob(occ_{x,y})}$$

Here $Prob(occ_{x,y})$ denotes the prior probability for occupancy, which, if set to 0.5, can be omitted in the equation.

Chapter 2

Discussion

Although grid-based method produce accurate metric maps, its complexity often prohibits efficient planning in large-scale environments. The getDst function and pathPlan function take most of the time in our program.

Due to the lack of the display of planning path in the Microsoft Visual Simulation Environment (MVSE), we are inaccessible to the flaws in our program. Therefore, we invoke achieve function every several seconds to append current map, path and robot position to the corresponding list. The lists will be saved as text files after simulation and pre-processed before loaded into Matlab. Then, getframe function is called to get frames from the superimposition of time lapse map, path and robot position. Movie function is called to play recorded movie frames. Movie 1 in the supplementary material shows the perfect path following performance under the sampling interval is 1 second. Movie 2 in the supplementary material shows rational destination decision and path planning, as well as perfect path following under the sampling interval is 5 seconds. We notice that the robot start to oscillate around the path in the later stage of exploration and this phenomenon disappears when the achieve function is not called. Thus we guess that the large volume of saving files damper the speed of computation.

Through video we find some interesting details:

One is that the boundaries of obstacles is highly variable. Specifically, previous occupied grid cells will become empty and their empty neighbors will become occupied, or empty grid cells near an occupied grid cell will become occupied. Except for the noise of the sensors, discretization is also a contributing factor for this variability. Even small deviation in the continuous world will cause appreciable difference in the discrete grid map. For example, -49.501 is divided into grid 0 and -49.499 is divided into grid 1 in our case.

Another interesting detail is that when the robot collide with an obstacle, the laser scanners will be influenced and generate “ghost grids” on the map. The ghost grid is the grid cell who is empty while considering it is occupied on the map. The ghost grid could be eliminated through repeating detection.

Figure 3 shows the map of assigned square in the factory environment constructed by robot “Curiosity” under the input $x_1 = -50,\ y_1 = -50,\ x_2 = 50$ and $y_2 = 50$. Figure 4 shows the visual and physics rendering of the factory environment screenshotted from the MVSE. Although two figures are extremely identical, still some problems could be found in our map. Problems, labeled in figure 3, could be classified into three categories, which indicate the defect of our sensor model.

The first problem is the absent street lights. We suspect that the small sectional area of street light can’t guarantee it can be detected by laser scanners every time. To check whether “Curiosity” detect the street lights, we visualize the frequency of every grid cell detected as occupied and empty. As shown in figure 5, the upper left and right panels are the frequency map of occupied or empty signal detected respectively. The value of each

Figure 3. The map of assigned square in the factory environment constructed by robot "Curiosity" under the input $x_1 = -50$, $y_1 = -50$, $x_2 = 50$ and $y_2 = 50$. The circled number labels where the problem is.

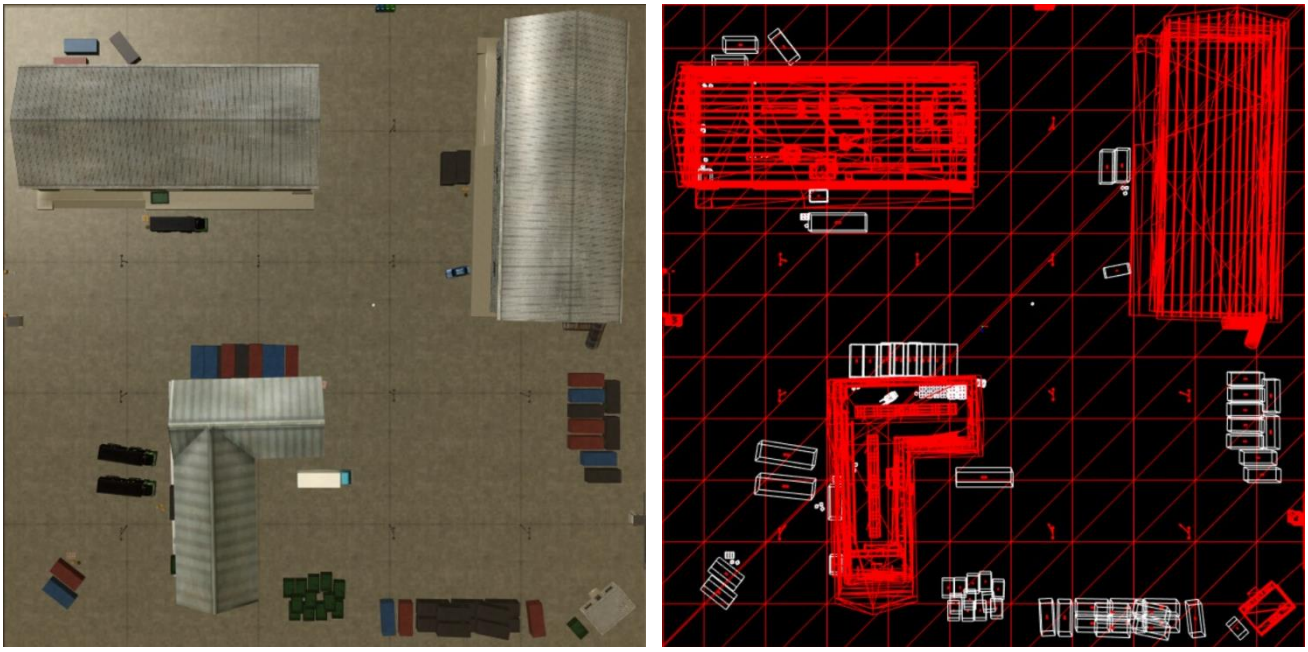

Figure 4. The rendering of the factory environment
(left panel) The visual rendering of the factory environment. (right panel) The physics rendering of the factory environment.

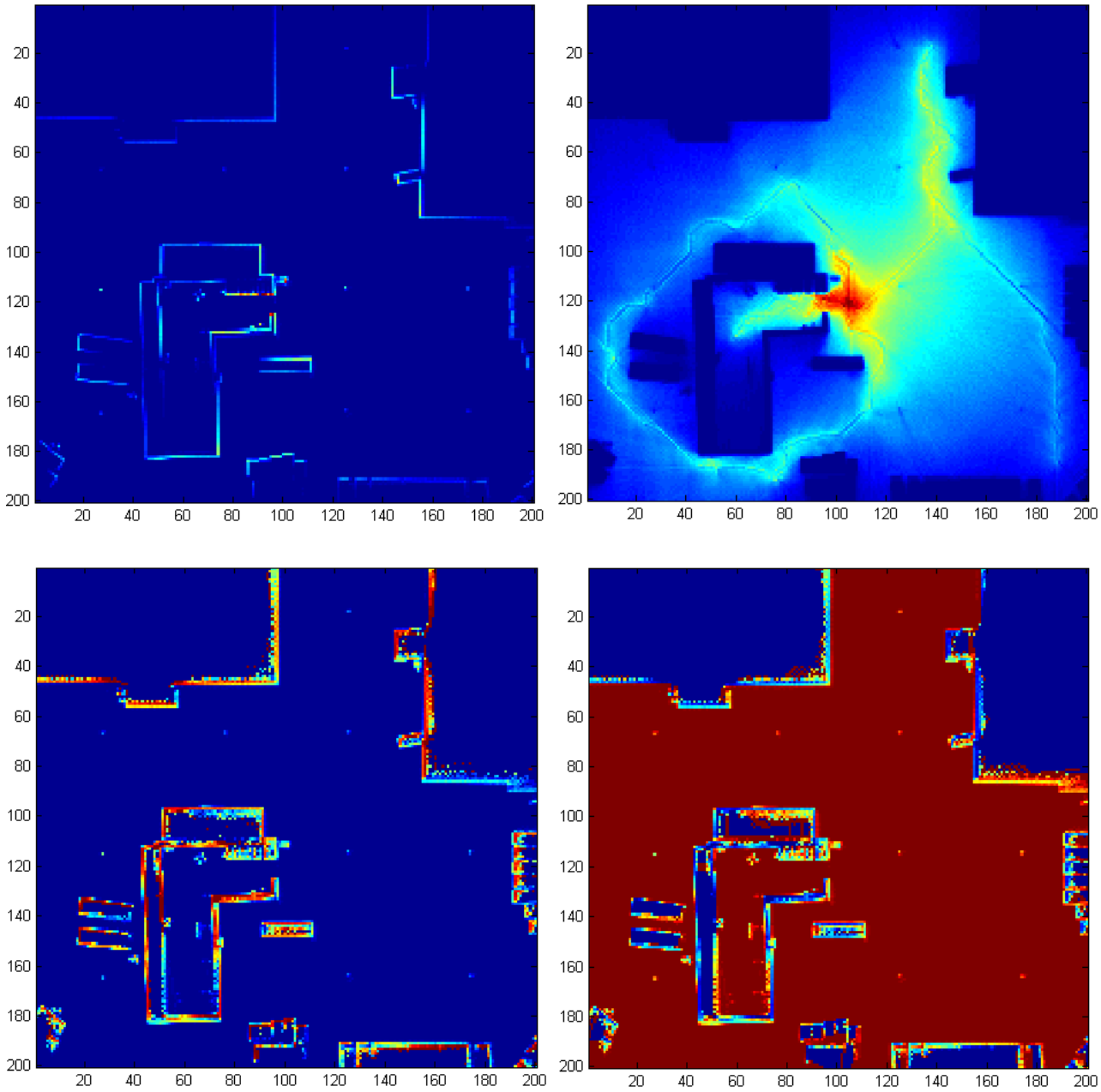

Figure 5. The frequency map. (upper left panel) The frequency map of occupied signal detected. (upper right panel) The frequency map of empty signal detected. (lower left panel) The normalized frequency map of occupied signal detected. (lower right panel) The normalized frequency map of empty signal detected.

grid cell is the times this grid cell detected as occupied or empty. The lower left and right panels are the normalized frequency map of occupied or empty signal detected respectively. The value of each grid cell is the normalized frequency this grid cell detected as occupied or empty. Apart from the frequency map of empty signal detected, the other three frequency maps all sketch the outline of obstacles, especially the missing street lights. Therefore, the grid cells where street lights locate are detected more times as empty, which covers the times detected as occupied.

The second problem is the inscrutable wall passing, which is also observed in the video.

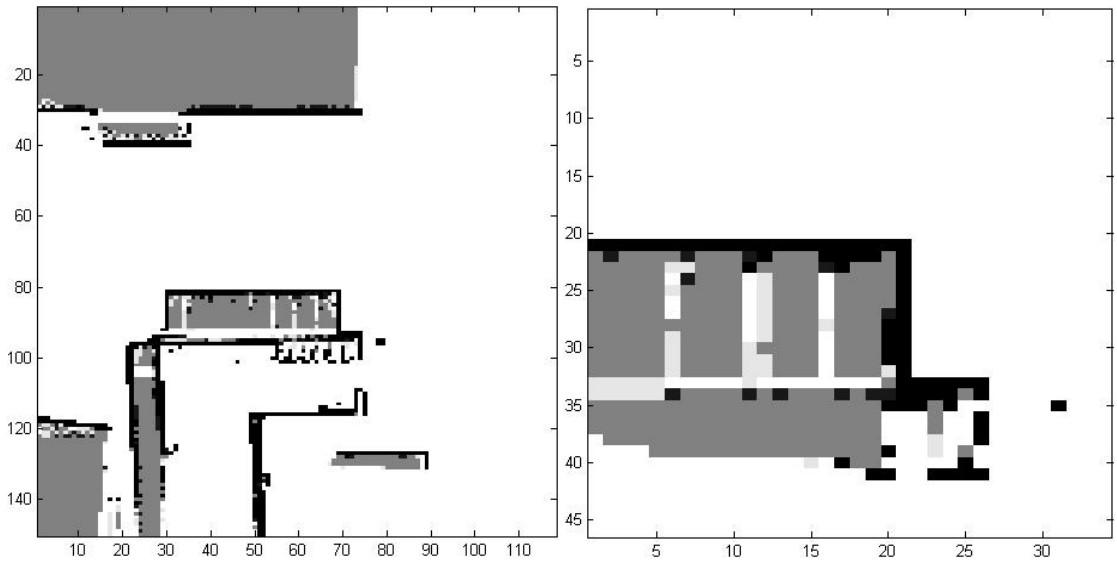

Figure 6. The maps constructed by “Curiosity” under different inputs. (left panel) The map constructed under the input $x_1 = -42,\ y_1 = -39,\ x_2 = 17$ and $y_2 = 36$. (right panel) The map constructed under the input $x_1 = -12,\ y_1 = -15,\ x_2 = 5$ and $y_2 = 8$.

This may ascribed to the noise of sensors and the complexity of topography. The only one wall passing example we figure out happens at the entrance of the plant located in the center of the map. There is a threshold at the entrance and a slope in front of the entrance. Robot won’t detect anything behind the slope until it drives across the threshold.

The third problem is the missing boundaries. There should be a wall in the place where the label is. Unlike the absent street lights in the first problem, the wall is easily detected. This problem is relevant to the aforementioned variability of the boundaries of obstacles. Indeed, we can find these missing boundaries in four frequency maps.

In conclusion, the above three problems instruct us to reduce the probability that a grid cell is empty conditioned on when sensor reading is empty.

Figure 5 also reveals that the grid cells in the center of the map are more detected than the peripheral grid cells. Besides, there is a hot spot near the entrance of the plant located in the center of the map because of the high centrality of that node in the geographical network.

Next, we test whether the controller can guide “Curiosity” around any given square area. As shown in figure 6, “Curiosity” constructs two maps under different inputs. Both maps are similar to the visual and physics rendering of the factory environment shown in figure 3. Movie 3 in the supplementary material shows the process “Curiosity” explores the map under the input $x_1 = -12,\ y_1 = -15,\ x_2 = 5$ and $y_2 = 8$.

We try to create some index to evaluate the performance of exploration. The first index is the time spent on exploration. The average exploration time of our program is about 150 seconds with the linear speed 3.0.

Another index is the efficiency of exploration, which is define as the ratio between the path length taken by the omniscience and the path length taken by the robot. An intelligent agent would try to shorten the exploration path and avoid passing explored

region. Due to the dense equipment of the laser scanners, the grid cells close to the robot are more detected. Therefore, we can discern the trajectory of the robot from the frequency map of empty signal detected or the general frequency map. The trajectory implicates that our robot takes a relatively efficient strategy.

Other exploration methods, such as the value iteration algorithm, also have excellent performance. The basic idea of value iteration algorithm is that the algorithm updates the value of all explored grid cells by the value of their best neighbors, plus the costs of moving to this neighbor. Each value measures cumulative cost for moving to the nearest unexplored cell. Once value iteration converges, greedy exploration simply amounts to steepest descent in the value function [1].

From figure 3 we could see that not all unknown grid cells could be detected. Take the plant located in the upper left corner of the map as an instance, there is no entrance to that plant. Now, the program will end at the 77% of completion, actually it is the maximum the robot can detect. Hence an new function is required to help isEnd function judging the completion of the exploration: If a terrain is surrounded by obstacles, it would be regarded as obstacle either.

Reference

[1] Thrun S. (1998). Learning maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21-71.

[2] Romero L. *et al*.(2001). An exploration and navigation approach for indoor mobile robots considering sensor's perceptual limitations. *Proceedings - IEEE International Conference on Robotics and Automation* 2001(3):3092-3097.

[3] Ringdahl O. (2003). Path tracking and obstacle avoidance algorithms for autonomous forest machines.

Appendix

1. main.py

```
from time import sleep
from math import atan2
from threading import Thread
import matplotlib
matplotlib.use('Agg')
from matplotlib import pyplot

from basic import size, save, distance
from input import row, col, gridizeX, gridizeY
from pathPlan import pathPlan
from pathFollow import pathFollow, nextPoint
from getDst import getDst, getRoster, replan
from mapMake import mapUpdate, integration, gro
from MRDS import getPose, getLaser, postSpeed
from vfh import vfh


def archive():
    global mapProb
    sampleInterval = 5
    while not end:
        plt.imshow(mapProb)
        plt.savefig('map.jpg')
        sleep(sampleInterval)

def isVfh(front):
    for i in range(len(front)):
        if front[i] < 1.0:
            return True
    return False

def isEnd(mapFreq):
    global end
    row, col = size(mapFreq)
    detected = 0.0
    end = False
    for i in range(row):
        for j in range(col):
```

```
            if mapFreq[i][j][2] > 0:
                detected = detected + 1
    progress = detected / (row * col)
    if progress > 0.75:
        end = True
    return end

def main():
    global mapFreq, mapProb, count, path, gridRobotX, gridRobotY, end
    pose = getPose()
    x = pose['Pose']['Position']['X']
    y = pose['Pose']['Position']['Y']
    gridRobotX = gridizeX(x)
    gridRobotY = gridizeY(y)

    laser = getLaser()
    mapFreq = mapUpdate(pose, laser, mapFreq)
    blurMapFreq = gro(mapFreq)
    mapProb = integration(mapFreq)
    candidate = getDst(pose, mapFreq)
    while len(path) == 0:
        dst = candidate.pop()
        roster = getRoster(pose, dst, mapFreq)
        path = pathPlan([gridRobotX, gridRobotY], dst, blurMapFreq)
        target = nextPoint(x, y, path, 0)

    while True:
        pose = getPose()
        x = pose['Pose']['Position']['X']
        y = pose['Pose']['Position']['Y']
        gridRobotX = gridizeX(x)
        gridRobotY = gridizeY(y)
        laser = getLaser()
        front = laser['Echoes'][130 : 140]

        end = isEnd(mapFreq)
        if end:
            break;
        if replan(roster, mapFreq) or distance(x, y, path[-1][0], path[-1][1]) < 1:
            path = []
            response = postSpeed(0, 0)

            for i in range(20):
                candidate = getDst(pose, mapFreq)
```

```
            while len(path) == 0:
                dst = candidate.pop()
                roster = getRoster(pose, dst, mapFreq)
                path = pathPlan([gridRobotX, gridRobotY], dst, blurMapFreq)
                target = nextPoint(x, y, path, 0)
            if len(path) != 0:
                break
        mapFreq = mapUpdate(pose, laser, mapFreq)
        blurMapFreq = gro(mapFreq)
        mapProb = integration(mapFreq)
        if isVfh(front):
            psi = atan2(path[target][1] - y, path[target][0] - x)
            # vfh(pose, laser, psi)
            response = postSpeed(0, 0)
            sleep(0.5)
            response = postSpeed(0, -1.0)
            sleep(3)
            path = []
            response = postSpeed(0, 0)
            for i in range(20):
                candidate = getDst(pose, mapFreq)
                while len(path) == 0:
                    dst = candidate.pop()
                    roster = getRoster(pose, dst, mapFreq)
                    path = pathPlan([gridRobotX, gridRobotY], dst, blurMapFreq)
                    target = nextPoint(x, y, path, 0)
                if len(path) != 0:
                    break
        else:
            target = pathFollow(path, pose, target)
        count += 1

# [numOccupied, numEmpty, total]
mapFreq = [[[0 for i in range(3)] for j in range(row)] for k in range(col)]
mapProb = [[0.5 for i in range(row)] for j in range(col)]
mapTimeLapse = []
robotPosTimeLapse = []
pathTimeLapse = []
pathLength = []
count = 1
pose = getPose()
x = pose['Pose']['Position']['X']
y = pose['Pose']['Position']['Y']
gridRobotX = gridizeX(x)
```

```
gridRobotY = gridizeY(y)
path = []
gridPath = []
end = False

threads = list()
t1 = Thread(target = archive)
threads.append(t1)

for t in threads:
    t.setDaemon(True)
    t.start()

try:
    main()
except:
    pass
```

2. input.py

```
from sys import argv
from math import floor, ceil

def getInput():
    if(len(argv) != 6):
        print("Invalid Number of Arguments")
        print("Format: url, x1, y1, x2, y2 give the coordinates of the lower left and
upper right corners of the area the robot should explore and map.")
        exit(1)

    MRDS_URL = argv[1]
    x1 = argv[2]
    y1 = argv[3]
    x2 = argv[4]
    y2 = argv[5]
    if(x1 >= x2) or (y1 >= y2):
        print("Invalid Meaning of Arguments")
        print("Format: url, x1, y1, x2, y2 give the coordinates of the lower left and
upper right corners of the area the robot should explore and map.")
        exit(1)
    return MRDS_URL, x1, y1, x2, y2

def floorGrid(c):
    global gridResolution
    cFloored = floor(c / gridResolution) * gridResolution
    return cFloored

def ceilGrid(c):
    global gridResolution
    cCeiled = ceil(c / gridResolution) * gridResolution
    return cCeiled

def gridizeX(x):
    global x0, gridResolution
    gridX = int((floorGrid(x) - x0) / gridResolution)
    return gridX

def gridizeY(y):
    global y0, gridResolution
    gridY = int((floorGrid(y) - y0) / gridResolution)
    return gridY
```

```
def metrizeX(x):
    global x0, gridResolution
    metricX = x * gridResolution + x0 + gridResolution / 2
    return metricX

def metrizeY(y):
    global y0, gridResolution
    metricY = y * gridResolution + y0 + gridResolution / 2
    return metricY

gridResolution = 0.5
MRDS_URL, lowerLeftX, lowerLeftY, upperRightX, upperRightY = getInput()
MRDS_URL = MRDS_URL.replace('http://', '')
x0 = floorGrid(float(lowerLeftX))
y0 = floorGrid(float(lowerLeftY))
xn = ceilGrid(float(upperRightX))
yn = ceilGrid(float(upperRightY))

row = int((yn - y0) / gridResolution)
col = int((xn - x0) / gridResolution)
```

3. MRDS.py

```
import httplib, json

from input import MRDS_URL

# MRDS_URL = 'localhost:50000'
HEADERS = {"Content-type": "application/json", "Accept": "text/json"}

class UnexpectedResponse(Exception): pass

def postSpeed(angularSpeed,linearSpeed):
    """Sends a speed command to the MRDS server"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    params =
json.dumps({'TargetAngularSpeed':angularSpeed,'TargetLinearSpeed':linearSpeed})
    mrds.request('POST','/Lokarria/differentialdrive',params,HEADERS)
    response = mrds.getresponse()
    status = response.status
    #response.close()
    if status == 204:
        return response
    else:
        raise UnexpectedResponse(response)

def getLaser():
    """Requests the current laser scan from the MRDS server and parses it into a dict"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    mrds.request('GET','/Lokarria/laser/echoes')
    response = mrds.getresponse()
    if (response.status == 200):
        laserData = response.read()
        response.close()
        return json.loads(laserData)
    else:
        return response

def getLaserAngles():
    """Requests the current laser properties from the MRDS server and parses it into a
dict"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    mrds.request('GET', '/Lokarria/laser/properties')
    response = mrds.getresponse()
    if (response.status == 200):
```

```
        laserData = response.read()
        response.close()
        properties = json.loads(laserData)
        beamCount = int((properties['EndAngle'] - properties['StartAngle']) /
properties['AngleIncrement'] + 1)
        a = properties['StartAngle'] + properties['AngleIncrement']
        angles = []
        for i in range(beamCount):
            angles.append(a)
            a += properties['AngleIncrement']
        return angles
    else:
        raise UnexpectedResponse(response)

def getPose():
    """Reads the current position and orientation from the MRDS"""
    mrds = httplib.HTTPConnection(MRDS_URL)
    mrds.request('GET', '/Lokarria/localization')
    response = mrds.getresponse()
    if (response.status == 200):
        poseData = response.read()
        response.close()
        return json.loads(poseData)
    else:
        return UnexpectedResponse(response)
```

4. mapMake.py

```
from math import sin, cos, floor
from copy import deepcopy

from basic import size, sign, quaternion2Euler
from input import gridizeX, gridizeY, row, col
from MRDS import getLaserAngles

laserAngles = getLaserAngles()

def mapUpdate(pose, laser, mapFreq):
    global laserAngles

    row, col = size(mapFreq)
    robotX = pose['Pose']['Position']['X']
    robotY = pose['Pose']['Position']['Y']
    gridRobotX = gridizeX(robotX)
    gridRobotY = gridizeY(robotY)
    robotOri = quaternion2Euler(pose)
    laserEchoes = laser['Echoes']
    laserOverflow = laser['Overflow']

    mapFreq[gridRobotX][gridRobotY][1] += 1
    for i in range(len(laserEchoes)):
        frontierX = robotX + laserEchoes[i] * cos(robotOri + laserAngles[i])
        gridFrontierX = gridizeX(frontierX)
        frontierY = robotY + laserEchoes[i] * sin(robotOri + laserAngles[i])
        gridFrontierY = gridizeY(frontierY)
        if (gridFrontierX < row) and (gridFrontierX >= 0) and (gridFrontierY < col) and
(gridFrontierY >= 0):
            if laserOverflow[i]:
                mapFreq[gridFrontierX][gridFrontierY][1] += 1
            else:
                mapFreq[gridFrontierX][gridFrontierY][0] += 1

        line = getLine([gridRobotX, gridRobotY], [gridFrontierX, gridFrontierY],
mapFreq)
        for i in range(len(line)):
            mapFreq[line[i][0]][line[i][1]][1] += 1

    for i in range(row):
        for j in range(col):
            mapFreq[i][j][2] = mapFreq[i][j][0] + mapFreq[i][j][1]
```

```
    return mapFreq

def getLine(startPoint, endPoint, mapFreq):
    startPointX = startPoint[0]
    startPointY = startPoint[1]
    endPointX = endPoint[0]
    endPointY = endPoint[1]
    row, col = size(mapFreq)
    line = []

    if abs(endPointX - startPointX) == 0 :
            numGridInside = int(abs(endPointY - startPointY) - 1)
            gridInsideX = startPointX
            gridInsideY =  startPointY
            for j in range(numGridInside):
                gridInsideY = gridInsideY + sign(endPointY - startPointY) * 1
                if (gridInsideY >= 0) and (gridInsideY < col):
                    line.append([gridInsideX, gridInsideY])
    else:
        k = float(endPointY - startPointY) / float(endPointX - startPointX)

        if abs(k) < 1:
            numGridInside = int(abs(endPointX - startPointX) - 1)
            gridInsideX = startPointX
            temp =  float(startPointY)
            for j in range(numGridInside):
                gridInsideX = gridInsideX + sign(endPointX - startPointX) * 1
                temp = temp + sign(endPointY - startPointY) * abs(k)
                gridInsideY = int(floor(temp))
                if (gridInsideY >= 0) and (gridInsideY < col) and (gridInsideX >= 0) and
(gridInsideX < row):
                    line.append([gridInsideX, gridInsideY])

        else:
            numGridInside = int(abs(endPointY - startPointY) - 1)
            temp = float(startPointX)
            gridInsideY = startPointY
            for j in range(numGridInside):
                temp = temp + sign(endPointX - startPointX) * abs(1 / k)
                gridInsideX = int(floor(temp))
                gridInsideY = gridInsideY + sign(endPointY - startPointY) * 1
                if (gridInsideY >= 0) and (gridInsideY < col) and (gridInsideX >= 0) and
(gridInsideX < row):
                    line.append([gridInsideX, gridInsideY])
```

```
    return line

def integration(mapFreq):
    row, col = size(mapFreq)
    mapProb = [[0.5 for i in range(col)] for j in range(row)]
    for i in range(row):
        for j in range(col):
            diff = mapFreq[i][j][0] - mapFreq[i][j][1]
            if abs(diff) > 7:
                diff = sign(mapFreq[i][j][0] - mapFreq[i][j][1]) * 7
            product = pow(9, diff)
            mapProb[i][j] = 1 - 1.0 / (1 + product)
    return mapProb

def gro(mapFreq):
    row, col = size(mapFreq)
    close = [[0 for i in range(col)] for j in range(row)]
    blurMapFreq = deepcopy(mapFreq)
    for i in range(row):
        for j in range(col):
            if mapFreq[i][j][0] - mapFreq[i][j][1] > 0 and close[i][j] == 0:
                neighbour = getNeighbour(mapFreq, i, j, 5)
                for k in range(len(neighbour)):
                    if mapFreq[neighbour[k][0]][neighbour[k][1]][0] == 0:
                        if close[neighbour[k][0]][neighbour[k][1]] == 0:
                            close[neighbour[k][0]][neighbour[k][1]] = 1
                            blurMapFreq[neighbour[k][0]][neighbour[k][1]][0] += 50
                    else:
                        blurMapFreq[neighbour[k][0]][neighbour[k][1]][0] += 50
    return blurMapFreq

def getNeighbour(map, i, j, a):
    row, col = size(map)
    neighbour = []
    for u in range(-a, a + 1):
        for v in range(-a, a + 1):
            if (i + u >= 0) and (i + u < row) and (j + v >= 0) and (j + v < col):
                neighbour.append([i + u, j + v])
        return neighbour
```

5. getDst.py

```
from math import sin, cos, atan2, exp, floor

from basic import size, sign, distance, filter, indexMaxLs, quaternion2Euler
from input import gridizeX, gridizeY, row, col
from MRDS import getLaserAngles
from mapMake import getLine

laserMaxRange = 40
laserAngles = getLaserAngles()

def getRoster(pose, dst, mapFreq):
    row, col = size(mapFreq)
    roster = []
    gridRobotX = gridizeX(pose['Pose']['Position']['X'])
    gridRobotY = gridizeY(pose['Pose']['Position']['Y'])
    frontierOri = atan2(dst[1] - gridRobotY, dst[0] - gridRobotX)
    cover = getCover(frontierOri, mapFreq)
    for i in range(len(cover)):
        x = dst[0] + cover[i][0]
        y = dst[1] + cover[i][1]
        if (x >= 0) and (x < row) and (y >= 0) and (y < col):
            if mapFreq[x][y][2] == 0:
                roster.append([x, y])
    return roster

def replan(roster, mapFreq):
    progress = 0.0
    for i in range(len(roster)):
        if mapFreq[roster[i][0]][roster[i][1]][2] > 0:
            progress += 1
    progress = progress / len(roster)
    if progress > 0.90:
        return True
    return False

def getCover(frontierOri, mapFreq):
    global laserAngles, laserMaxRange
    row, col = size(mapFreq)
    cover = []
    robotX = 0
    robotY = 0
```

```
    gridRobotX = gridizeX(robotX)
    gridRobotY = gridizeY(robotY)
    cover.append([gridRobotX - gridRobotX, gridRobotY - gridRobotY])

    for i in range(len(laserAngles)):
        gridFrontierX = gridizeX(robotX + laserMaxRange * cos(frontierOri +
laserAngles[i]))
        gridFrontierY = gridizeY(robotY + laserMaxRange * sin(frontierOri +
laserAngles[i]))
        if (gridFrontierX <= row) and (gridFrontierX >= 0) and (gridFrontierY <= col)
and (gridFrontierY >= 0):
            cover.append([gridFrontierX - gridRobotX, gridFrontierY - gridRobotY])
        line = getLine([gridRobotX, gridRobotY], [gridFrontierX, gridFrontierY],
mapFreq)
        for i in range(len(line)):
            cover.append([line[i][0] - gridRobotX, line[i][1] - gridRobotY])
    return cover

def isUndetectedAround(mapFreq, i, j, a):
    row, col = size(mapFreq)
    for u in range(-a, a + 1):
        for v in range(-a, a + 1):
            if (i + u >= 0) and (i + u < row) and (j + v >= 0) and (j + v < col):
                if mapFreq[i + u][j + v][2] == 0:
                    return True
    return False

def isObstacleNear(mapFreq, i, j, a):
    row, col = size(mapFreq)
    for u in range(-a, a + 1):
        for v in range(-a, a + 1):
            if (i + u >= 0) and (i + u < row) and (j + v >= 0) and (j + v < col):
                if mapFreq[i + u][j + v][0] > 0:
                    return True
    return False


def getFrontier(mapFreq):
    row, col = size(mapFreq)
    frontier = []
    for i in range(row):
        for j in range(col):
            if mapFreq[i][j][1] - mapFreq[i][j][0] > 0:
                if isUndetectedAround(mapFreq, i, j, 3):
```

```
                if not isObstacleNear(mapFreq, i, j, 4):
                    frontier.append([i, j])
    return frontier

def getDst(pose, mapFreq):
    row, col = size(mapFreq)

    gridRobotX = gridizeX(pose['Pose']['Position']['X'])
    gridRobotY = gridizeY(pose['Pose']['Position']['Y'])
    robotOri = quaternion2Euler(pose)

    frontier = getFrontier(mapFreq)
    frontier = filter(frontier, 20)
    d = [1 for i in range(len(frontier))]
    s = [0 for i in range(len(frontier))]
    a = [1 for i in range(len(frontier))]
    v = [1 for i in range(len(frontier))]

    for i in range(len(frontier)):
        dis = distance(gridRobotX, gridRobotY, frontier[i][0], frontier[i][1])
        d[i] = abs(dis - 30)

        frontierOri = atan2(frontier[i][1] - gridRobotY, frontier[i][0] - gridRobotX)
        a[i] = abs(frontierOri - robotOri)
        cover = getCover(frontierOri, mapFreq)
        cover = filter(cover, 10)
        for j in range(len(cover)):
            x = frontier[i][0] + cover[j][0]
            y = frontier[i][1] + cover[j][1]
            if (x >= 0) and (x < row) and (y >= 0) and (y < col):
                if mapFreq[x][y][2] == 0:
                    s[i] += 1

    maxD = max(d)
    maxA = max(a)
    maxS = max(s)
    for i in range(len(frontier)):
        d[i] = 1 - d[i] / maxD
        a[i] = 1 - a[i] / maxA
        s[i] = s[i] / maxS

    pa = 0.1
    pd = 0.5
    ps = 0.4
```

```
for i in range(len(v)):
    v[i] = pa * a[i] + pd * d[i] + ps * s[i]
if len(v) > 50:
    indexMax = indexMaxLs(v, 50)
else:
    indexMax = indexMaxLs(v, len(v))

dst = []
for i in range(len(indexMax)):
    dst.append(frontier[indexMax[i]])

dst.reverse()
return dst
```

6. pathPlan.py

```
from basic import size
from input import metrizeX, metrizeY

avoidD = 5

def topoEvaluate(x, y, map, a):
    row, col = size(map)
    local = 0
    for i in range(-a, a + 1):
        for j in range(-a, a + 1):
            if (x + i >= 0) and (x + i < row) and (y + j >= 0) and (y + j < col):
                if map[x + i][y + j][0] > 0:
                    local += 1
    s = local / pow(a, 2)
    return s


def pathPlan(init, goal, map):
    row, col = size(map)
    path = []

    cost = 1
    delta = [[-1,  0],  # go up
             [ 0, -1],  # go left
             [ 1,  0],  # go down
             [ 0,  1],  # go right
             [-1, -1],  # go upleft
             [-1,  1],  # go upright
             [ 1, -1],  # go downleft
             [ 1,  1]]  # go downright

    closed = [[0 for i in range(col)] for j in range(row)]
    closed[init[0]][init[1]] = 1

    expand = [[-1 for i in range(col)] for j in range(row)]
    expand[init[0]][init[1]] = 0
    action = [[-1 for i in range(col)] for j in range(row)]

    x = init[0]
    y = init[1]
    g = 0
    h = abs(goal[0] - init[0]) + abs(goal[1] - init[1])
```

```
f = g + h + 50 * topoEvaluate(x, y, map, avoidD)
open = [[f, g, h, x, y]]

found = False   # flag that is set when search complete
resign = False  # flag set if we can't find expand
count = 1

while found is False and resign is False:

    # check if we still have elements on the open list
    if len(open) == 0:
        resign = True
        return path

    # remove node from list
    else:
        open.sort()
        open.reverse()
        next = open.pop()
        x = next[3]
        y = next[4]
        g = next[1]
        expand[x][y] = count
        count += 1

        # check if we are done
        if x == goal[0] and y == goal[1]:
            found = True

        # expand winning element and add to new open list
        else:
            for i in range(len(delta)):
                x2 = x + delta[i][0]
                y2 = y + delta[i][1]

                if x2 >= 0 and x2 < row and y2 >= 0 and y2 < col:
                    if closed[x2][y2] == 0 and map[x2][y2][1] - map[x2][y2][0] > 1:
                        g2 = g + cost
                        h2 = abs(goal[0] - x2) + abs(goal[1] - y2)
                        f2 = g2 + h2 + 50 * topoEvaluate(x2, y2, map, avoidD)
                        open.append([f2, g2, h2, x2, y2])
                        closed[x2][y2] = 1
                        action[x2][y2] = i
```

```
x = goal[0]
y = goal[1]

while x != init[0] or y != init[1]:
    x2 = x - delta[action[x][y]][0]
    y2 = y - delta[action[x][y]][1]
    path.append([metrizeX(x2), metrizeY(y2)])
    x = x2
    y = y2

path.reverse()
return path
```

7. pathFollow.py

```
from time import sleep
from sys import maxint
from math import pi, atan2, sin, cos

from MRDS import getPose, postSpeed
from basic import quaternion2Euler, distance

lookAheadD = 1.6
k = 1.1
tolerance = 1.2
epsilon = 0.4
linearSpeed = 3.0

def nextPoint(x, y, path, lastPoint):
    global lookAheadD
    point = -1
    temp = maxint
    for i in range(lastPoint, len(path)):
        d = distance(x, y, path[i][0], path[i][1])
        if (d > lookAheadD) and (d < temp):
            point = i
            temp = d
    if point == -1:
        point = len(path) - 1
    return point

''' Follow The Carrot '''
def pathFollow(path, pose, target):
    global k, tolerance, epsilon

    x = pose['Pose']['Position']['X']
    y = pose['Pose']['Position']['Y']
    ori = quaternion2Euler(pose)

    psi = atan2(path[target][1] - y, path[target][0] - x)
    deltaOri = psi - ori
    if deltaOri > pi:
        deltaOri = deltaOri - 2 * pi
    if deltaOri < -pi:
        deltaOri = deltaOri + 2 * pi

    angularSpeed = k * deltaOri
```

```
response = postSpeed(angularSpeed, linearSpeed)

d = distance(x, y, path[target][0], path[target][1])
if (d > tolerance) or (d < epsilon):
    i = nextPoint(x, y, path, target)
else:
    i = target
return i
```

8. vfh.py

```
from MRDS import getLaserAngles, getPose, postSpeed
from basic import quaternion2Euler, indexMin
from math import ceil, pi
from time import sleep

laserAngles = getLaserAngles()
angularRes = 5
k = 1.2
linearSpeed = 0.8

def getRuns(ls):
    start = []
    end = []
    i = 0
    length = len(ls)
    while i < length:
        try:
            i += ls[i :].index(0)
            start.append(i)
            try:
                i += ls[i :].index(1)
            except ValueError:
                i = length
            end.append(i - 1)
            i += 1
        except ValueError:
            break
    return start, end

def sectorize(angle):
    global angularRes
    cAngle = ceil(angle / angularRes) * angularRes
    sector = int((cAngle - (-90)) / angularRes)
    return sector

def angularize(sector):
    global angularRes
    angular = (sector * angularRes + (-90) + angularRes / 2) / 180 * pi
    return angular

def vfh(pose, laser, psi):
    global laserAngles, angularRes, k, linearSpeed
```

```
robotOri = quaternion2Euler(pose)
if robotOri < -pi / 2:
    robotOri += pi
if robotOri > pi / 2:
    robotOri -= pi
sectorOri = sectorize(robotOri)

if psi < -pi / 2:
    psi += pi
if psi > pi / 2:
    psi -= pi
sectorTarget = sectorize(psi)

laserEchoes = laser['Echoes']
vfhAngles = laserAngles[45 : 224]
vfhEchoes = laserEchoes[45 : 224]

numSector = len(vfhAngles) / angularRes

# Polar Obstacle Density
bw = [0 for i in range(numSector)]
for i in range(numSector):
    temp = min(vfhEchoes[angularRes * i : angularRes * (i + 1) - 1])
    if temp < 5:
        bw[i] = 1

start, end = getRuns(bw)

if len(start) == 0:
    response = postSpeed(0, -0.8)
    sleep(5)
else:
    run = [0 for i in range(len(start))]
    deltaTarget = [0 for i in range(len(start))]
    sizeRun = [0 for i in range(len(start))]
    deltaOri = [0 for i in range(len(start))]

    for i in range(len(start)):
        run[i] = (start[i] + end[i]) / 2.0
        deltaTarget[i] = abs(sectorTarget - run[i])
        deltaOri[i] = abs(sectorOri - run[i])
        sizeRun[i] = end[i] - start[i] + 1
```

```
maxT = float(max(deltaTarget))
maxO = float(max(deltaOri))
maxS = float(max(sizeRun))

a = 0.2
b = 0.2
c = 0.6
g = [0 for i in range(len(run))]
for i in range(len(run)):
    deltaTarget[i] = deltaTarget[i] / maxT
    deltaOri[i] = deltaOri[i] / maxO
    sizeRun[i] = 1 - sizeRun[i] / maxS
    g[i] = a * deltaOri[i] + b * deltaTarget[i] + c * sizeRun[i]

index = indexMin(g)
targetOri = angularize(run[index])

count = 0
while count < 20:
    pose = getPose()
    robotOri = quaternion2Euler(pose)
    deltaOri = targetOri - robotOri

    if deltaOri > pi:
        deltaOri = deltaOri - 2 * pi
    if deltaOri < -pi:
        deltaOri = deltaOri + 2 * pi

    angularSpeed = k * deltaOri
    linearSpeed = 0.8
    response = postSpeed(angularSpeed,linearSpeed)
    count += 1
    sleep(0.2)
```

9. basic.py

```
from math import sqrt, atan2

def distance(x1, y1, x2, y2):
    d = sqrt(pow(x1 - x2, 2)+pow(y1 - y2, 2))
    return d

def sign(x):
    if x > 0:
        y = 1
    elif x < 0:
        y = -1
    else:
        y = 0
    return y

def quaternion2Euler(array):
    w = array['Pose']['Orientation']['W']
    x = array['Pose']['Orientation']['X']
    y = array['Pose']['Orientation']['Y']
    z = array['Pose']['Orientation']['Z']
    phi = atan2(2*(w*z+x*y),1-2*(pow(y,2)+pow(z,2)))
    return phi

def save(filename, var):
    f = open(filename, 'w')
    f.write(str(var))
    f.close

def size(x):
    row = len(x)
    col = len(x[0])
    return (row, col)

def indexMaxLs(ls, num):
    clone = []
    for i in range(len(ls)):
        clone.append(ls[i])
    indexMax = []
    for i in range(num):
        temp = max(clone)
        indexMax.append(ls.index(temp))
        clone.remove(temp)
```

```
    return indexMax

def indexMin(ls):
    temp = ls[0]
    index = 0
    for i in range(len(ls)):
        if ls[i] < temp:
            temp = ls[i]
            index = i
    return index

def filter(ls, resolution):
    length = len(ls) / resolution
    temp = []
    if length == 0:
        temp.append(ls[0])
    else:
        for i in range(length):
            temp.append(ls[resolution * i])
    return temp
```

10. visualize.m

```
clc; clear;
close all
load('map.txt')
load('pose.txt')
load('path.txt')
load('pathLength.txt')
load('mapFreq.txt')
row = 200;
col = 200;
timeStep = length(map) / (row * col);

occupied = zeros(row, col);
empty = zeros(row, col);
freq = zeros(row, col);
for i = 1 : row
    for j = 1 : col
        occupied(i, j) = mapFreq(((i - 1) * col + j - 1) * 3 + 1);
        empty(i, j) = mapFreq(((i - 1) * col + j - 1) * 3 + 2);
        freq(i, j) = mapFreq(((i - 1) * col + j) * 3);
    end
end
figure
imagesc(occupied)
figure
imagesc(empty)
figure
imagesc(occupied./freq)
figure
imagesc(empty./freq)
figure
imagesc(freq)

mapProb = zeros(row, col, timeStep);
for i = 1 : timeStep
    for j = 1 : row
        mapProb(j, :, i) = map(((i - 1) * row + j - 1) * col + 1 : ((i - 1) * row
+ j) * col );
    end
end

robotPose = zeros(timeStep, 2);
for i = 1 : timeStep
```

```
    robotPose(i, :) = pose((i - 1) * 2 + 1 : 2 * i);
end

maxL = max(pathLength);
trajectory = zeros(maxL, 2, timeStep);
k = 1;
for i = 1 : timeStep
    temp = path(k : k - 1 + 2 * pathLength(i));
    k = k + 2 * pathLength(i);
    for j = 1 : pathLength(i)
        trajectory(j, :, i) = temp((j - 1) * 2 + 1 : 2 * j);
    end
end

for i = 1 : timeStep
%     figure
    imagesc(-mapProb(:, :, i))
    colormap(gray)
    hold on
    plot(robotPose(i, 2), robotPose(i, 1), 'r*')
    hold on
    plot(trajectory(1 : pathLength(i), 2, i), trajectory(1 : pathLength(i), 1,
i), 'b-', 'LineWidth', 2)
    F(i) = getframe();
end
figure
movie(F, 1, 10)
% movie2avi(F, 'Exploration.avi','compression','None','fps',2);

figure
imagesc(-mapProb(:, :, timeStep))
colormap(gray)
```

11. mapper

```
#!/bin/bash

# Example script for calling mapper. Don't forget to make it executable (chmod +x mapper)
# Change the last line (java Mapper ...) to suit your needs
# Author: Ola Ringdahl
#
# Inputs:
# url specifies the address and port to the machine running MRDS.
# x1, y1, x2, y2 give the coordinates of the lower left and upper right corners of the
area the robot should explore and map.

if [ "$#" -ne 5 ]; then
    echo "Usage: ./mapper url x1 y1 x2 y2"
    exit
fi

url="$1"
x1="$2"
y1="$3"
x2="$4"
y2="$5"

python main.py $url $x1 $y1 $x2 $x2
```