Implementation of Obstacle Avoidance and HIMM Mapping on Nomadic Robot Simulator

Karan M. Gupta
Department of Computer Science
Texas Tech University
gupta@cs.ttu.edu

Abstract

This is a report on the Final Project for the Graduate AI Robotics Class, Fall 2003. The project has been subdivided into four parts, of which two were implemented:

1. Obstacle Avoidance for a Wandering Robot
2. Mapping of the World by a Wandering Robot

The project is implemented on the Nomadic Technologies Super Scout II Robot Simulator. The programming language used is C++. The compiler used is g++ (GCC) version 3.2.2. The project has been compiled and tested on a RedHat Linux machine successfully.

1 Introduction: The Objective

The objective of this project is to implement Obstacle Avoidance and Mapping for a mobile robot. The implementation is on a robot simulator. As a minimum the robot is required to wander around its environment without colliding with any obtacles that may be present in the environment. When the robot is able to carry out this task successfully, another behavior is added to it: mapping. So the second part boils down to mapping the environment while wandering in it, staying away from all obstacles. This study is a part of my course curriculum at the Department of Computer Science, Texas Tech University. The concepts used in this project are explained in the course textbook [4].

2 Implementation

The project has been implemented on a Nomadic Technologies Super Scout II Robot Simulator. The simulator presents a graphic environment, and an API which is usable in C or C++. The code for this project has been written using C++.

The implementation makes the use of three main behaviors:

1. A Wander behavior
2. An Avoid behavior
3. A Mapping behavior

The map is stored in an Occupancy Grid, and is generated by using the HIMM sonar sensor model. The addition of a signal handler allows the program to exit cleanly. Before exiting, the program creates a pgm £le (map.pgm) and writes the data in the occupancy grid to this £le, thereby creating a copy of the learned map.

2.1 The Wander Behavior

The wander behavior is made up of two behaviors, move_straight (`pr`) and `turn_little`. The wander behavior is carried out at all times, however, if there is an obstacle in front of the robot, then the wander behavior is subsumed by the avoid (`turn_away`) behavior. While all these behaviors are going on, the program keeps refreshing the state of the robot to re¤ect the actions of the commands on the screen simultaneously.

A certain amount of randomness has been introduced into the program. The wander behavior

```
while(1)
   random(distance)
   pr(distance, distance)
   while(robot_moving)
      getstates()
   random(turn or noturn)
   if (turn)
      random(direction)
      turn_little(direction)
```

Figure 1: The Wander Behavior

chooses a random distance to move, and at times, chooses a random direction to turn. This is implemented by the `turn_little` function. The probability for choosing to turn is 0.5. The algorithm for the wander behavior is shown in Figure 1.

2.2 Obstacle Avoidance

The Obstacle Avoidance behavior keeps the robot safe by not allowing it to bump into objects in the environment. This behavior is implemented by two functions: land_ahead and turn. This behavior only uses the sensors at the front (numbered 0 - 2 and 13 - 15). While wandering the sensors keep looking a watch out for proximity to obstacles. When an obstacle is closer than a certain threshold, the turn behavior completely inhibits Wander and causes the robot to keep turning until the robot is no longer facing the obstacle. Then the robot resumes wandering. The direction in which the robot turns (clockwise or anticlockwise) is chosen at random with both having an equal probability of being chosen. The algorithm for the Avoid Behavior is shown in Figure 2.

```
while(1)
   while(not(land_ahead))
      Wander
   random(direction)
   turn(while(land_ahead))
```

Figure 2: The Avoid Behavior

2.3 Mapping

The mapping function is called by the wander behavior. So, wandering and map-making go hand-in-hand. As the robot moves around in its environment it keeps storing the sonar readings that bounce off nearby obstacles. An Occupancy Grid structure is used to store the generated map. In this technique, a 2D cartesian grid is superimposed on the world space. If there is an obstacle in the area covered by a grid element, that element is marked as being occupied. This implementation uses the Histogrammic In-Motion Mapping [1] (HIMM) algorithm to generate the map.

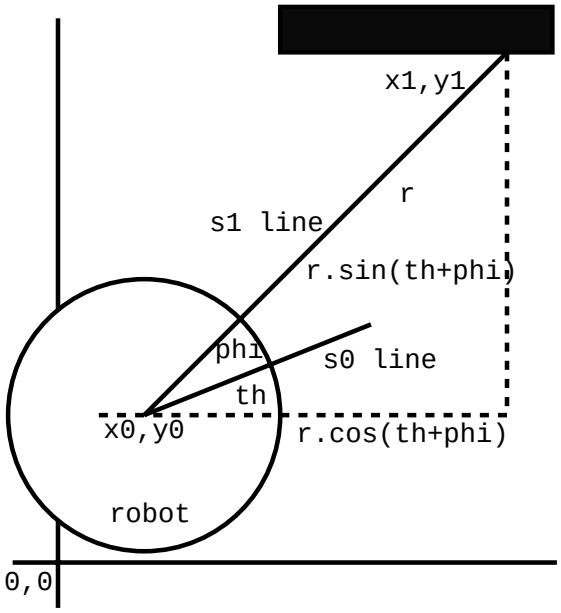

Figure 3: Calculating Index of an Element

In the HIMM sonar sensor model only the elements of the occupancy grid that fall directly under the acoustic axis of the sonar are updated. The uncertainty score is expressed as an integer in the range 0 to 15. Like other sonar sensor models, HIMM assumes that the sonar's range reading is coming from an element on its acoustic axis. Upon receiving a sonar range reading the algorithm updates all the elements directly under the acoustic axis as being empty, and updates the element at the sonar reading as being occupied. A Growth Rate Operator is applied as an extra step everytime an element is updated with an occupied reading. The Growth Rate

```
forall sensors
   getstates()
   x0, y0 = co-ords of robot
   φ = angle of sonar to sonar0
   θ = angle of sonar0 to horizontal
   r = range reading from the sonar
   x1 = x0 + r * cos(φ + θ)
   y1 = y0 + r * sin(φ + θ)
   UseBresenham's Algo find
   all values xb, yb from
   x0, y0 to x1-1, y1-1
      forall xb, yb
         grid[xb][yb] = empty
   If r < Range Threshold
      grid[x1][y1] = occupied
      Apply W, the GRO mask
```

Figure 4: The Mapping Behavior

Operator uses a mask, W, to extract the occupancy reading of the neighbouring elements, of the one being updated, and uses that information to better judge the occupancy of the element of interest.

To perform the above steps, the robot needs to know which element of the occupancy grid the sonar reading is coming from. This is done by applying some trigonometry to the problem. The diagram for the method used is in Figure 3. A standard geometrical formula [3] is being used to calculate the grid index of the element where the obstacle is at. By looking at the £gure we see that:

$x_1 = x_0 + rcos(\theta + \phi)$

$y_1 = y_0 + rsin(\theta + \phi)$

Since the value of x_1 and y_1 is now known, we can apply Bresenham's Line Drawing Algorithm [5] to £nd out the grid index of all elements that are on the acoustic axis of the sonar. All the elements but one on the acoustic axis are thus updated as being empty and the element at x_1 and y_1 is marked as being occupied. The complete Mapping Behavior algorithm is shown in Figure 4.

If the program is run for a suf£cent amount of time, depending in the size of the environment, a map can now be generated. This map is written to a pgm £le upon exit. If the value of an in the occupancy grid is more than a certain threshold it is marked as being occupied (black), otherwise its marked as being empty (white).

3 Results

The wander and obstacle avoidance were tested in a number of environments and they work correctly. However, the Mapping procedure is not working as desired. It was noticed that the map that the robot makes is of its own path while wandering around the world. This path is slightly shifted with repect to the actual path the robot has taken. This leads me to believe that the mathematical equation for getting the values of x_1 and y_1 are incomplete or incorrect. However, upon looking at the £gure the given mathematical formulae would be the most logical way of getting the values for x_1 and y_1.

4 Conclusion

Currently this implementation is limited by the size of the array used to store the occupancy grid. This limits the size of the map which this implementation can work on. Dynamically expanding occupancy grids have been explored by Bharani [2] which seek to remove dependency on an array for representing an occupancy grid. This can allow a working implementation to store any map, only limited by the robot's memory. On the other hand, if a very large array is being used to store the occupancy grid, but the map is a small one, a large amount of memory is being wasted. Dynamically expanding occupancy grids tackle both these issues.

Also, the project can be improved to use an existing map. This way the robot can repair or improve upon the map, and as a side-effect we will also be able to introduc redundancy in the implementation.

5 Acknowledgements

I would like to thank Dr. Larry Pyeatt for providing very useful and sometimes, much needed, guidance all throughout this course.

References

[1] J. Borenstein and Y. Koren. Histogrammic in-motion mapping for mobile robot obstacle avoidance. *IEEE Journal of Robotics and Automation*, 7(4):535 – 539, 1991.

[2] Bharani K. Ellore. Dynamically expanding occupancy grids. Master's thesis, Texas Tech University, Department of Computer Science, November 2002.

[3] M. L. Khanna and J. N. Sharma. *I.I.T. Mathematics*, chapter 18, page 1059. J.P.N. Publications, Meerut, U.P. India, hundred thirty £rst edition, 2002-2003.

[4] Robin R. Murphy. *Introduction to AI Robotics*. The MIT Press, Cambridge, Massachusetts, England, 2000.

[5] Wikipedia. http://en.wikipedia.org/wiki/bresenham's_line_algorithm_c_code. World Wide Web.