Optimal Path Planning in a Dynamic Environment

Aakash Gupta, Sachin Umrao and Sumit Kumar

Abstract— **The aim of this project is to study and implement path planning algorithms on a mobile robot working in a dynamic environment like warehouse delivering packages from one place to another. During its journey if the robot encounters any obstacle, it replans its trajectory and proceed to the target. The system was developed entirely in Robot Operating System(ROS) and Gazebo. A 2D occupancy grid map has also been built using MIT-CSAIL datset.**

Index Terms– Mobile Robot, Path Planning, Robot Operating System, Gazebo, Dynamic Environment

I. INTRODUCTION

Mobile robots are widely used in many industrial fields. Research on path planning for mobile robots is one of the most important aspects in mobile robots research. Path planning for a mobile robot is to find a collision-free route, through the robot's environment with obstacles, from a specified start location to a desired goal location while satisfying certain optimization criteria. One promising field of implementation of optimal path planning is in warehouses like those of Amazon, DHL and FedEx. Robots bring shelves of goods out of storage and carry them to employees for shipment of product. The motivation behind using robots instead of humans is to achieve lesser delivery time of goods to desired location, and thus generate larger profits for companies. These warehouses are not always the same and keep changing with time. A robot can encounter dynamic obstacles in its path like an object or some other robot. So, path planning algorithms with the ability to handle dynamic obstacles are implemented in such systems.

In this project, we have demonstrated simulation of a differential drive robot in a warehouse like environment and also tested path planning algorithms on it, like Dijkstra's Algortithm[1], A*[2], Relaxed A*, Lifelong Plannig A*[3] and D* Lite[4]. To accomplish the task, we have made use of Robot Operating System(ROS) and Gazebo simulator. In ROS[5], we have used urdf tool for modeling a robot and navigation stack for moving that robot in the simulated environment. Gazebo[6] has been used as a virtual world for the robot.

A. Terminology

We have indistinguishably used words like 'world' and 'environment' to refer to robot's configuration space. The grid cells of the environment have been referred by 'vertex' and 'node'.

This project is a part of Probabilistic Mobile Robotics Course at Indian Institute of Technology Kanpur, India under the guidance of Dr. Gaurav Pandey.

Aakash Gupta, Sachin Umrao and Sumit Kumar are junior undergraduate students in the department of Mechanical Engineering at Indian Institute of Technology Kanpur, India.

II. SYSTEM DEVELOPMENT

We modeled a simple differential drive robot with two motor actuated wheels on the sides and two castors in front and back as shown in Fig.1. We attached a Hokuyo 2D laser scanner in front of the robot to receive information about the world during its motion. Wheels are equipped with wheel encoders to obtain odometry data of the robot. The modeling of robot has been done in ROS using its URDF[7] tool. We created a 3D world like a warehouse in Gazebo simulator as shown in Fig.2.

III. ROS NAVIGATION STACK

ROS Navigation stack[8] is a collection of packages built with the objective of performing navigation of a mobile robot in an environment. It takes in information from odometry, sensor streams, and a goal pose and outputs safe velocity commands that are sent to a mobile base which then moves the robot. If properly configured, the navigation stack can move robot to the desired goal location preventing collisions with obstacles or getting lost. There are three main requirements-

- The navigation stack can only handle a differential drive and holonomic wheeled robots.
- A planar laser (or equivalent sensors like Kinect) must be mounted on the mobile base of the robot to perform localization in the environment.
- It is designed and hence works best for robots that are nearly square or circular.

Navigation stack requires a map of the world to move the robot. One of the techniques used by robots to build a map within an unknown environment while keeping track of the positions is Simultaneous localization and mapping[9] (SLAM). We manually moved our robot in Gazebo modeled world and obtained corresponding lidar and odometry data. These data were then used by ROS gmapping package to build a 2D occupancy grid map[10] (OGM) of the world. Gmapping implements FastSLAM[11], which is a particle filtering based algorithm.

The costmap is the data structure that represents places that are safe for the robot to be in a grid of cells. Usually, the values in the costmap are binary, representing free space or places where the robot would be in collision. There are two types of costmaps- global and local. Both are generated using the 2D occupancy grid map (OGM).

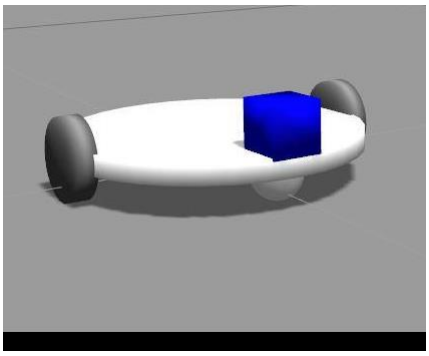

Fig. 1: Robot modeled using ROS urdf

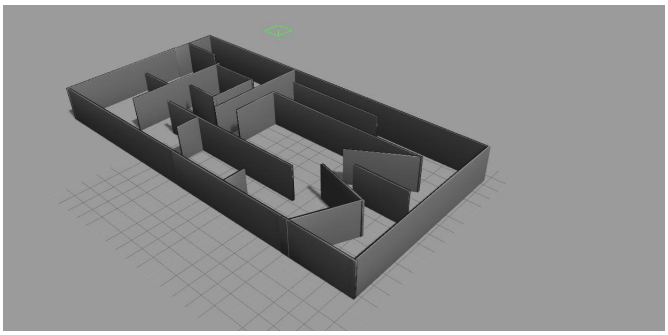

Fig. 2: Robot's world modeled in Gazebo

- The global costmap is used for the global navigation. The global navigation is used to create paths for a goal in the map or a far-off distance.
- The local costmap is used for the local navigation. The local navigation is used to create paths in the nearby distances and avoid obstacles.

The robot moves through the map using both global and local navigation. The global navigation plan is computed before the robot starts moving toward the next destination. The planner assumes a circular robot and operates on global costmap to find a minimum cost plan from a start point to an end point in a grid. By default, the navigation function is computed using Dijkstra's algorithm. Instead of using it, we implemented Relaxed A* and D* Lite path planning algorithms for our robot.

The local navigation planner monitors incoming sensor data and chooses appropriate linear and angular velocities for the robot to traverse the current segment of the global path. It combines odometry data with both global and local costmaps to select a path for the robot to follow. It also has the ability to re-compute the robot's path during its motion to avoid collision with obstacles yet still allowing it to reach its destination. If the local planner is not able to plan a path for robot to avoid obstacles and proceed to goal, then it asks the global planner to do so. In that case, the global planner replans a path from current position to goal location.

Once a plan is computed, the robot is required to move accordingly in the world. This task is done by ROS move_base package using the navigation stack. The move_base node links together global and local planner to accomplish its global navigation task. It uses the linear and angular velocities provided by the local planner and then computes the required motor commands. In cases where the robot is lost or can't find a way to its goal location, the move_base node performs recovery behaviors to get back to a previous location.

Since robot's motion is not 100% perfect, so there is a need to continuously track robot's location in the environment while it is traversing towards its target. This problem of estimating the pose of the robot relative to a map is known as localization. Localization is not terribly sensitive to the exact placement of objects so it can handle small changes to the locations of objects. ROS uses amcl (adaptive monte carlo localization[12]) package for localization. Amcl is a probabilistic localization system for a robot moving in 2D world. It implements the adaptive Monte Carlo localization approach, which uses a particle filter to track the pose of a robot against a known map. Currently, ROS amcl package supports only laser scans and laser maps.

For visualization of robot's motion in the world, we used Rviz (ROS Visualization Tool). Rviz also allows us to set the pose of the robot for a localization system like amcl. It can also display all the visualization information that the navigation stack provides. We can also send goal locations to the navigation stack with Rviz.

IV. PLANNING ALGORITHMS

A Path planning task consists of finding a set of consecutive actions that safely transforms a robot or a body from some initial configuration to a final configuration while avoiding any obstacles. The robot and obstacles are described in 2D/3D workspace while the trajectory (or path) is described as a curve in its configuration space.

A common method for robot's path planning is to represent the configuration space of the robot as a directed graph $G = (S, E)$ where S is a set consisting of all possible robot locations and E is a set of edges connecting these locations. A cost is associated to each edge denoting the cost of transition between the two vertices on the edge. The cost of traversing to or from an obstacle location is infinite. Doing so, the path planning problem simply reduces to a search problem on the graph G. An optimal path is the one having minimum total cost (sum of the all the connecting edges' cost) across all possible paths from start position to goal position.

A number of classical graph search algorithms have been developed across the years for calculating optimal paths on a graph. A few of them have been described briefly.

A. Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest paths between nodes in a graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later. The algorithm maintains a queue of nodes to be expanded in increasing order of their value. The algorithm works as follows-

1) Set the value of start node as 0. Put the start node on the queue. Mark it as the current node.

2) Expand the current node and calculates its neighboring nodes' value as the sum of parent node's value and the transition cost on the connecting edge. If any cell is an obstacle, then don't calculate its value. So, the value of a node can be seen as the minimum cost of reaching there from the start node.
3) Remove the current node from the queue and put all the neighboring nodes on the queue in increasing order of their value.
4) Mark the node with the least value on the queue as the current node and goto step 2.

The algorithm terminates when either the goal node is reached or the queue has emptied. In the latter case, there doesn't exist any path from start node to goal node.

*B. A**

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute first described the algorithm in 1968. It is an extension of Edsger Dijkstra's algorithm. A* achieves better performance by using heuristics to guide its search.

A* algorithm calculates the value of a node (called g-value) bit differently from Dijkstra's algorithm. The value of a node is assigned the sum of the cost from the start node as done in Dijkstra's algorithm and the heuristic value of the node. There are different ways in which this heuristic is computed like Manhattan, Euclidean, Octile and Chebyshev. A* is more efficient than Dijkstra's algorithm especially in large open spaces because unlike exploring every neighboring node, it guides its search only towards those nodes which have a high probability to lead robot to the goal in less number of steps.

*C. Relaxed A**

RA* is a time linear relaxed version of A*. It is proposed to solve the path planning problem for large scale grid maps. The objective of RA* consists of finding optimal or near optimal solutions with small gaps, but at much smaller execution times than traditional A*. The core idea consists of exploiting the grid-map structure to establish an accurate approximation of the optimal path, without visiting any cell more than once.

All the above algorithms work well in a static environment, i.e., with known locations of start node, goal node and obstacles. However operating in real world is different - robot has none or some information about the world. So, during its motion if any new obstacle is found or it can not continue on the estimated optimal path, then it should be able to dynamically update its map and compute the optimal path to the goal location. One possible solution is to recompute the trajectory in the updated map from scratch but it is computationally expensive and inefficient.

Path planning algorithms like Lifelong Planning A*, D*[13] and D* Lite can perform local modifications in the path of robot to handle dynamic obstacles.

*D. Lifelong Panning A**

Lifelong Panning A* (LPA*) is an incremental version of A*. It repeatedly finds shortest paths from a start vertex to a goal vertex in a given graph as edges or vertices are added or deleted or the costs of edges are changed.

The finite set of vertices of the graph is denoted by S. $Succ(s)$ and $Pred(s)$ denotes the set of successors and predecessors respectively of vertex $s \in S$. $0 < c(s, s') <= \infty$ denotes the cost of moving from vertex s to vertex s'. Heurisitics $h(s, s_{goal})$ are used by LPA* to approximate the goal distances of the vertices s. The heuristices need to be non-negative and consistent,i.e, obey the triangle inequality $h(s_{goal}, s_{goal}) = 0$ and $h(s, s_{goal}) <= c(s, s') + h(s', s_{goal})$ for all vertices $s \in S$ and $s' \in Succ(s)$ with $s \neq s_{goal}$.

LPA* maintains two estimates of the start distance of each cell, namely a g-value($g(s)$) and an rhs-value($rhs(s)$). The g-values directly correspond to the g-values of an A* search. The rhs-values are one-step lookahead values based on the g-values and thus potentially better informed than the g-values. The rhs-value of the start cell is zero. The rhs-value of any other cell is the minimum over all of its neighbors of the g-value of the neighbor and the cost of moving from the neighbor to the cell in question. Mathematically, it can be expressed as:

$$rhs(s) = \begin{cases} 0 & s = s_{start} \\ min_{s' \in Pred(s)}(g(s') + c(s', s)) & otherwise \end{cases} \quad (1)$$

LPA* maintains key values of all the vertices. A key is a vector with two components: $k(s) = [k1(s); k2(s)]$ where $k1(s) = min(g(s), rhs(s)) + h(s, sgoal)$ and $k2(s) = min(g(s), rhs(s))$. The first component of the keys $k1(s)$ corresponds directly to the f-values used by A* and the h-values of LPA* correspond to the h-values of A*. The second component of the key $k2(s)$ corresponds to the g-values of A*. Keys are compared according to a lexicographic ordering.

A node s is overconsistent if $g(s) > rhs(s)$, underconsistent if $g(s) < rhs(s)$ and consistent otherwise. LPA* maintains a priority queue of vertices. The priority queue always contains the locally inconsistent vertices at that time. These are the vertices whose g-values needs to updated to make them locally consistent. The priority of a vertex in the priority queue is always the same as its key. The vertices are kept in increasing order of their key. Hence, LPA* always expands the vertex in the priority queue with the smallest key.

LPA* sets the initial g-values and rhs values of all nodes to infinity. It then updates $rhs(s_{start})$ as 0. Thus, the start node is initially the only locally inconsistent state and is inserted into the otherwise empty priority queue. It thus expands the locally inconsistent states in non-decreasing order of their priorities.

When a locally overconsistent node is expanded, then its g-value is set equal to its rhs value. On the other hand, when a locally underconsistent node is expanded, then its

g-value is assigned to be infinity. If the expanded vertex was overconsitent, then the change of its g-value affects its successors' consistency. Similarly, expanding an underconsistent node affects consistency of its successors and itself. All those vertices who become inconsistent after expansion of a vertex are put on the priority queue.

LPA* expands nodes until s_{goal} is locally consistent and the key of the first vertex on the priority queue is greater than or equal to that of s_{goal}. If $g(s_{goal})$ is finite after the search, then there exists a path from s_{start} to s_{goal}. One can then locate the shortest path from s_{start} to s_{goal} by always moving from the current vertex s, intially s_{goal}, to the predecessor s' that minimizes $g(s') + c(s, s')$ until s_{start} is reached.

During robot's motion, if any obstacle is detected in the path,i.e, any edge cost has changed, then LPA* updates the rhs-values and keys of all the vertices potentially affected by it. All the inconsistent vertices are put on the priority queue and LPA* determines the new shortest path from start vertex to goal vertex.

E. D Lite*

D* Lite algorithm is inspired from LPA*. It repeatedly determines shortest paths between the current vertex of the robot and the goal vertex as the edge costs of a graph change while the robot moves towards the goal vertex. It is derived from LPA* by exchanging the start and goal vertex and reversing all edges in the pseudo code. Following is the pseudo code of D* Lite algorithm:

```
procedure CalculateKey(s)
{01'} return [min(g(s), rhs(s)) + h(s_start, s) + k_m; min(g(s), rhs(s))];
procedure Initialize()
{02'} U = ∅;
{03'} k_m = 0;
{04'} for all s ∈ S rhs(s) = g(s) = ∞;
{05'} rhs(s_goal) = 0;
{06'} U.Insert(s_goal, CalculateKey(s_goal));
procedure UpdateVertex(u)
{07'} if (u ≠ s_goal) rhs(u) = min_{s'∈Succ(u)}(c(u, s') + g(s'));
{08'} if (u ∈ U) U.Remove(u);
{09'} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{10'} while (U.TopKey() < CalculateKey(s_start) OR rhs(s_start) ≠ g(s_start))
{11'}   k_old = U.TopKey();
{12'}   u = U.Pop();
{13'}   if (k_old < CalculateKey(u))
{14'}     U.Insert(u, CalculateKey(u));
{15'}   else if (g(u) > rhs(u))
{16'}     g(u) = rhs(u);
{17'}     for all s ∈ Pred(u) UpdateVertex(s);
{18'}   else
{19'}     g(u) = ∞;
{20'}     for all s ∈ Pred(u) ∪ {u} UpdateVertex(s);
procedure Main()
{21'} s_last = s_start;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while (s_start ≠ s_goal)
{25'}   /* if (g(s_start) = ∞) then there is no known path */
{26'}   s_start = arg min_{s'∈Succ(s_start)}(c(s_start, s') + g(s'));
{27'}   Move to s_start;
{28'}   Scan graph for changed edge costs;
{29'}   if any edge costs changed
{30'}     k_m = k_m + h(s_last, s_start);
{31'}     s_last = s_start;
{32'}     for all directed edges (u, v) with changed edge costs
{33'}       Update the edge cost c(u, v);
{34'}       UpdateVertex(u);
{35'}     ComputeShortestPath();
```

Fig. 3: D* Lite pseudo code

V. MAP GENERATION TOOL

To test our algorithm on various environments quickly, we created a utility for creating small maps. This ensured that we could create our own maps without relying on datasets for testing of path planning algorithms. So, until the algorithm for generating map from the Lidar sensor data was developed, we relied on MGT (Map Generation Tool) for various test case scenarios. MGT was created with Processing 2.2.1, which is a framework based on java for creating visuals. The size of the world is taken to be 600x600 sq. cm and size of individual grid is 30x30 sq. cm thus we get 20 rows and 20 columns of grids- a total of 400 grids.

The GUI is very easy to work with, a mere click on a grid will create an obstacle on that grid. After creating all the obstacle one needs, by clicking generate button, one gets an array of zeros and ones, which corresponds to whether a particular grid is empty, the size of the one-dimensional output array is 400, denoting whether a grid is empty or not (since this is not a probabilistic map). The minimalistic demo of MGT can be found on the following link:

http://home.iitk.ac.in/ aakashg/ee698g/

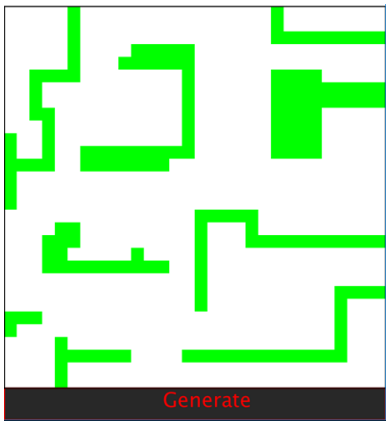

Fig. 4: Map generation tool in Processing

VI. BUILDING MAP FROM DATSET

Light Detection and Ranging sensors are most commonly used in autonomous robot navigation to create a map of an unknown environment so that the robot will avoid obstacles during its motion. We used a 180° LIDAR sensor to get wide view of the environment. LIDAR gives an array consisting of N values where N is given by: $N = 180/m$ (where m = angular difference between consecutive readings of LIDAR.) In our case of dataset, m = 0.5° and N = 361. This values correspond to 0°, 0.5°, and 1.0° and so on till 360°. Using above information, we construct an array of [r, θ] consisting of N values. Now we convert this array of polar coordinates [r, θ] to an array of Cartesian coordinates [x, y] to get the corresponding LIDAR values in Cartesian coordinates for easy grid assignment. X= r*cos (θ) Y= r*sin (θ) All the sensors have some error in measurement, and so is the case with LIDAR. We need a co-variance matrix for LIDAR

sensor. The standard co-variance matrix for a 2-D LIDAR is taken to be 2x2 matrix. Co-variance matrix of 2-d Lidar sensor is as follow-

Σ r =

$$\begin{bmatrix} (\delta r)^2 & 0 \\ 0 & (\delta r)^2 * (\delta\theta)^2 \end{bmatrix}$$

Where δr is the error in range measurement and $\delta\theta$ is the error in angle measurement. For the dataset we used δr = 5 cm and $\delta\theta$ = 0.2°. To convert this co-variance matrix Σ r from polar to Cartesian coordinates, we need to employ propagation of uncertainty in case of non-linear combinations. We overcome this problem by taking Linear approximation to the [x,y] = f(r,θ). This gives

$$\Sigma x = J*(\Sigma r)*J'$$

where J is the jacobian matrix defined as

$$J= \partial f/\partial x$$

After finding the co-variance matrix, we find the probability of each grid cell to be occupied. We have a 2-D array consisting of [x,y] and co-variance matrix for sensor. So, given a point, we find the area under the grid of the bi-variate normal distribution whose mean is the point of LIDAR data and co-variance is the corresponding co-variance matrix.

$$P(mi) = \frac{1}{(2pi|\Sigma x|)} * exp(-(x - x_0)'(\Sigma x)^{-1}(x - x_0/2)$$

where $x = [x, y]'$ and $x_0 = [X, Y]'$ Where X = x-coordinate of centre of grid in which point (x,y) lies and Y = y-coordinate of centre of grid in which point (x,y) lies. This represents the probability of obstacle in grid due to the sensor measurement at that instant. Now we combine this probability to previous probability of occupancy of same grid as follows

$$P(m_i) = P(m_i)/(1-P(m_i))xP(m_{i-1})/(1-P(m_{i-1}) * Pi/(1-Pi)$$

Where Pi is taken to be 0.5 and $P(m_{i-1})$ is probability of occupancy of grid in previous measurement. We repeat same process for all the observed data points from Lidar to find Probabilistic map of environment. After finding the probability values in each grid, we create the map using a given threshold. If the Probability value of a grid is greater than the set threshold (0.6 in our case), we assign that grid to be full, and else it is assigned empty. In a way, we converted the probabilistic map into a static one.

VII. RESULTS

We used ROS navigation stack to autonomously navigate our robot in the environment created in Gazebo.We successfully implemented Relaxed A-star and D-star Lite path planning algorithms in global planner of ROS (Robot Operating System) navigation stack in real time. To show the simulation of robot we used Rviz and Gazebo. Robot successfully detected the obstacles introduced in the environment and modified its path to reach the destination.

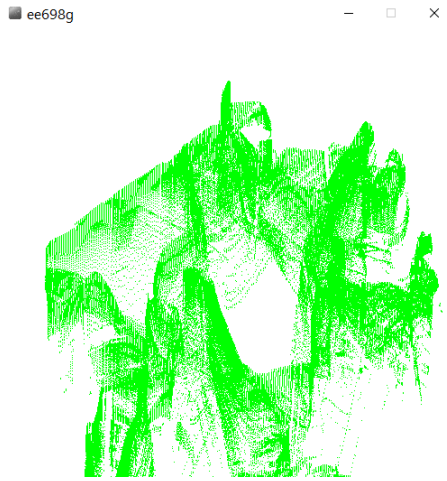

Fig. 5: Map build from MIT-CSAIL dataset in Processing software

Following map was obtained by implementing mapping algorithm on the dataset obtained from MIT-CSAIL. In the map, green points denote the obstacles detected by Lidar sensor and white area shows the available path to move.

VIII. FUTURE WORKS

At present our robot is able to detect static obstacles introduced in environment and replan its path but gets confused when several dynamic obstacles are introduced in the environment. In next stage we will work to make our path planning algorithm more robust so that it could handle several dynamic obstacles at same time.

In mapping in next stage we will work to filter scattered points which don?t belong to obstacles but present due to noise in sensor. We will also modify our occupancy grid map algorithm to account for effect of a point present in one cell on probability of other cell. This will provide us more precise estimation of probabilistic map.

ACKNOWLEDGMENT

We are highful indebted to Dr. Gaurav Pandey, assistant professor in Electrical Engineering Department, Indian Institute Of Technology Kanpur for guiding us throughout the project and clarifying all our queries regarding project and concepts we implemented.

We are also thankful to Lentin Joseph, founder and CEO of Qbotics Lab for providing us useful insights regarding ROS navigation stack and Gazebo and helping us in developing our codes.

We also thank Professor Dr. Cyrill Stachniss, University of Bonn for providing Lidar data set of MIT-CSAIL[14].

REFERENCES

[1] Dijkstra, E. 1959. A note on two problems in connexion with graphs Numerische Mathematik 1:269-271.

[2] Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEETransactions on Systems Science and Cybernetics SSC4 4 (2): 100-107.

[3] Koenig, S.; Likhachev, M.; Furcy, D. (2004), "Lifelong Planning A*", Artificial Intelligence Journal 155 (1-2): 93-146..
[4] Koenig, S.; Likhachev, M. (2005), "Fast Replanning for Navigation in Unknown Terrain", Transactions on Robotics 21 (3): 354-363.
[5] http://www.ros.org/about-ros/
[6] http://gazebosim.org/).
[7] http://wiki.ros.org/urdf.
[8] http://wiki.ros.org/navigation.
[9] (S. Thrun, W. Bugard, and D. Fox, A real-time algorithm for mobile robot mapping with applications to multi-robot and 3D mapping, in Proc. Int. Conf. Robotics Automation, 2000, pp. 321-328.
[10] Thrun, S. 2001. Learning occupancy grids with forward models. In Proceedings of the Conference on Intelligent Robots and Systems (IROS-2001), Hawaii.
[11] M. Montemerlo, S. Thrun, D. Koller, and B. Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. Proc. AAAI, 2002.
[12] F. Dellaert, D. Fox, W. Burgard, S. Thrun, Monte Carlo localization for mobile robots, in: Proc. IEEE International Conference on Robotics and Automation (ICRA-99), Detroit, MI, 1999.
[13] Stentz, Anthony (1994), "Optimal and Efficient Path Planning for Partially-Known Environments", Proceedings of the International Conference on Robotics and Automation: 3310-3317.
[14] http://www2.informatik.uni-freiburg.de/ stachnis/datasets.html .