2015

FACULTY OF COMPUTER AND INFORMATION SCIENCES

[CO-SWARM:]

** CO-Swarm -- is a Multi-Robot System (MRS) used for solving one of challenging real life problems which is "Area Exploration" based on the invented bio-inspired techniques. This system implements some of well-known approaches like Frontier-Based exploration enhanced with shortest paths techniques that work on static environments like (A*) algorithm and dynamic environments as well like (D* Lite).The system is built on real ground vehicles named Robo-Cars and on a simulated environment with Robot Operating System (ROS) which is a trusted robotics environment and simulation software named (Gazebo and RVIZ)

Co-Swarm

This documentation submitted as required for the degree of bachelors in Computer and Information Sciences July 2015.

By

Aya Hassan [Computer System]
Hoda Khaled [Computer System]
Elshaimaa Hassan [Computer System]
Mennat-Ullah Hesham [Computer System]

Under Supervision of

[Professors]

[**DR . Eman shaaban**],
Computer Systems Department.
[**DR. Sherin Rady**],
Information Systems Department.

[Teacher Assiatants]

[**DR. Abd El-Rahman Ehab**],
Computer Systems Department.
[**DR. Dalia Shoman**],
Computer Systems Department.
[**DR. Mona Essam**],
Computer Systems Department.

[Non Academic supervisors]

[Anas Awad],
[Ahmed Morsy],
[Mohamed Ali],
Computer Systems Department ’12-‘13.

Acknowledgement

We would like to express our appreciation to the advisors who helped us on the project, **DR. Eman Shaaban** from Computer Systems department and **DR. Sheiren Rady** from Information Systems department. Their advices, helpful suggestions and guidance throughout the project really helped us during the project's planning and implementation phases.

We also would like to express our sincere gratitude and appreciation to **T.A. Abdelrahman**, **Anas A. Ismail**, **Ahmed Emad Moursy** and **Mohamed Aly Ehab** for their continuous support and valuable input which led us to the right way. Without their help, support and advices we may haven't done that much.

And very special thanks to **"Dr. Menna seyam"** and "**Mr.Juan de Dios Flores Mendez"** from Denmark for their great support, advices and guidance in the field of simulation.

Finally, we would like to express appreciations to our families, friends and people who continuously supported us during the project's implementation and studying phases.

We are extremely thankful to you all.

Abstract

** **CO-Swarm --** is a Multi-Robot System (MRS) used for solving one of hardest real life problems which is "***Area Exploration***" based on the invented bio-inspired techniques.

This system implements some of well-known approaches like Frontier-Based exploration enhanced with shortest paths techniques that work on static environments like (A*) algorithm and dynamic environments as well like (D* Lite).

The system is built on real ground vehicles named Robo-Cars and on a simulated environment with Robot Operating System (ROS) which is a trusted robotics environment and simulation software named (Gazebo and RVIZ)

Table of Contents

Chapter 1: Introduction 1

1.1 Overview 2

1.2 Motivation 3

1.2.1 Applications 5

1.3 Problem Definition 6

1.4 Objectives 6

Chapter 2: Background 9

2.1 Area Exploration Problem 10

2.2 Exploration Strategies 11

2.3 Swarm Robotics Survey 11

2.3.1 Multi-Robot Systems (MRS) 11

2.3.2 Swarm Robotics Characteristics 12

2.3.3 Towards Cooperative Swarm Robotic Systems (CO-SWARM) 14

2.4 Self-Organizing Models 16

2.4.1 Mobility models of mobile sensors 17

2.4.2 Emergent Motion Control model 17

2.4.2.1 Anti-Flocking algorithm 18

2.5 Frontier-Based Exploration 20

2.5.1 Mutli-Robot Exploration extended 22

2.6 Previous Work 24

Chapter 3: ROS Robot Operating System 29

3.1 Introduction 30

3.1.1 ROS Software Distributions 31

3.1.2 ROS Supported Platforms 32

3.1.3 ROS Package 33

3.1.4 Programming with ROS 33

3.1.5 ROS License 33
3.2 ROS Architecture 34
3.2.1 ROS File system Level 35
3.2.2 ROS Computation Graph Level 36
3.2.3 ROS Community Level 39
3.3 Simulation 40
3.4 Visualization 41

Chapter 4: System Design 43
4.1 Overall System Architecture 44
4.2 Robo-Car Framework '13 46
4.3 Motion Planning Design 48
4.3.1 A*Package 48
4.3.2 D*-Lite Package 50

Chapter 5: Implementation 55
5.1 Motion Planning Module 56
5.1.1 A* Search Algorithm 57
5.1.2 D*-Lite Search Algorithm 59
5.2 ROS and Simulation implementation 65
5.2.1 Robot's 3D model implementation 65
5.2.1.1 Joints and Links 65
5.2.1.2 Sensors 65
5.2.1.3 Transform Tree 70

Conclusion and future work 75

References 79

CHAPTER 1: INTRODUCTION

1.1 Overview

Living through the computer era or the information age or whatever it's called, every new emerged problem in different fields like medicine, communication, transportation, military and others, is expected to be solved using computing techniques.

But as the problems get more complicated, traditional hardware and software can do nothing in solving them, hence new computing techniques are needed which can solve these problems efficiently and in the most optimized way.

One of the most important techniques in this era which is a new alternative for other standard computing techniques is the ***Bio-Computing***.

In ―Co-Swarm'13― project and this project ―Co-Swarm'15― we picked one of these challenging problems which is "***Area Exploration Problem***" and we tried to solve it using one of the Bio-Computing techniques called ***Swarm Intelligence (SI)***.

Swarm Intelligence (SI) systems are typically made up of a population of simple agents interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems. The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave or localize, interactions between such agents lead to the emergence of "intelligent" global behavior, unknown to the individual agents.

"The emergent collective intelligence of groups of simple agents."

― (Bonabeau et al, 1999)

From this sense, we got the terminology ***Swarm Robotics***; this is a relatively new approach that focuses on controlling large-scale multi-robot systems, which consists of large numbers of simple physical robots. This approach emerged from the field of artificial swarm intelligence, as well as the biological studies of insects, ants and other fields in nature where swarm behavior occurs.

1.2 Motivation

SI (Swarm Intelligence) systems are typically made up of a population of simple agents interacting locally with one another and with their environment. The inspiration often comes from nature, especially biological systems.
The agents follow very simple rules, and although there is no centralized control structure dictating how individual agents should behave, localize and interactions between such agents lead to the emergence of "intelligent" global behavior which was unknown to the individual agent

Swarm robotics is a relatively new field that focuses on controlling large-scale homogeneous multi-robot systems.
These systems are used to develop useful macro-level behaviors while being made of modules that are very simple in design and compact in size.
These properties allow robot swarms to reach populations ranging from dozen of modules to hundreds of modules.
The theme of simplicity and elegance resonates throughout swarm robotics research in both the designs of the robots as well as the algorithms that are devised for these systems.

Robotic swarms have several advantages over their more complex individual robot counterparts and that's the result of using many robots instead of just one, this is made possible by the simple design of the robot modules because they are often less expensive and easier to build.

Advantages of Multi-Robot over Single Robot:

When comparing the capabilities of a robot swarm to the capabilities of an individual robot, it is best to view the swarm as an individual entity performing complex behaviors at the macro-level.

1-The first improvement

Robot swarms are able to cover more area than an individual robot.

This is analogous to distributed search algorithms that are able to cover different parts of a search space at once.

2-The second improvement

Robot swarms are fault tolerant because the swarm robotics algorithms do not require robots to depend on one another.

If a single module fails, the rest of the swarm can continue performing its actions as if that module never existed.

Meanwhile, an individual robot system may become worthless if there is a failure in a critical component. This type of robustness is an extremely important feature in complex or hostile environments.

3-Another feature of robot swarms

their effectiveness scales well with the number of members.

Adding more robots is all that has to be done to increase the effectiveness of a swarm.

The algorithms for swarms scale well and do not depend on the number of robots. On the other hand, it is not always clear how to improve the effectiveness of an individual robot system. Oftentimes improvements in hardware require additional software upgrades, which is not the case with swarms. These properties make multi-robot systems suitable for several application domains.

Simulation is getting information about how something will behave without actually testing it in real life. There are many advantages of using simulator.

1. Using simulations is generally cheaper and safer than conducting experiments with a prototype of the final product.
2. Simulations can often be even better than Real-life experiments, as they allow the free configuration of environment parameters found in the operational application field of the final product.

1.2.1 Applications

Such of these Robotics systems and approaches are mainly designed to be used in what can't be done easily and effectively by humans in different fields such as:

- Military Application.
- 3D Area mapping and reconstruction.
- Search and Rescue.
- Surveillance Systems.
- Mining.
- Traffic Monitoring

Applications of simulation such as:

- Supporting deep water operation of the US Navy
- Simulating the surface of neighbored planets in preparation of NASA missions.

1.3 Problem Definition

The proposed problem is to provide some enhancements to the previous co-swarm project (12-13).

After examining the project, we found that the project wasn't tested on a large scale of robots. Also, due to the current limited robots' capabilities, the restrictions on the environment increased.

More specifically, this affected the way of exploration which appears clearly in using the A* algorithm for navigation in a static environment, by restricting the motion of the robots to movements in 4 consecutive directions.

For these limitations, and because of the importance of solving the Area Exploration problem, we are going to work on enhancing the (12-13) Co-Swarm by building a simulated environment with enhanced robot's capabilities to speed up the performance of area exploration task.

1.4 Objectives

Building a simulated environment with enhanced robots' capabilities in the domain of mobile robot exploration, which is based on a trusted simulator for testing and improving the performance of (12- 13) Co-Swarm project.

CHAPTER 2: BACKGROUND

2.1 Area Exploration

In Robotics, the exploration problem deals with the use of a robot to maximize the knowledge over a particular area.
The exploration problem arises in mapping and search & rescue situations, where an environment might be dangerous and inaccessible to humans.

Today a large number of algorithms to solve the problem of autonomous exploration and mapping has been presented.
In general mobile robots need a map in order to operate in a particular environment.

The ability of robots to autonomously travel around an unknown environment gathering the necessary information to obtain a useful map for navigation is called autonomous exploration.

Current state of the art system in absence of global position information, simultaneous localization and mapping (SLAM) techniques are deployed to construct a map As the robots move to unexplored zones, these zones are included in the map.
Thus, the problem of autonomous exploration could be understood as a traveling salesman problem where the robot must plan the order to visit the remaining unexplored zones while minimizing the total traveled distance.
This way, there are greedy approaches where the robots move to the nearest or biggest unexplored zone, and more elaborated heuristic approaches.
However, this is dynamic process in which the set of places to visit is changing continuously.

Unfortunately, closed form solutions to these problems are computationally expensive, have poor scalability, and are impractical in real world applications.

In this sense, there are lots of techniques available in the literature that provides partial solutions to the problem.

In the wide spectrum of exploration methods, we can find techniques that focused in different aspects of the exploration, for instance trying to reduce the exploration time or increasing the map quality, in Co-Swarm we are focusing in reducing the exploration time rather than increasing the map quality.

2.2 Exploration Strategies

In 2012 Miguel Julia, publish a good research showing a good classification for different exploration strategies and introduced a good comparison between them, the classification was as follows:

- Classic Non-Coordinated strategies
- Classic Coordinated strategies
- Integrated Non-Coordinated strategies
- Integrated Coordinated strategies

Co-Swarm system relative to this classification lies **in the second category (Classic Coordinated Strategies)** as our system is depending on Frontier-Based Exploration such that the exploration can be speeded up by means of using multiple robots and the robots can share perceptions and build a common map of the environment, this global map is constructed independently in each robot in a distributed and more robust manner.

2.3 Swarm Robotics Survey

2.3.1 Multi-robot systems (MRS)

MRS's are a group of robots that are designed aiming to perform some collective behavior. The MRS's are gaining great interest because of the following:

- Resolving task complexity
- Increasing performance
- Reliability
- Simplicity in design

2.3.2 Swarm Robotics Characteristics

Swarm robotics is a relatively new field that focuses on controlling large-scale homogeneous multi-robot systems. These systems are used to develop useful macro-level behaviors while being made of modules that are very simple in design and compact in size. These properties allow robot swarms to reach populations ranging from a dozen modules to hundreds of modules. The theme of simplicity and elegance resonates throughout swarm robotics 16 research in both the designs of the robots as well as the algorithms that are devised for these systems.

Robotic swarms have several advantages over their more complex individual robot counterparts and are the results of using many robots instead of just one. This is made possible by the simple design of the robot modules because they are often less expensive and easier to build. When comparing the capabilities of a robot swarm to the capabilities of an individual robot, it is best to view the swarm as an individual entity performing complex behaviors at the macro-level.

The **first improvement** is an obvious one: robot swarms are able to cover more area than an individual robot. This is analogous to distributed search algorithms that are able to cover different parts of a search space at once.

The **second improvement** over individual robots is that swarm robots are fault tolerant because the swarm robotics algorithms do not require robots to depend on one another. If a single module fails, the rest of the swarm can continue performing its actions as if that module never existed. Meanwhile, an individual robot system may become worthless if there is a failure in a critical component. This type of robustness is an extremely important feature in complex or hostile environments. Another feature of robot swarms is that their effectiveness scales well with the number of members.

Adding more robots is all that is needed to be done to increase the effectiveness of a swarm. The algorithms for swarms scale well and do not depend on the number of robots. On the other hand, it is not always clear how to improve the effectiveness of an

individual robot system. Oftentimes improvements in hardware require additional software upgrades, which is not the case with swarms. These properties make multi-robot systems suitable for several application domains.

Algorithms for Swarm Robotics:

A variety of algorithms have been implemented to be run on swarms of robots. Some provide basic functionality, such as dispersion, while others demonstrate seemingly complex teamwork, such as chain formation. Although the algorithms all produce different emergent behavior, they all have many features in common. These features drive from the basic goals of swarm robotics discussed earlier and include:

- **Simple and elegant –** The robot controller that dictates the behavior of the individual robot is very simple. The behaviors of the individual robot can usually be represented as a state machine with few states and edges.

- **Scalable –** Swarm robotics algorithms are designed so that they work for any number of robots. Also, they are expected to scale well as new robots are added.

- **Decentralized –** The robots in a swarm are autonomous and do not follow any exterior commands. Although a member of a swarm can be directly and predictably influenced by the behavior of another, the choice is under its own accord. Being decentralized is often coupled with being scalable.

- **Usage of local interactions –** Local interactions are used over broadcasting messages in the majority of these algorithms. Even broadcasts are implemented as message hopping protocols. This ideal is a major factor in the scalability of the system.

2.3.3 Cooperative Swarm Robotic Systems

The three main blocks in any CO-SWARM system are the following

1. Coordination
2. Planning
3. Formation

Coordination

Coordination is a tricky subject for multi-agents. We have two issues to deal with first issue, what kind of communication is available?

- **Full** communication means that the agents can freely communicate and exchange Information.
- **Limited** means that the communication channel is either unstable or very limited
- **None** means that the agents are not able to communicate.

The second issue, is the agents distributed or centralized? This affects how they work together.

- **Distributed** means the agents are mainly governing themselves.
- **Centralized** means there is a leader that is giving orders or making plans for the other agents.

The rationale is that communication issues plague teams of either centralized or distributed alike. Though in theory, distributed teams are supposed to be more resilient to communication failure, but a team is no longer a team if they are not communicating anymore. At that point, the distributed robots are no different from single robot systems.

Planning

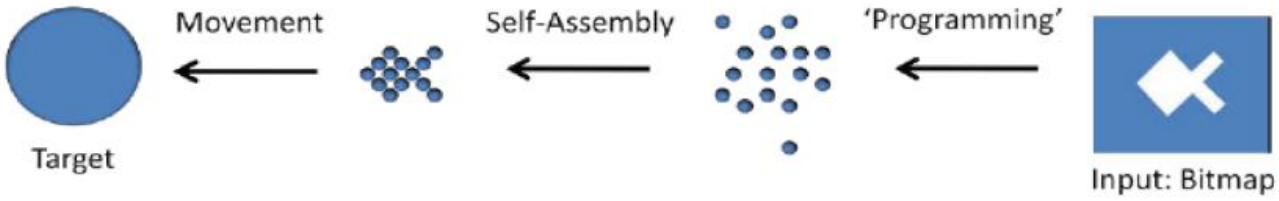
Like coordination, planning is a very broad term. The current literature generally uses the term to mean path or goal planning. We categorize this stage into 3 types:

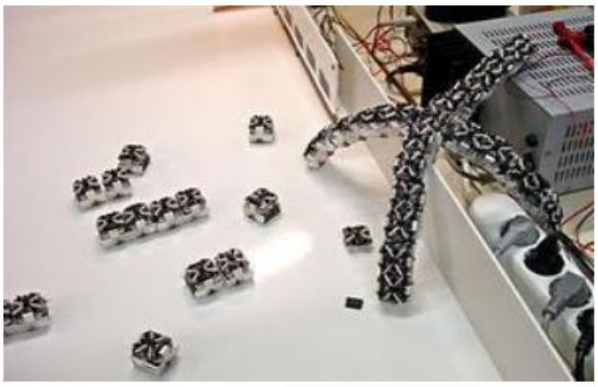
- Planning Using Constraint, Heuristic, or Hybrid Method.
- Planning Using Particle Physics Theory Treating robots as particles.
- Planning using neural networks

Formation

Formation suffers from similar communication as coordination. However, we will just treat formation as a subset of coordination and assume that the robots can communicate to the level of achieving formation.
We can categorize this stage in 2 main types:

1. Formation through Leader-Follower and Consensus Based Approaches.
2. Formation Frameworks and Architectures.





2.4 Self-Organizing Models

In field of swarm robotics, we have to define a model that describes the way that each robot behaves locally and the set of robots globally, as they are acting as a one unit, some questions must be asked: how they can communicate with each other? How they will take their decisions?! , what if more than one robot took the decision to navigate to the same point?! .. And so on. This is **the self Organizing**.

Self-Organization is the process where some form of global order or coordination arises out of the local interactions between the components of an initially disordered system. This process is not controlled by any Agent or subsystem inside or outside of the system The result of this process is that the organization is wholly decentralized over all of the components of the system, such of these systems are very robust and able to survive and self -repair substantial damage.

Examples of Self-Organizing systems:

- Swarming in groups of animals
- The way neural networks learn to recognize complex patterns
- Swarm Robotics

Self-Organization usually relies on 3 ingredients:

- Strong dynamical nonlinearity, often though not necessarily involving positive and negative feedback.
- Balance of exploitation and exploration.
- Multiple interactions.

We can notice all of these ingredients in Swarm Robotics Systems.
In swarm robotics, self-Organization is used to produce emergent behavior.

2.4.1 Mobility Models of Mobile Sensors

The mobility control strategy for mobile is a coordination strategy that addresses the inter-dependency management among these distributed agents. Based on this strategy, mobile sensors can move together in such a concert that achieves certain objectives. The mobility models lay under three categories, fully-coordinated model, fully random and emergent motion control model.
In Co-Swarm we chose the Emergent Motion Control Model and picked one of its algorithms which are Anti-Flocking.

2.4.2 Emergent Motion Control Model

Self-Organizing formation is an emergent process of making whole forms by local interactions of distributed simple autonomous entities without global information at all and without depending on initial position and orientation of the elements. These simple autonomous entities store local information and guiding rules needed for the colonial self-organization. The outcome is an adaptable complex system that can perform many tasks, learn and change itself accordingly.

Properties of Self-Organizing systems:

1. The main property in this model is the absence of central control
2. Adaptable due to emerging structures
3. High scalability
4. The most noticeable matter in all of these models, all of them are inspired by nature, and this self-organization can be easily noticed in fish schooling where each fish can utilize the pressure field created by the next fish when they cruise in a close group, also can be noticed clearly in bird flocking … etc

2.4.2.1 Anti-Flocking

Unlike many social animals that stay together as a group showing a collective behaviors such as flocking, there are some animals that survive due to different social behaviors called solitary behaviors, and that's what we are going to talk about in this post — the behavior of solitary animals [Anti-Flocking].

The Cooperative behavior of the solitary animals can be thought of high-level cooperative communication based on neighborhood or without explicit communication in most of the cases.

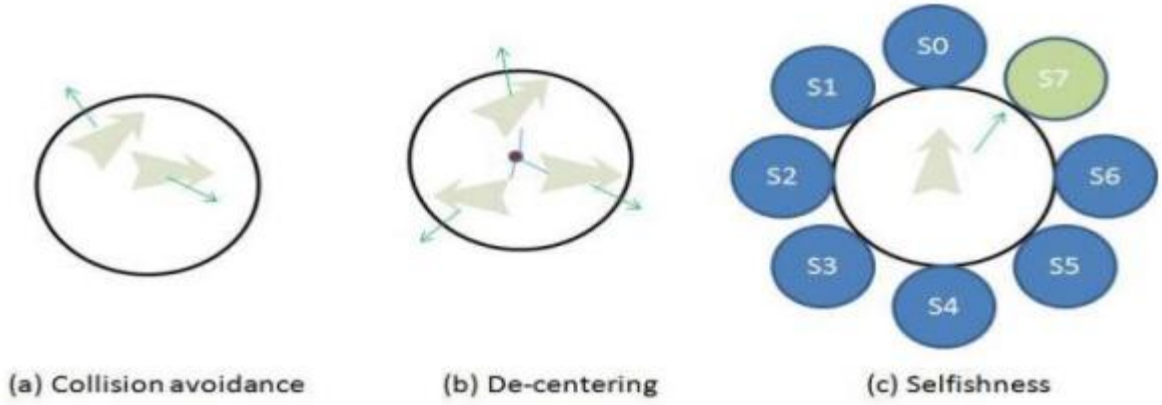
This cooperation pattern relies on achieving implicit communication between the cooperating agents via the environment, for example through marking trees by spraying urine and anal gland secretions.

Anti-flocking algorithm is inspired by this cooperative behavior that targets mainly **maximizing search coverage** and **minimizing overlapping**.

Anti-Flocking Algorithm

Rules of Anti-Flocking algorithm can be summarized as follows:

1. Collision Avoidance: Stay away from the nearest obstacle that's within safe distance.
2. De-Centering: attempt to move apart from its neighbors
3. Selfishness: if neither the above two situations happen, move to a directions which can maximize one's own interest.



(a) Collision avoidance (b) De-centering (c) Selfishness

- About the first rule — **Collision avoidance** — it's very easy to be understood and implemented, using mobile sensors the obstacles can be detected whether it's an outer obstacle or a neighbor unit, and the mentioned safe distance can be achieved through experiments or other more intelligent adaptive algorithms .

- The second rule is — **decentralizing** — or we can name it dispersing which is the method of scattering the available sensing resources over the whole area of interest. With setting the neighborhood radius, which is usually decided by the communication range of mobile sensors, each mobile sensor moves away from the center of its neighbors.

- About the last rule — **Selfishness** — is applied when neither obstacles nor neighbors appears, in this case, the mobile sensor moves to the direction that maximizes one's interest.

Algorithm pseudo code

```
Require: AOI Map, Neighborhood Radius, Gain Growth Rate.
1: Initialization environment:
2: Setup specfications of mobile sensors
3: Setup gain model of AOI
4: repeat
5:     Sensing obstacles
6:    Sensing neighbors
7:    if obstacles exist then
8:         Invoke obstacle avoidance routine
9:    else if neighbors  != null then
10:        Calculate centroid of neighbors
11:        Move to the opposite direction to the centroid
12:   else
13:        Calculate gains of surrounding directions
14:        Move to the direction with maximum gain
15:   end if
16: Update gain of AOI
17: until Command of Task-End
```

2.5 Frontier-Based Exploration

The central idea behind frontier-based exploration is to gain the most new information about the world, move to the boundary between open space and uncharted territory.

Frontiers are regions on the boundary between open space and unexplored space. When a robot moves to a frontier, it can see into unexplored space and add the new information to its map. As a result, the mapped territory expands, pushing back the boundary between the known and the unknown. By moving to successive frontiers, the robot can constantly increase its knowledge of the world.

1. We use Occupancy grid as their spatial representation .Occupancy grids are Cartesian grids containing cell which stores the probability that the corresponding region in space is occupied. Initially all of the cells are set to the prior probability of occupancy, which is a rough estimate of the overall probability that any given location will be occupied. Occupancy grids have the advantage of being able to fuse information from different types of sensors.

2. The standard occupancy grid formulation assumes that each sensor reading is independent of every other sensor reading. In reality, this is not the case and we have to take advantage of this issue. Using some of the sensors fusion techniques, we can construct the map by using some cheap not accurate sensors instead of using an expensive sensor. As before starting any robotic system, we have to consider the cost issue.

3. After an occupancy grid has been constructed, each cell in the grid is classified by comparing its occupancy probability to the initial (prior) probability assigned to all cells. A value of 0.5 was used in all of the experiments described in this paper. In general, this does not need to be an accurate estimate of the actual amount of occupied space. A prior probability of 0.5 works fine in environments where only a small fraction of the total space is occupied, as well as in far more cluttered environments

4. BFS algorithm is used to find the boundaries between open space and unknown space. Any open cell adjacent to an unknown cell is labeled a frontier edge cell. Adjacent edge cells are grouped into frontier regions. Any frontier region above a certain minimum size (roughly the size of the robot) is considered a frontier.

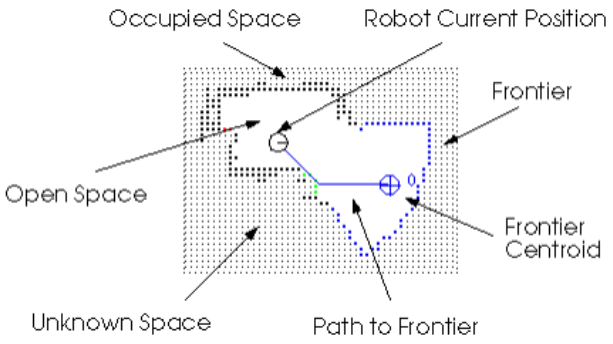
5. Once frontiers have been detected within a particular occupancy grid, the robot attempts to navigate to the nearest accessible, unvisited frontier. The path planner uses any graph search algorithm like A* & D*Lite on the occupancy grid, starting at the robot's current cell and attempting to take the shortest obstacle-free path to the cell containing the goal location.

6. While the robot moves toward its destination, reactive obstacle avoidance behaviors prevent collisions with any obstacle not present while the occupancy grid was constructed. If an obstacle was found which wasn't detected before, In case of A* algorithm a re-planning for the path is carried out while in the case of D*Lite some updates is carried out on the calculated path to generate a new path from the new current position to the goal position.

7. When the robot reached the destination point, it sweeps the area again with sensors and adds new information to the occupancy map then detects new frontiers in the updated grid and Finally navigate to the nearest accessible and unvisited frontier.

Each cell is placed into one of three classes:

- **open:** occupancy probability < prior probability
- **unknown:** occupancy probability = prior probability
- **occupied:** occupancy probability > prior probability

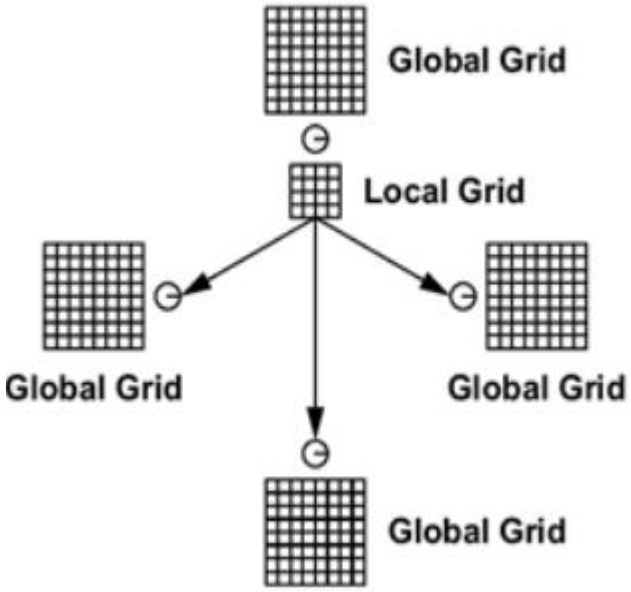


2.5.1 Multi-Robot Exploration Extended

Robots share perceptual information, but maintain separate global maps, and make independent decisions about where to explore. This approach enables robots to make use of information from other robots to explore more effectively, but it also allows the team to be robust to the loss of individual robots.

It is the same idea of the previously mentioned single exploration technique but with some extra steps as the following:

1. Each robot stores the local grids received from other robots.
   When a robot arrives at a new frontier, it integrates these local grids with its global map, along with the new local grid it constructs at the frontier.

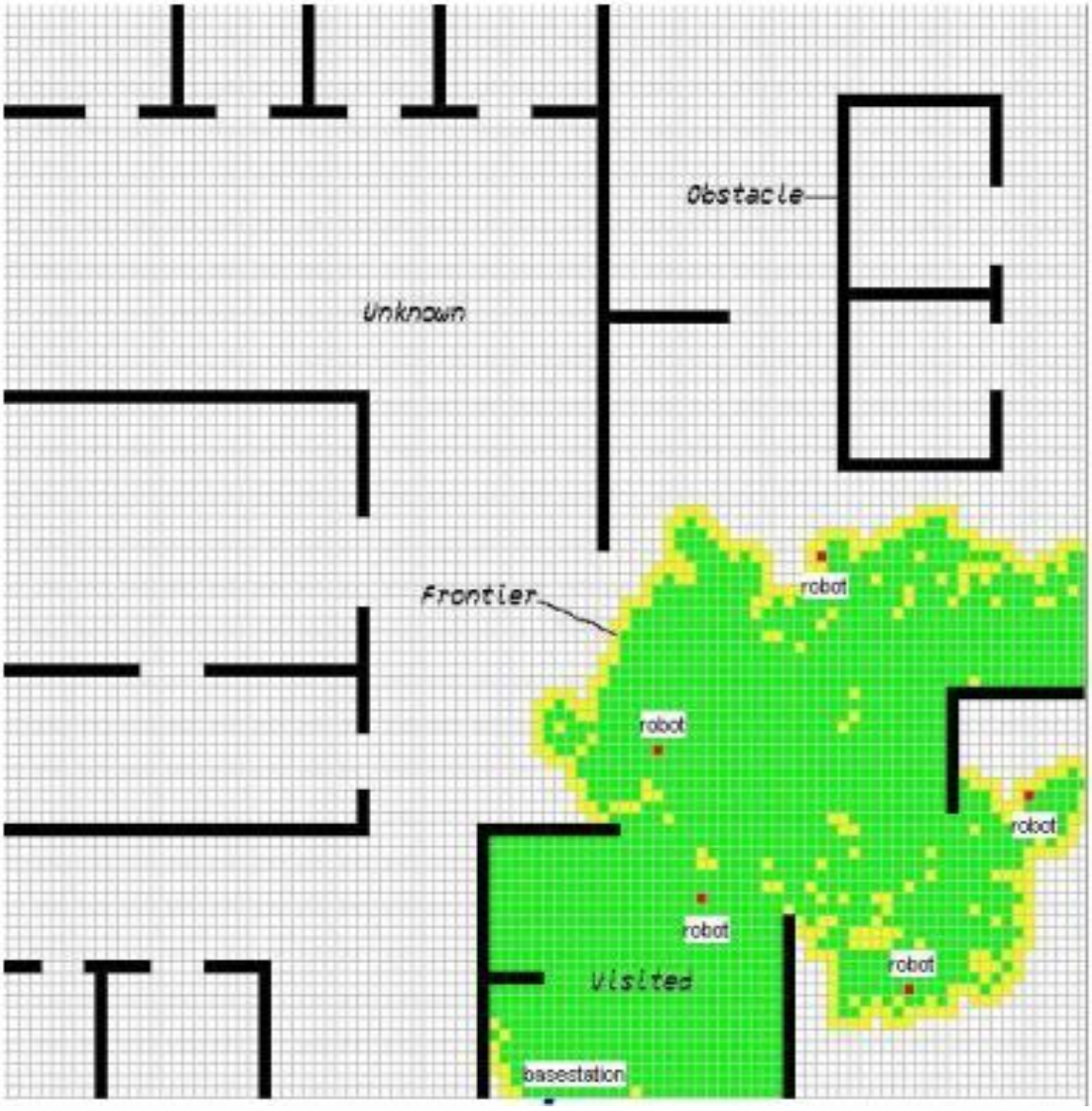


2. Two occupancy grids can be integrated in a straightforward way. A log odds representation is used. Using this representation, independent probabilities can be combined using additions rather than multiplications. The log odds probabilities of each pair of corresponding cells are added and the sum is stored in the corresponding cell of a new grid. These probabilities are normalized so that the log prior probability is equal to zero, so that cells with no information in either grid remain equal to the prior probability in the new grid.

Advantages of this Algorithm in Multi-Exploration (Swarm Features)

- Cooperative and decentralized.
- All of the information obtained by any robot is available to each robot. This allows robots to use the data from other robots to determine where to navigate.
- Based on this information, a robot can determine which areas have already been explored by other robots, and then choose to explore another region.
- A robot can also discover that a frontier detected by another robot is nearby, and decide to investigate it.

The next figure illustrates the idea of the frontier-based exploration in MRS



2.6 Previous Work

A Behavior-Based Strategy for Single and Multi-Robot Autonomous Exploration

Sensors 2012, 13 July 2012

In this paper, they consider the problem of autonomous exploration of unknown environments with single and multiple robots. This is a challenging task, with several potential applications. We propose a simple yet effective approach that combines a behavior-based navigation with an efficient data structure to store previously visited regions. This allows robots to safely navigate, disperse and efficiently explore the environment. A series of experiments performed using a realistic robotic simulator and a real tested scenario demonstrate that our technique effectively distributes the robots over the environment and allows them to quickly accomplish their mission in large open spaces, narrow cluttered environments, dead-end corridors, as well as rooms with minimum exits.

Navigation Strategies for Exploring Indoor Environments (NBV)

The International Journal of Robotics Research
October 1, 2002 vol. 21 no. 10-11 829-848

In this paper, they investigate safe and efficient map -building strategies for a mobile robot with imperfect control and sensing. In the implementation, a robot equipped with a range sensor builds a polygonal map (layout) of a previously unknown indoor environment. The robot explores the environment and builds the map concurrently by patching together the local models acquired by the sensor into a global map. A well-studied and related problem is the simultaneous localization and mapping (SLAM) problem, where the goal is to integrate the information collected during navigation into the most accurate map possible. However, SLAM does not address the sensor-placement portion of the map-building task.

That is, given the map built so far, where should the robot go next?
This is the main question addressed in this paper. Concretely, an algorithm is proposed to guide the robot through a series of 'good' positions, where 'good' refers to the expected amount and quality of the information that will be revealed at each new location. This is similar to the next-best-view (NBV) problem studied in computer vision and graphics. However, in mobile robotics the problem is complicated by several issues, two of which are particularly crucial. One is to achieve safe navigation despite an incomplete knowledge of the environment and sensor limitations (e.g., in range and incidence). The other issue is the need to ensure sufficient overlap between each new local model and the current map, in order to allow registration of successive views under positioning uncertainties inherent to mobile robots. To address both issues in a coherent framework, in this paper we introduce the concept of a safe region, defined as the largest region that is guaranteed to be free of obstacles given the sensor readings made so far. The construction of a safe region takes sensor limitations into account. In this paper we also describe an NBV algorithm that uses the safe region concept to select the next robot position at each step. The new position is chosen within the safe region in order to maximize the expected gain of information under the constraint that the local model at this new position must have a minimal overlap with the current global map. In the future, NBV and SLAM algorithms should reinforce each other. While a SLAM algorithm builds a map by making the best use of the available sensory data, an NBV algorithm, such as that proposed here, guides the navigation of the robot through positions selected to provide the best sensory inputs .

Comparative study of Algorithms for frontier based area exploration and Slam for mobile robots

International Journal of Computer Applications
(0975 – 8887) Volume 77 – No.8, September 2013

In this paper a survey of existing approaches in frontier based area exploration and various SLAM algorithms which can be useful for the process of area exploration are discussed.

A comparison between 5 different area exploration algorithms was discussed briefly. The differences were with respect to the environment representation, basic algorithm and path planning algorithm.

This paper shows that till September 2013 D*Lite wasn't used at all as a path planning algorithm with the Frontier based approach.

You can find the table of the main comparison points displayed below.

Table 1. Comparison between different area exploration algorithms

	BGS and MGS	Circle portioning	Pruning	MinPos	Fast Frontier
Environment representation	Hexagonal OG	Circle partitioning and OG	OG	OG	Contour
Basic Algorithm	BGD and MGS	Circle partitioning	K-means clustering, Hungarian Method and pruning technique	MinPos	WFD and FFD
Path planning algorithm	Using goal seeking index	EA*	A*	WPA	BFS

Co-Swarm 2012-2013

Co-Swarm'13 is an **intelligent robotics system** based on the Swarm Robotics techniques where a team of robots is exploring an **unknown indoor environment**.
Robots share perceptual information, but maintain separate global maps and make independent decisions about where to explore.

This approach enables robots to make use of information from other robots to explore more effectively, but it also allows the team to be robust to the loss of individual robots.

The system based mainly on **frontier-based approach** to achieve the exploration task and on shortest path algorithms for navigation on **static environment** like **A* Algorithm**.

The enhancement that made to enhance the exploration time was in controlling the emergent behavior of the swarm by implementing one of mobility sensing models which is **Anti-Flocking approach**.

The system was tested on real environment with a navigation area and **2 ground vehicles** and it achieved good results against simple scenarios , but for testing complex scenario we need large number of robots and this not applicable in real life and for this we decided to rebuild the system on a simulated environment.

CHAPTER 3:
ROBOT OPERATING SYSTEMS (ROS)

3.1 Introduction

Robot Operating System (ROS) is a framework that is widely used in robotics by hundreds of research groups and companies in the robotics industry. The philosophy is to make a piece of software that could work in other robots by making little changes in the code. What we get with this idea is to create functionalities that can be shared and used in other robots without much effort so that we do not reinvent the wheel.

ROS was originally developed in 2007 by the Stanford Artificial Intelligence Laboratory (SAIL) with the support of the Stanford AI Robot project. As of 2008, development continues primarily at Willow Garage, a robotics research institute, with more than 20 institutions collaborating within a federated development model.

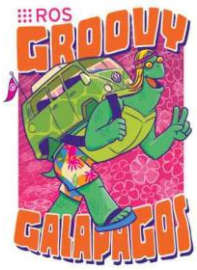
A lot of research institutions have started to develop projects in ROS by adding hardware and sharing their code samples. Also, the companies have started to adapt their products to be used in ROS.

The goal of ROS is not to be a framework with the most features. Instead, the primary goal of ROS is to support code reuse in robotics research and development

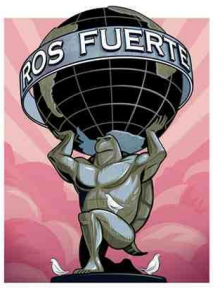
3.1.1 ROS Software Distributions

ROS Hydro Medusa (September, 2013)

ROS Groovy (December, 2012)

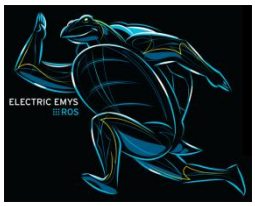


ROS Fuertes (April, 2012)



ROS Diamondback (March, 2011)

ROS Electric (August, 2011)



ROS C Turtle (August, 2010)

ROS Box Turtle (March, 2010)

It is possible to have two or more versions of ROS installed on your computer. We are using Fuertes because it is a stable version.

3.1.2 ROS Supported Platforms

In the following image, you can see some fully supported platforms.

Erle-HexaCopter : is a "heavy lifter " It's capable of the different flight modes and ideal for outdoor operations. It has been designed for an extended flight time and it can carry a takeoff weight of about 3-4 kilograms.
is powered by Erle-brain Linux autopilot which includes ROS and communicates with the flight software using the Mavros ROS package for communication.

Maggie robot: is a social robot platform built by the Social Robots Group from Robotics Lab at University Carlos III of Madrid. The software system is written entirely in ROS. As such, all Maggie capabilities are available via ROS interfaces.

Normally, these platforms are published with a lot of code, examples, and simulators to permit the developers to start their work easily.

The sensors and actuators used in robotics have also been adapted to be used with ROS. Every day an increasing number of devices are supported by this framework.

ROS provides standard operating system facilities such as:

- Hardware abstraction.
- Low-level device control.
- Implementation of commonly used functionalities.
- Message passing between processes, and package management

Operating Systems:

ROS currently only runs on Unix-based platforms. Software for ROS is primarily tested on Ubuntu and Mac OS X systems, though the ROS community has been contributing support for Fedora, Gentoo, Arch Linux and other Linux platforms. While a port to Microsoft Windows for ROS is possible, it has not yet been fully explored.

3.1.3 ROS Package

The *-ros-pkg package is a community repository for developing high-level libraries easily. Many of the capabilities frequently associated with ROS, such as the navigation library and the RVIZ visualize, are developed in this repository.

These libraries give a powerful set of tools to work with ROS easily, knowing what is happening every time.

3.1.4 Programming with ROS

ROS is language-independent. At this time, three main libraries have been defined for ROS, making it possible to program ROS in Python, Lisp or C++. In addition to these three libraries, two experimental libraries are offered, making it possible to program ROS in Java or Lua.

3.1.5 ROS License

ROS is released under the terms of the BSD (Berkeley Software Distribution) license and is an open source software. It is free for commercial and research use.
The * -rospkg contributed packages are licensed under a variety of open source licenses.

ROS promotes code reutilization so that the robotics developers and scientists do not have to reinvent the wheel all the time. With ROS, you can do this and more. You can take the code from the repositories, improve it, and share it again.

3.2 ROS Architecture

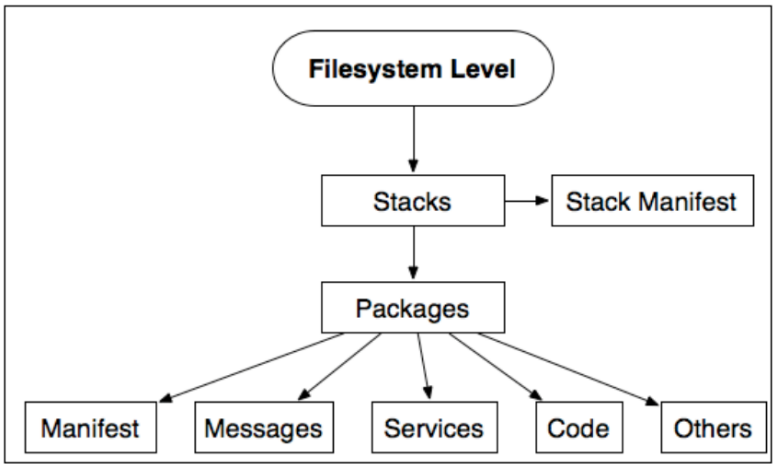
The ROS architecture has been designed and divided into three sections or levels of concepts:

1. **The File system level:** In this level, a group of concepts is used to explain how ROS is internally formed, the folder structure, and the minimal files that it needs to work.
    - Packages
    - Manifests
    - Stacks
    - Stack manifests
    - Message
    - Service

2. **The Computation Graph level** where communication between processes and systems happen. In this section, we will see all the concepts and systems that ROS has to set up systems, to handle all the processes, to communicate with more than a single computer, and so on.
    - Nods
    - Topics
    - Service
    - Bags
    - Parameter Server
    - ROS Master

3. **The Community level** where we will explain the tools and concepts to share knowledge, algorithms, and code from any developer. This level is important because ROS can grow quickly with great support from the community.

3.2.1 ROS File system Level



Similar to any operating system, ROS is divided into folders, and these folders have some files that describe their functionalities.

• **Packages:** Packages form the atomic level of ROS. A package has the minimum structure and content to create a program within ROS. It may have ROS runtime processes (nodes), configuration files, and so on.

• **Manifests:** Manifests provide information about a package, license information, dependencies, compiler flags, and so on. Manifests are managed with a file called manifests.xml.

• **Stacks:** a collection of packages. It offers a set of functionalities such as navigation, positioning, etc. A stack is a directory containing package directories plus a configuration file called stack.xml. In ROS, there exists a lot of these stacks with different uses.

• **Stack manifests:** Stack manifests (stack.xml) provide data about a stack, including its license information and its dependencies on other stacks.

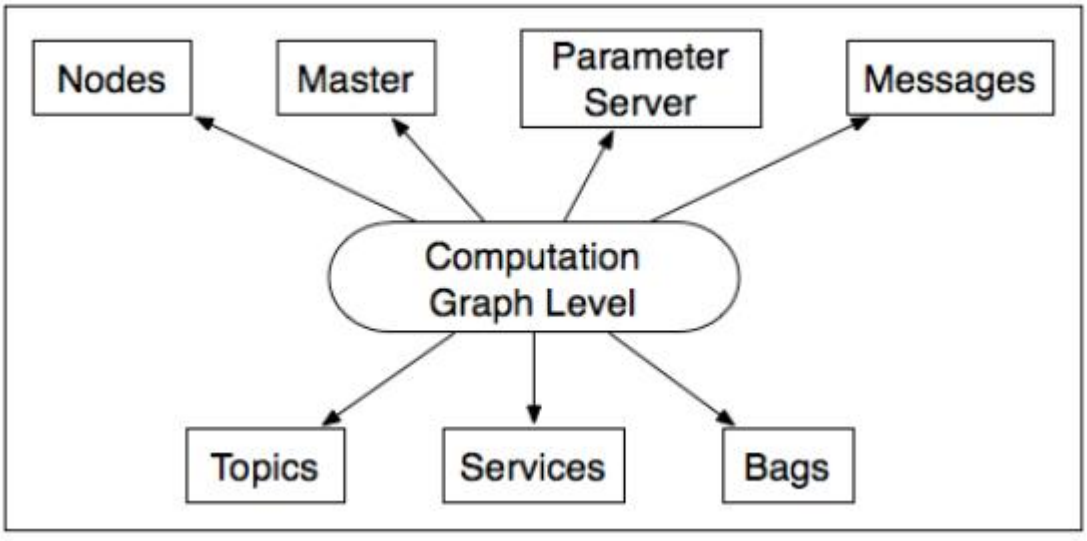
• **Message (msg) types**: A message is the information that a process sends to other processes. ROS has a lot of standard types of messages. Message descriptions are stored in my_package / msg / MyMessageType.msg.

• **Service (srv) types:** Service descriptions, it defines the request and response data structures for services in ROS and stored in my_package / srv/ MyServiceType.srv.

3.2.2 ROS Computation Graph Level

The basic principle of a robot operating system is to run a great number of executables in parallel that need to be able to exchange data synchronously or asynchronously. For example, a robotics OS needs to query robot sensors at a set frequency (ultrasound or infra-red distance sensor, pressure sensor, temperature sensor, gyroscope, accelerometer, cameras, microphone, etc.), retrieve this data, process it , pass it to processing algorithms (speech processing, artificial vision,
SLAM – simultaneous localization and mapping, etc.) and lastly control the motors in return. This whole process is carried out continuously and in parallel.

The concepts brought together in ROS under the name of "ROS Computation Graph", enabling these objectives to be reached, are described below. These are concepts used by the system as it is running, whereas the ROS File System described in the previous section is a static concept.



Nodes

In ROS, Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale; a robot control system usually comprises many nodes. For example, one node controls a laser range-finder, one node controls the wheel motors, one node performs localization, one node performs path planning, one Node provides a graphical view of the system, and so on.

A node is an instance of an executable. It may equate to a sensor, motor, processing or monitoring algorithm, and so on. Every node that starts running declares itself to the **Master**.

Master

The **Master** is a node declaration and registration service, which makes it possible for nodes to find each other and exchange data. If you don't have it in your system, you can't communicate with nodes, services, messages, and others. But it is possible to have it in a computer where nodes work in other computers.

The Master is implemented via XMLRPC (where XML-RPC is a remote procedure call (RPC) protocol which uses XML to encode its calls and HTTP as a transport mechanism).

The Master includes a heavily-used component called the **Parameter Serve**r, also implemented in the form of XMLRPC, and which is, as the name implies, a kind of centralized database within which nodes can store data and share system-wide parameters.

Topics

Each message must have a name to be routed by the ROS network. When a node is sending data, we say that the node is publishing a topic. Nodes can receive topics from other nodes simply by subscribing to the topic.

A topic is a data transport system based on a subscribe/publish system. One or more nodes are able to publish data to a topic, and one or more nodes can read data on that topic.

Data is exchanged asynchronously by means of a topic and synchronously via a service.

Messages

Nodes communicate with each other through messages. A message is a compound data structure. ROS has many types of messages, and you can also develop your own type of message using standard messages.

A message comprises a combination of primitive types (character strings, Booleans, integers, floating point, etc.).

For example, a node representing a robot servo motor will certainly publish its status on a topic with a message containing an integer representing the motor's position, a floating point for its temperature, another floating point for its speed, and so on.

The message description is stored in *package_name/msg/myMessageType.msg* , This file describes the message structure

Services

A topic is an asynchronous communication method used for many-to-many communication. But when you need a request or an answer from a node, you can't do it with topics. The idea is similar to that of a procedure call.

Services must have a unique name. When a node has a service, all the nodes can communicate with it.

The service description is stored in *package_name/srv/myServiceType.srv*. This file describes the data structures for requests and responses.

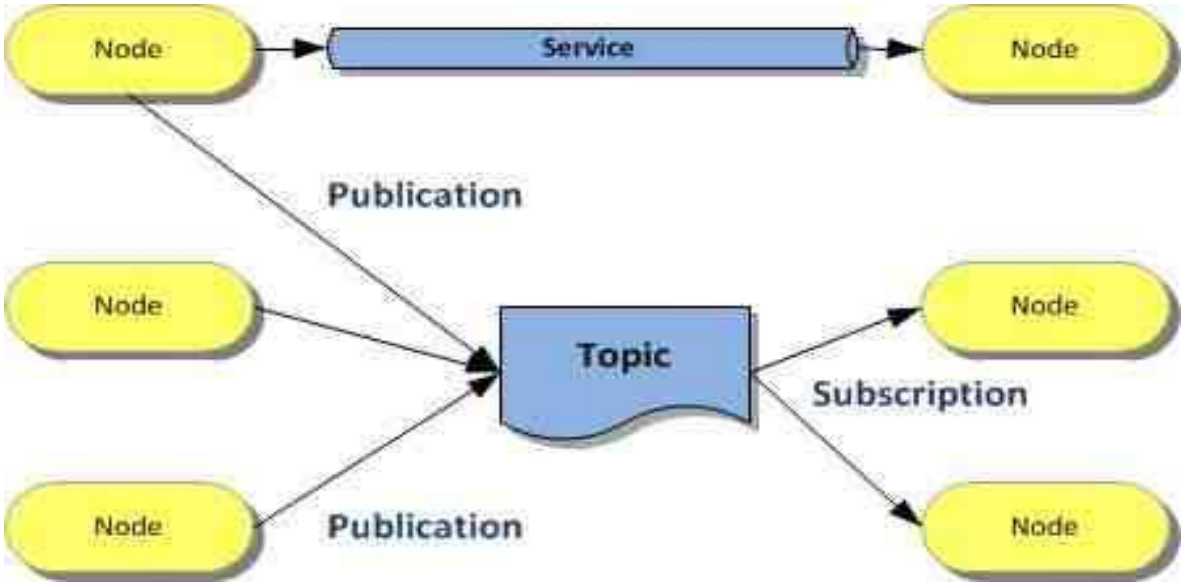
Bags

Bags are formats for storing and playing back message data. This mechanism makes it possible, for example, to collect data measured by sensors and subsequently plays it back as many times as desired to simulate real data. It is also a very useful system for debugging a system after the event. Bags are useful when working with complex robots

The rxbag tool can be used to display data saved in bag files in graphical form

Parameter Server

The Parameter Server allows data to be stored by key in a central location. It is currently part of the Master.



3.2.3 ROS Community Level

The ROS Community level concepts are ROS resources that enable separate Communities to share software and knowledge.

These resources include:

- **Repositories:** ROS relies on a network of code repositories, where different institutions can develop and release their own robot software components.
- **The ROS Wiki:** The ROS Wiki is the main forum for documenting information about ROS. Anyone can sign up for an account and contribute their own documentation, provide corrections or updates, write tutorials, and more.
- **Mailing lists:** The ROS user-mailing list is the primary communication channel about new updates to ROS as well as a forum to ask questions about the ROS software.

3.3 Simulation

Programming directly on a real robot gives us good feedback and it is more impressive than simulations, but not everybody has possible access to real robots. For this reason, we have programs that simulate the physical world.

ROS support many simulators.

- Simulation Stage (2d simulator)
- Simulation Gazebo (3d simulator)

Why Gazebo?

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, and perform regression testing using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. At your fingertips is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community

History

Gazebo development began in the fall of 2002 at the University of Southern California. The concept of a high-fidelity simulator stemmed from the need to simulate robots in outdoor environments under various conditions. As a complementary simulator to Stage, the name Gazebo was chosen as the closest structure to an outdoor stage. The name has stuck despite the fact that most users of Gazebo simulate indoor environments.

Open Source Robotics Foundation (OSRF) continues development of Gazebo with support from a diverse and active community.

3.4 Visualization

Visualizing and logging sensor information is an important part in developing and debugging controllers.

RVIZ is 3D visualizer for displaying sensor data and state information from ROS. Using RVIZ, we can visualize Baxter's current configuration on a virtual model of the robot. We can also display live representations of sensor values coming over ROS Topics including camera data, infrared distance measurements, sonar data, and more.

Visualizing Sensor Data with RVIZ

- RVIZ is a powerful robot visualization tool. It provides a convenient GUI to visualize sensor data, robot models, environment maps, which is useful for developing and debugging your robot controllers.

- RVIZ is a ROS package that visualizes robots, point clouds, etc.

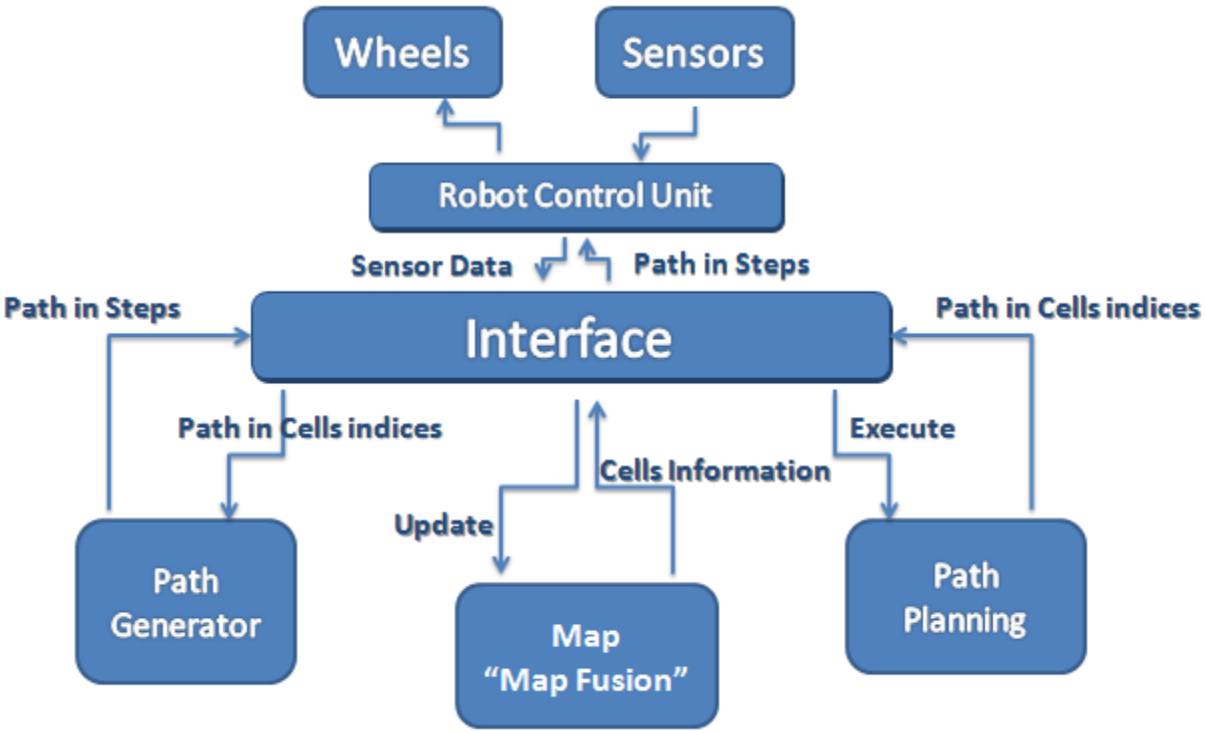
- RVIZ is perfect for figuring out what went wrong in a vision system. The list on the left has a check box for each item. You can show or hide any visual information instantly.

- RVIZ provides 3D visualization which you could navigate with just your mouse.

- You can have multiple vision processes showing vision data in the same RVIZ. You simply have to publish the point cloud or shape you want to show using the ROS publishing method. Visualizing is relatively painless once you get used to it.

CHAPTER 4: DESIGN

4.1 Overall System Architecture



ROS System Architecture

Path Planning:

It consists of the Logical Control of the Robot, Where it is responsible for generating the shortest path for the robot to navigate in.
it consists of Shortest Path Techniques for static environment like A* Or for Dynamic environment Like D* Lite , With Frontier Based approach enhanced With anti-Flocking technique were explained in chapter 2.

Path Generator:

It's responsible for generating the path which maps to the actual Robot motion to be send to the control unit to translate it to PID controller ticks then the robot will move.
It receives the generated path by the Path Planning module which is in cells and translates it to steps.

Map:

Consists of Local map of each robot and supporting methods for getting the appropriate information in which the planning module depends mainly on to get information about surrounding environment.

Interface:

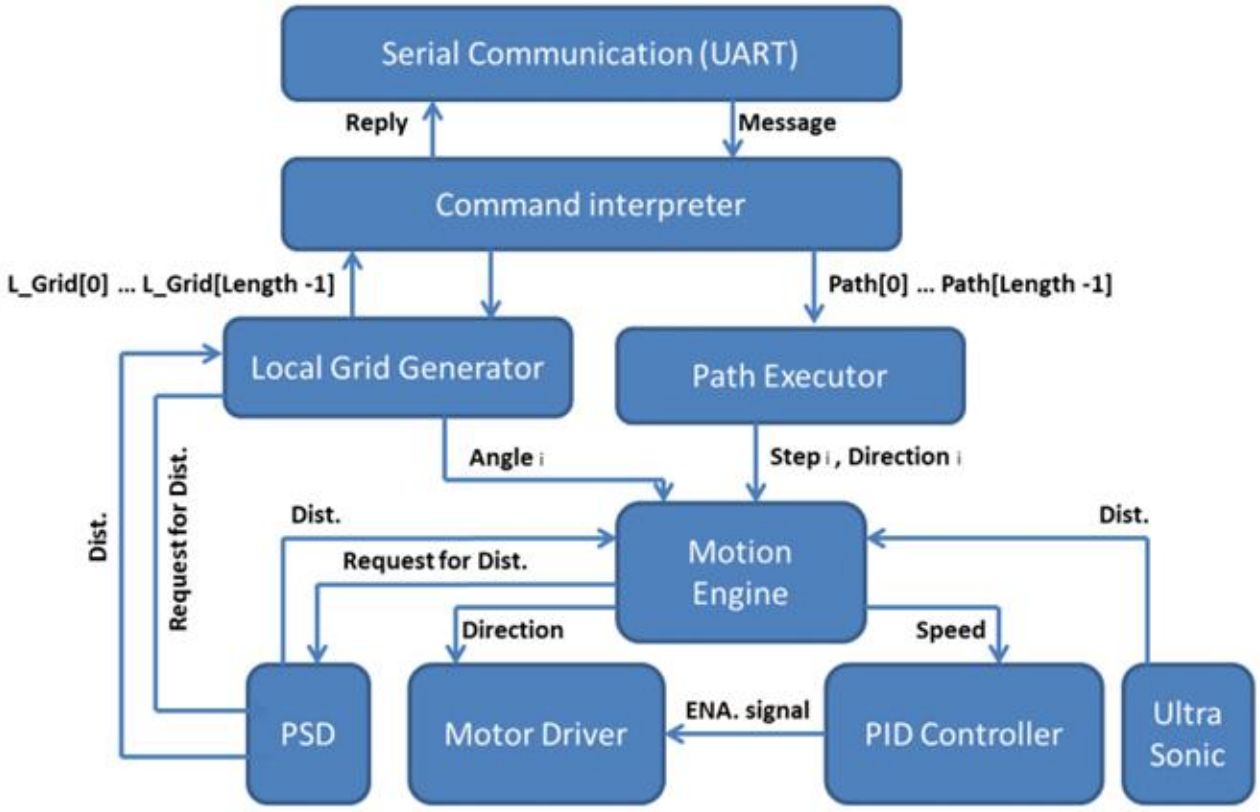
It's the main part which connects all previously mentioned parts together to communicate with motor control unit.
It send, receive and process data in a certain way depends on destination.

Robot Control Unit:

It's the part which gets sensor data and sends signals to the wheels.

4.2 Robo-Car Framework Design 2013

Next figure shows the block diagram for the framework.



Robot Frame work Block Diagram

Robot receives the commands and data from the brain serially via USART.

Serial communication module is responsible for reading the messages sent from AVR Microprocessor UART buffer and groups it then send it for the Command Interpreter Module.

Command Interpreter Module takes the message and decide whether the message data or command.

- If the Message is a **path**, which the robot should go through it, then Command Interpreter Module will forward the message for The Path Executor Module.

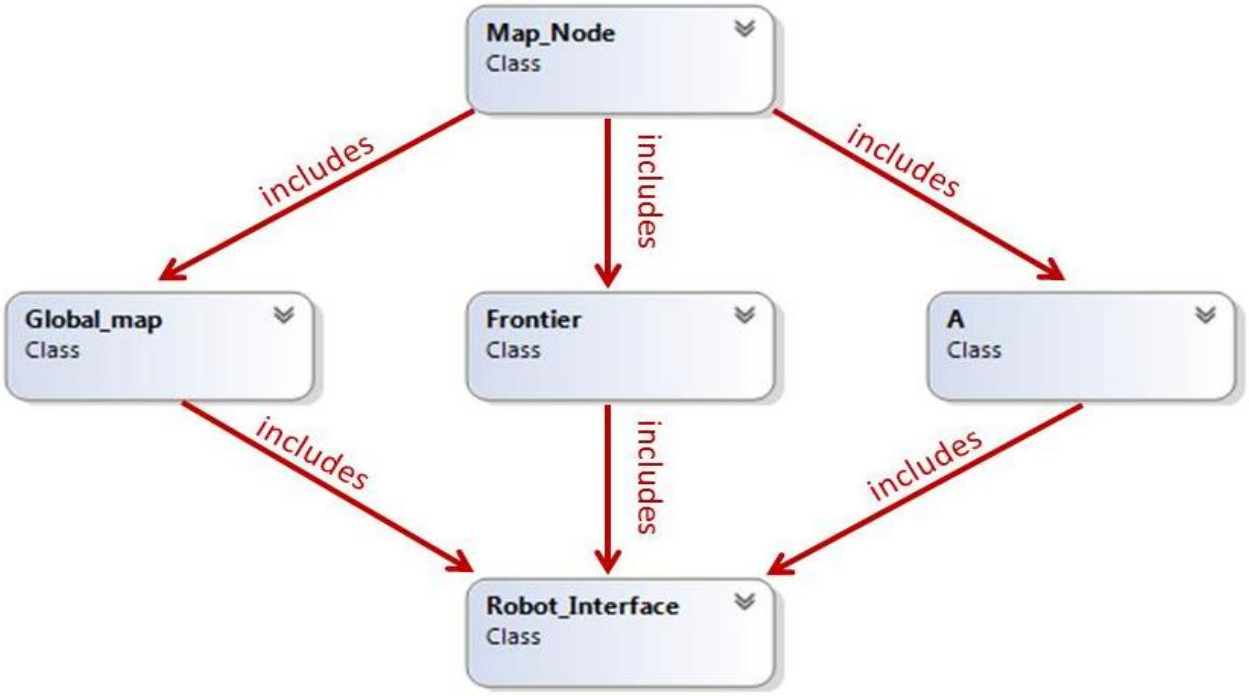
- And if the message is a command for the robot, then the command Interpreter will forward the message for the Local Grid Generator Module ,

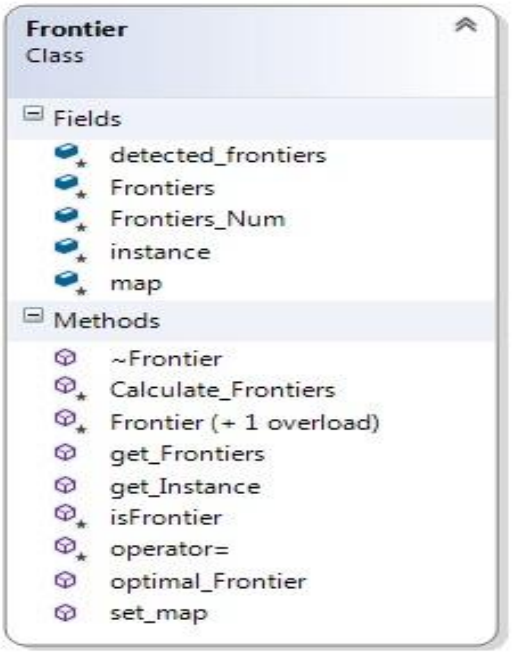
- else the Command Interpreter Module will parse the message and execute its function and send the suitable reply message for the brain . Path Executor Module receives the message which contains the path, and parses it and then starts to execute the path by sending the direction and distance for each step for the Motion Engine.

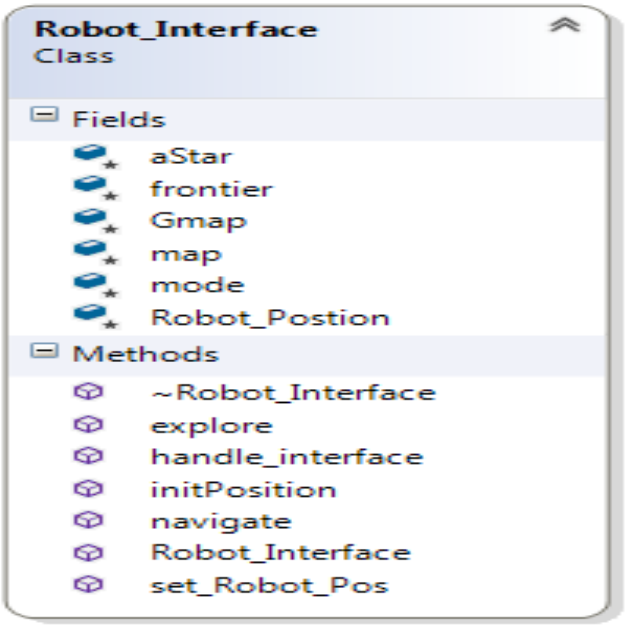
Motion Engine starts to perform the steps received from the path executor module, and send the suitable speed for the PID controller ,which its role is generating the pulse width modulation(PWM) needed from motor driver to be able to control the speed of the wheels in regularly way .
On the other side, Motion Engine send the direction control signal to motor driver which perform the direction received from path executor module (e.g. Forward, Backward).

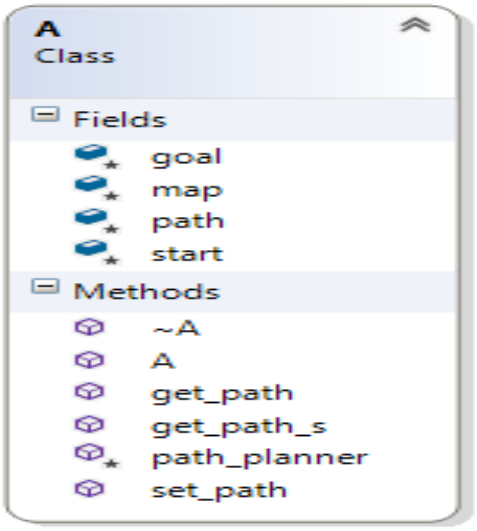
4.3 Motion Planning Design

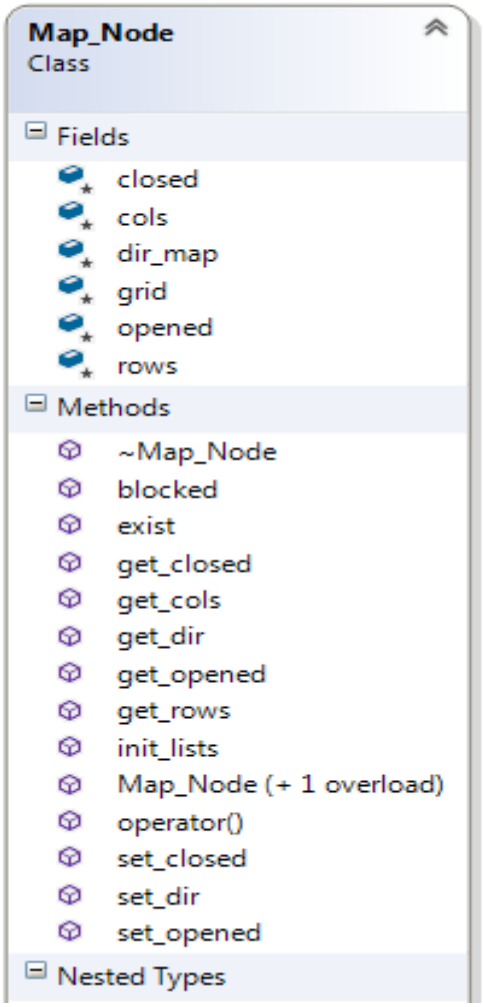
4.4.1 A Package*

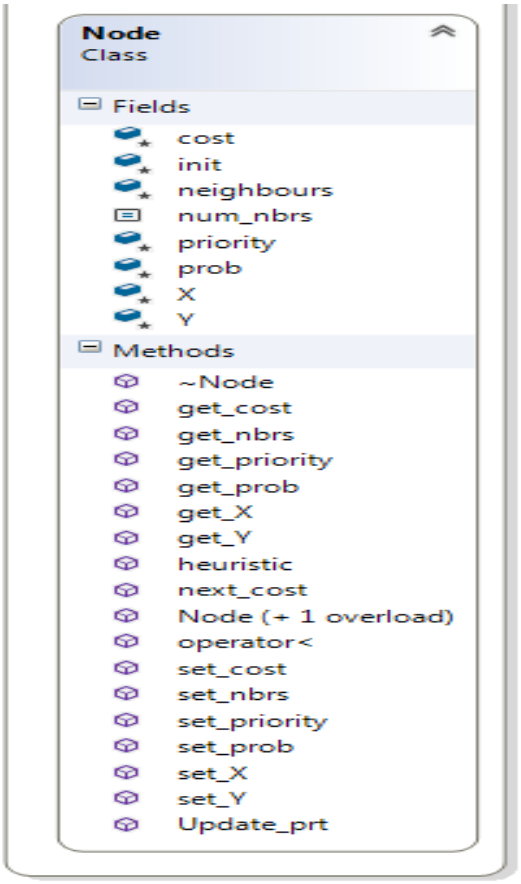




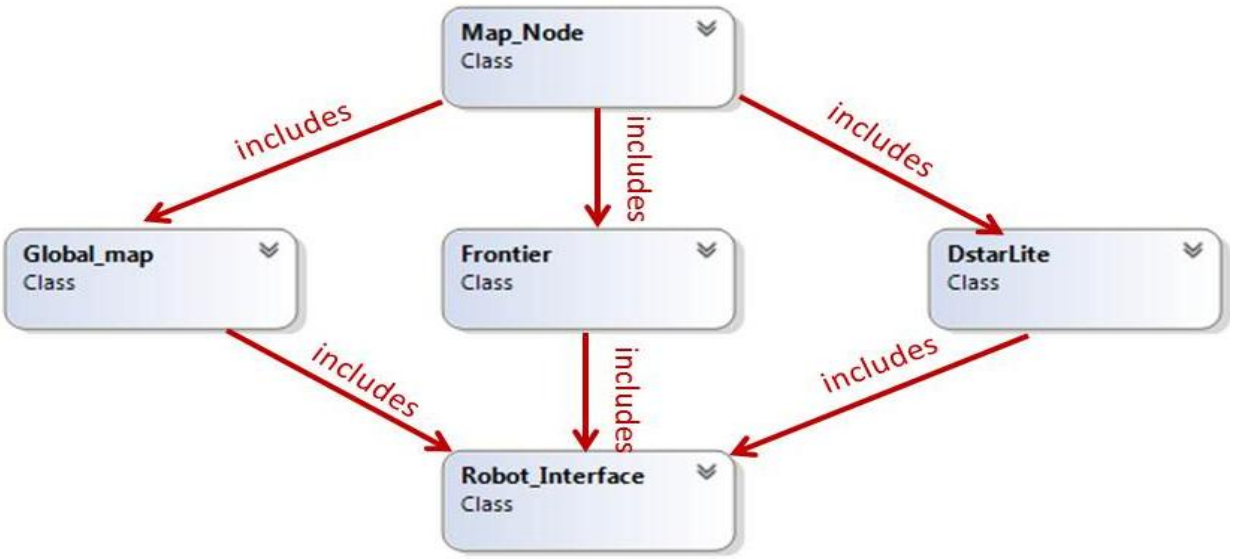

Robot_Interface
Class
Fields
aStar
frontier
Gmap
map
mode
Robot_Postion
Methods
~Robot_Interface
explore
handle_interface
initPosition
navigate
Robot_Interface
set_Robot_Pos

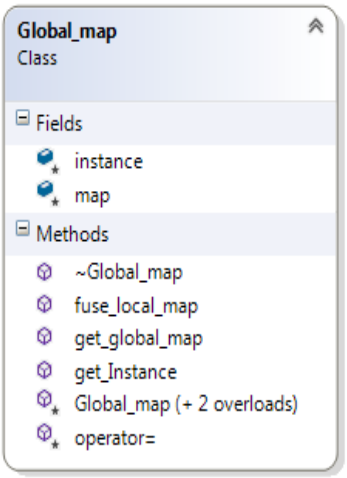

A
Class
Fields
goal
map
path
start
Methods
~A
A
get_path
get_path_s
path_planner
set_path

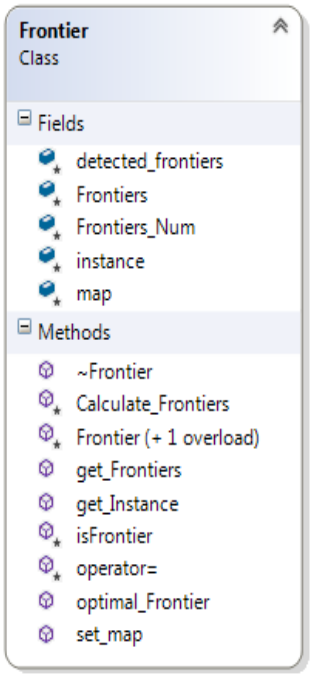

Map_Node
Class
Fields
closed
cols
dir_map
grid
opened
rows
Methods
~Map_Node
blocked
exist
get_closed
get_cols
get_dir
get_opened
get_rows
init_lists
Map_Node (+ 1 overload)
operator()
set_closed
set_dir
set_opened
Nested Types

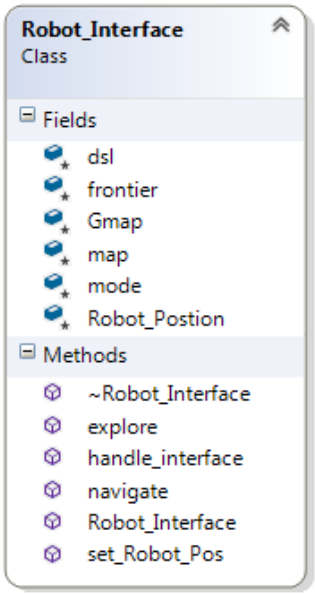

Node
Class
Fields
cost
init
neighbours
num_nbrs
priority
prob
X
Y
Methods
~Node
get_cost
get_nbrs
get_priority
get_prob
get_X
get_Y
heuristic
next_cost
Node (+ 1 overload)
operator<
set_cost
set_nbrs
set_priority
set_prob
set_X
set_Y
Update_prt

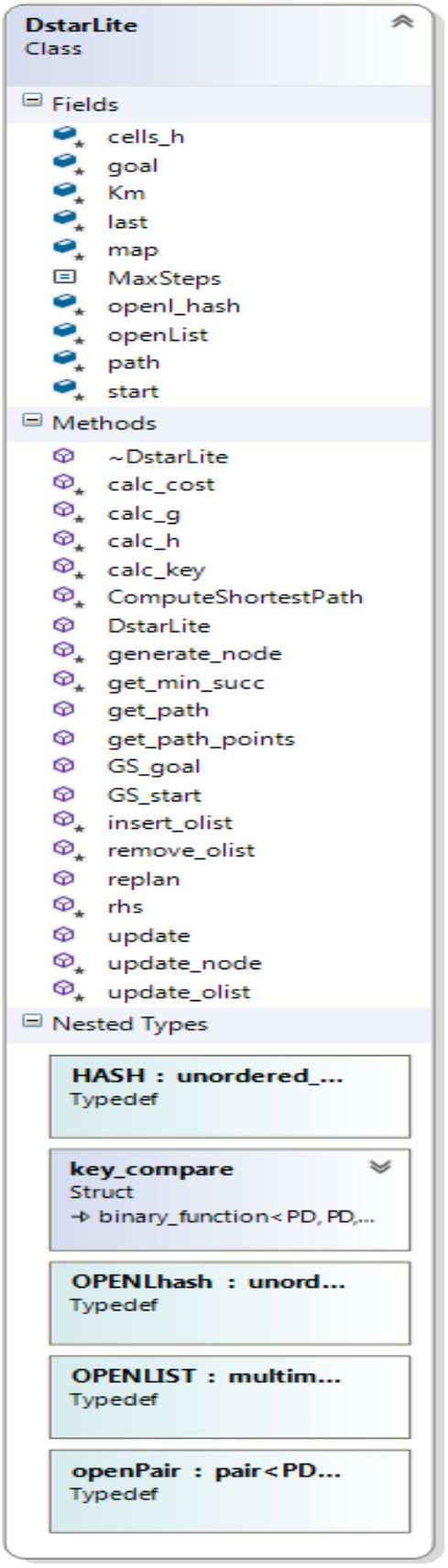
4.4.2 D*Lite Package

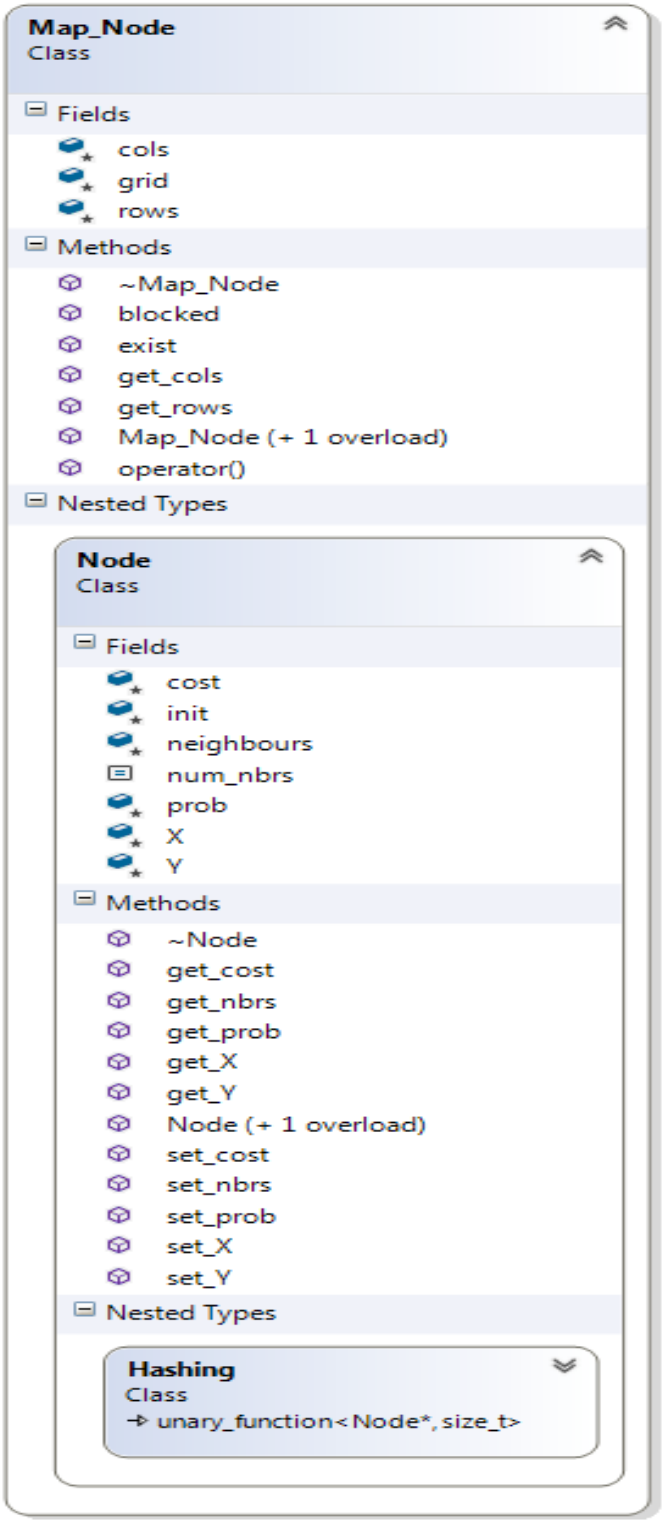









DstarLite
Class
Fields
cells_h
goal
Km
last
map
MaxSteps
openl_hash
openList
path
start
Methods
~DstarLite
calc_cost
calc_g
calc_h
calc_key
ComputeShortestPath
DstarLite
generate_node
get_min_succ
get_path
get_path_points
GS_goal
GS_start
insert_olist
remove_olist
replan
rhs
update
update_node
update_olist
Nested Types
HASH : unordered_...
Typedef
key_compare
Struct
binary_function< PD, PD,...
OPENLhash : unord...
Typedef
OPENLIST : multim...
Typedef
openPair : pair< PD...
Typedef


Map_Node
Class
Fields
cols
grid
rows
Methods
~Map_Node
blocked
exist
get_cols
get_rows
Map_Node (+ 1 overload)
operator()
Nested Types
Node
Class
Fields
cost
init
neighbours
num_nbrs
prob
X
Y
Methods
~Node
get_cost
get_nbrs
get_prob
get_X
get_Y
Node (+ 1 overload)
set_cost
set_nbrs
set_prob
set_X
set_Y
Nested Types
Hashing
Class
unary_function< Node*, size_t>

The main benefit of these 2 modules is to represent the occupancy grid in Cartesian grid which will be the reference for the explored are; also it finds the robot's next destination and guides it to reach its next destination.

Obviously there are many similarities between the 2 packages specially in the 3 modules (Robot interface, Frontier, Global-map), and also the flow of orders through the 2 packages is the same. Next we will talk about each module separately.

Map-node

This class is responsible for the formation and modulation of the 2D occupancy grid. Each cell in the grid represents one of 3 states (OPEN, OCCUPIED, UNKNOWN). The 2D grid is updated via its robot exploration or via updates from other robots.

Frontier

This module is responsible for detecting the nearest frontier to the robot. By doing that it updates the next destination of the robot which is sent afterwards to the path planner module.

Path planner classes (A*/D*Lite)

These classes (AI agents) are responsible for planning a path from the robot's current position to its next destination point. This is done based on the states of the occupancy grid and the destination point received from the Frontier class.

Global-map

This class is responsible for the map fusion carried at each robot where the data gathered by other robots is fused with the robot's own data. After each robot carry out this process, the whole robots will be sharing the same information.

Robot-Interface

This is the wrapper class for the whole module which means that any communication between this module and a robot's module (whether from the simulator or a hardware robot) is carried through this class.

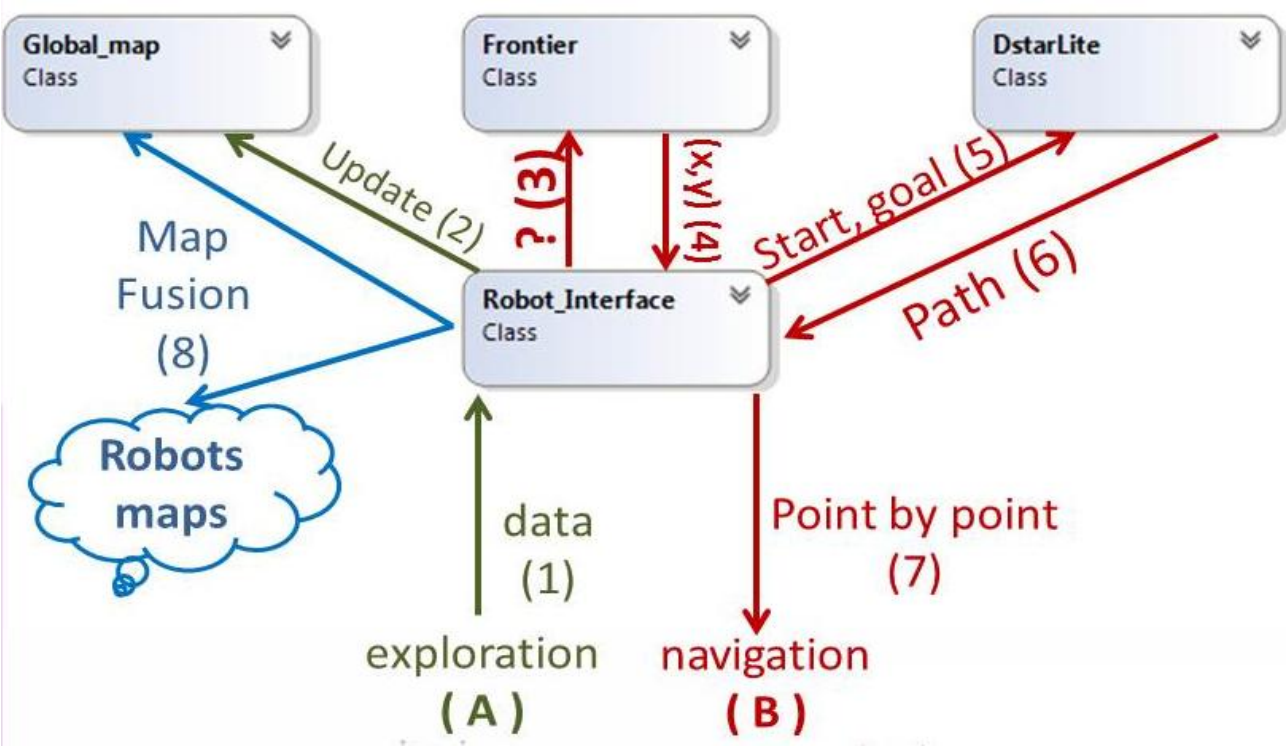
The next image illustrates better the expected flow of orders through the whole module and with the other modules.

Notice that, there exist 2 modes of operation:

First the ***exploration mode***, where the robot gather information from the outer space and update its occupancy grid.

Second the ***navigation mode***, where a new destination point is detected by the Frontier class, sent to the path planner class and finally the path is sent back to the robot interface class.

At any time, map fusion can take place between the robot and any nearest other robot.



CHAPTER 5: IMPLEMENTATION

5.1 Motion Planning Module

This is the module that represents the occupancy grid and that plans a path for the robot to reach its goal point; which is determined by the Frontier Module.

Motion planning (also known as the "navigation problem" or the "piano mover's problem") is a term used in robotics for the process of breaking down a desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement.
For example, consider navigating a mobile robot inside a building to a distant waypoint. It should execute this task while avoiding walls and not falling down stairs. A motion planning algorithm would take a description of these tasks as input, and produce the speed and turning commands sent to the robot's wheels. Motion planning algorithms might address robots with a larger number of joints (e.g., industrial manipulators), more complex tasks (e.g. manipulation of objects), different constraints (e.g., a car that can only drive forward), and uncertainty (e.g. imperfect models of the environment or robot).

Wikipedia definition

The goal of Motion Planning Module is to produce a continuous motion that connects a start configuration S and a goal configuration G, while avoiding collision with detected obstacles.

A motion planner is said to be **complete** if the planner in finite time either produces a solution or correctly reports that there is none. The robot and obstacle geometry is described in a 2D or 3D workspace, while the motion is represented as a path in configuration space.

This module is implemented using C++, as using .dll files it can be integrated with both the simulator and the base station of the hardware robot.

As mentioned before we have 2 path finding algorithms to compare between their performances, they are:

The A* and D*Lite algorithms which will be discussed next.

5.1.1 A* search algorithm

The A* is an algorithm used in path finding and graph traversal problems. It's an extension of Edsger Dijkstra's 1959 algorithm.

Starting from a start point, it uses best-first-search to find an optimal path (path with minimum cost) to a Goal point.

While keeping a sorted priority-queue of alternate path segments (nodes), it combines knowledge plus estimation (heuristic) to generate the cost used in sorting the nodes.

The above function represents the cost function in the A* algorithm.
f(x)=g(x)+h(x)

F(x) denotes the cost function of node x. This function is the summation of two other functions.

1. G(x) or the past path cost function which represents the knowledge part as it's a known distance from the start node to the goal node.
2. H(x) or the future path cost function (heuristic estimation) which is an estimated distance to the goal and it's commonly calculated as the straight distance to the goal node as it must be an admissible heuristic; that is, it must not over-estimate the distance to the goal.

Although the algorithm only calculates the length of the shortest path, the path itself can be tracked by allowing each node to keep track of its predecessor nodes which can be done using a map to save the directions.

The Heuristic function we used is the Euclidian distance because the robot can move in 8 directions, so we need to find the shortest path with respect to number of cells passed by the straight line distance to the goal.

A* Pseudo code:

```
function A*(start,goal)
    closedset := the empty set    // The set of nodes already evaluated.
    openset := {start}    // The set of tentative nodes to be evaluated, initially containing the start node
    came_from := the empty map    // The map of navigated nodes.

    g_score[start] := 0    // Cost from start along best known path.
    // Estimated total cost from start to goal through y.
    f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

    while openset is not empty
        current := the node in openset having the lowest f_score[] value
        if current = goal
            return reconstruct_path(came_from, goal)

        remove current from openset
        add current to closedset
        for each neighbor in neighbor_nodes(current)
            if neighbor in closedset
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from,current)
    total_path := [current]
    while current in came_from:
        current := came_from[current]
        total_path.append(current)
    return total_path
```

A* Pseudo code

*5.1.2 D*Lite algorithm*

Introduction:

Before talking about D*Lite we need to have a sufficient background about some topics.

First: the A* algorithm and the heuristic function which was discussed before.

Second: incremental search method.

> ***Incremental search methods:** methods that reuse information from previous searches to find shortest paths for series of similar path-planning problems which is faster than solving each path-planning problem from scratch.*
> A good example for incremental search method is Dynamic SWSF-FP (strict weak superior function- fixed-point) which is an algorithm used in solving the grammar problem.

> ***The grammar problem:** is a generalization of the single-source shortest-path problem (SSSP) introduced by Knuth which is to compute the minimum cost derivation of a terminal string from one or more non-terminals of a given context-free grammar.*

Third: Combining both incremental search methods and heuristic we get Incremental and heuristic search.

> ***Incremental heuristic search:** algorithms combine both incremental and heuristic search to speed up searches of sequences of similar search problems, which is important in domains that are only incompletely known or change dynamically.

The incremental heuristic search has three main classes, which are:

- A class restarts A* at the point where its current search deviates from the previous one.
- A class updates the h-values from the previous search during the current search to make them more informed.

- A class updates the g-values from the previous search during the current search to correct them when necessary, which can be interpreted as transforming the A* search tree from the previous search into the A* search tree for the current search. An example for it is LPA* which is our fourth step.

Fourth: LPA*.

LPA* (life-long planning A*):

1. Is an incremental version of A*.

2. It repeatedly finds shortest paths from a given start vertex to a given goal vertex while the edge costs of a graph change or vertices are added or deleted.

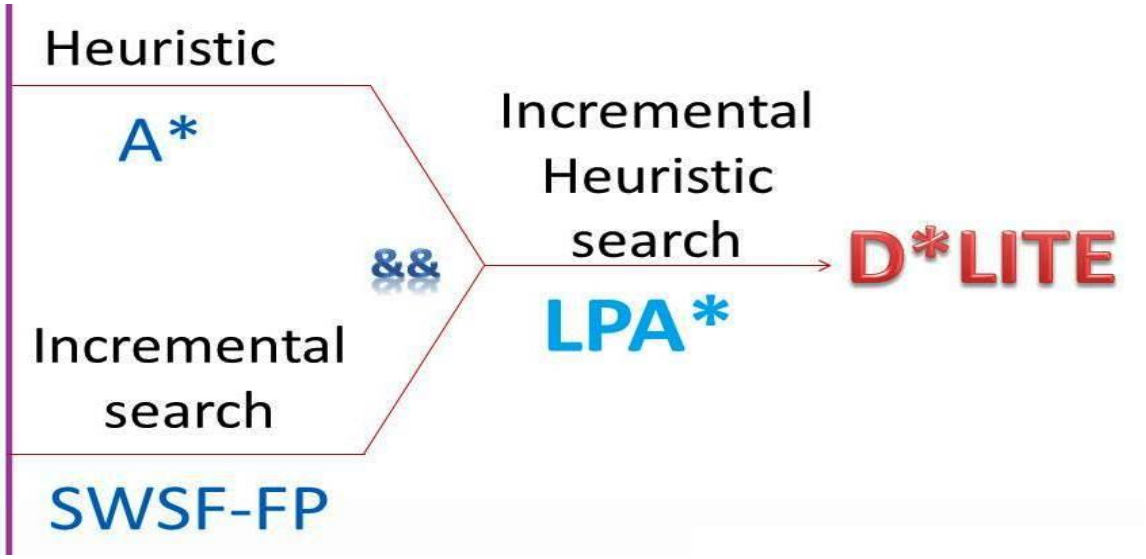
3. Its first search is the same as A* in selecting the next node in a path depending on its g-values (the smaller the better) but many of the subsequent searches are potentially faster because it reuses similar parts in previous search tree which are identical to the new tree.

Now comes the D*Lite.

D* Lite is an incremental heuristic search algorithm which is built on LPA* and combines ideas of A* and SWSF-FP.

Note: D* stands for Dynamic A*.

The following picture may illustrate better the relation between D*Lite, LPA*, A* and SWSF-FP.



It's widely used for mobile robots and autonomous vehicle navigation. It's basically used in making assumptions in exploring unknown area (terrain).

Like any other search algorithm it finds the most optimal path (shortest path) from a start point to a goal point but what distinguishes it from other searching algorithms is its way in handling previously unknown obstacles.

The D* Lite –when encountering such an obstacle- modifies only a part of its path which is unlike what happened in A* which in such a case will need to re-plan the whole path.

About the D* Lite implementation

While A* and LPA* searches from the start vertex to the goal vertex, D* Lite searches from the goal vertex to the start vertex and thus its g-values are estimates of the goal distances.

Like LPA* it maintains a priority queue to store the currently open nodes where each node has 2 components.

min (g(x),rhs(x))+h(x)

min (g(x),rhs(x))

Where,

x: current node

g(x): distance from start node to node x

rhs(x): one-step look ahead value based on g(x)

h(x): heuristic function (estimation of distance from node x to the goal node)

It uses a method derived from the original D* to avoid reordering the priority queue when the robot notices a change in edge costs when it moves.
In such a case, the priority queue probably contains a large number of vertices and the repeated reordering of it would be expensive. It does so by adding every new emerged priority to a variable Km, then adding this variable to the first component of every node.

D*Lite Pseudo code:

procedure CalculateKey(s)
{01"} return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m ; \min(g(s), rhs(s))]$;

procedure Initialize()
{02"} $U = \emptyset$;
{03"} $k_m = 0$;
{04"} for all $s \in S\ rhs(s) = g(s) = \infty$;
{05"} $rhs(s_{goal}) = 0$;
{06"} U.Insert$(s_{goal}, [h(s_{start}, s_{goal}); 0])$;

procedure UpdateVertex(u)
{07"} if $(g(u) \neq rhs(u)$ AND $u \in U)$ U.Update$(u, CalculateKey(u))$;
{08"} else if $(g(u) \neq rhs(u)$ AND $u \notin U)$ U.Insert$(u, CalculateKey(u))$;
{09"} else if $(g(u) = rhs(u)$ AND $u \in U)$ U.Remove(u);

procedure ComputeShortestPath()
{10"} while (U.TopKey()$\dot{<}$CalculateKey(s_{start}) OR $rhs(s_{start}) > g(s_{start}))$
{11"} u = U.Top();
{12"} k_{old} = U.TopKey();
{13"} k_{new} = CalculateKey(u));
{14"} if$(k_{old} \dot{<} k_{new})$
{15"} U.Update(u, k_{new});
{16"} else if $(g(u) > rhs(u))$
{17"} $g(u) = rhs(u)$;
{18"} U.Remove(u);
{19"} for all $s \in Pred(u)$
{20"} if $(s \neq s_{goal})\ rhs(s) = \min(rhs(s), c(s, u) + g(u))$;
{21"} UpdateVertex(s);
{22"} else
{23"} $g_{old} = g(u)$;
{24"} $g(u) = \infty$;
{25"} for all $s \in Pred(u) \cup \{u\}$
{26"} if $(rhs(s) = c(s, u) + g_{old})$
{27"} if $(s \neq s_{goal})\ rhs(s) = \min_{s' \in Succ(s)}(c(s, s') + g(s'))$;
{28"} UpdateVertex(s);

D*Lite Pseudo code

procedure Main()
{29"} $s_{last} = s_{start}$;
{30"} Initialize();
{31"} ComputeShortestPath();
{32"} while $(s_{start} \neq s_{goal})$
{33"} /* if $(rhs(s_{start}) = \infty)$ then there is no known path */
{34"} $s_{start} = \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
{35"} Move to s_{start};
{36"} Scan graph for changed edge costs;
{37"} if any edge costs changed
{38"} $k_m = k_m + h(s_{last}, s_{start})$;
{39"} $s_{last} = s_{start}$;
{40"} for all directed edges (u, v) with changed edge costs
{41"} $c_{old} = c(u, v)$;
{42"} Update the edge cost $c(u, v)$;
{43"} if $(c_{old} > c(u, v))$
{44"} if $(u \neq s_{goal})$ $rhs(u) = \min(rhs(u), c(u, v) + g(v))$;
{45"} else if $(rhs(u) = c_{old} + g(v))$
{46"} if $(u \neq s_{goal})$ $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$;
{47"} UpdateVertex(u);
{48"} ComputeShortestPath();

D*Lite Pseudo code

5.2 ROS and Simulation implementation

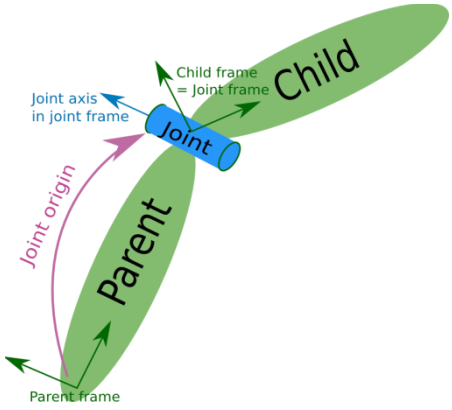
5.2.1 Robot's 3D model Implementation

The way ROS uses the 3D model of a robot or its parts, to simulate them is by means of the URDF files

(URDF Unified Robot Description Format is an XML format that describes a robot, its parts, its joints, dimensions, and so on.)

5.2.1.1 Joints & Links:

There are two principal fields that describe the geometry of a robot, links and joints where for each joint there exists parent link and child link and names for both joints and links must be unique.



In our model we have

- Five links, Base link for the car base and four links for the four wheels.
- Four joints each connect the base link to one wheel.
- Two Hokuyo laser sensors.

If you want to simulate the robot on Gazebo or any other simulation software, it is necessary to add physical and collision properties. This means that we need to set the dimension of the geometry to calculate the possible collisions, for example, the weight that will give us the inertia, and so on.

It is necessary that all links on the model file have these parameters; if not, the robot could not be simulated.

5.2.1.2 Sensors:

Normally, when you want to add a new sensor you need to implement the behavior. Fortunately, the laser is implemented in **the erratic_description package**. This is the magic of ROS; you can re-use code from other packages for your development.

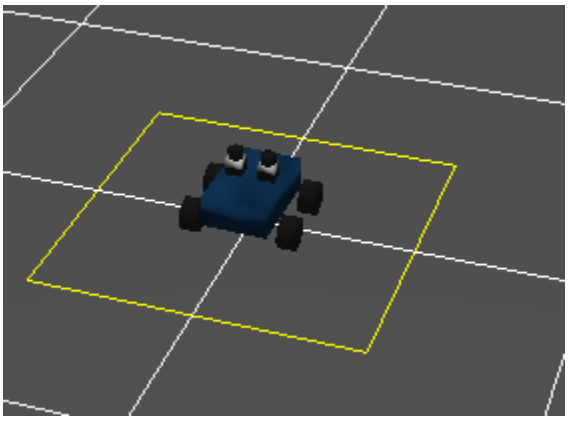
Hokuyo laser

In mobile robotics, it is very important to know where the obstacles are, the outline of a room, and so on. The sensor used for these purposes is Lidar. This sensor is used to measure distances between the robot and objects.

Hokuyo rangefinder is a device used for navigation and building maps in real time. It is a low-cost Lidar that is widely used in robotics.

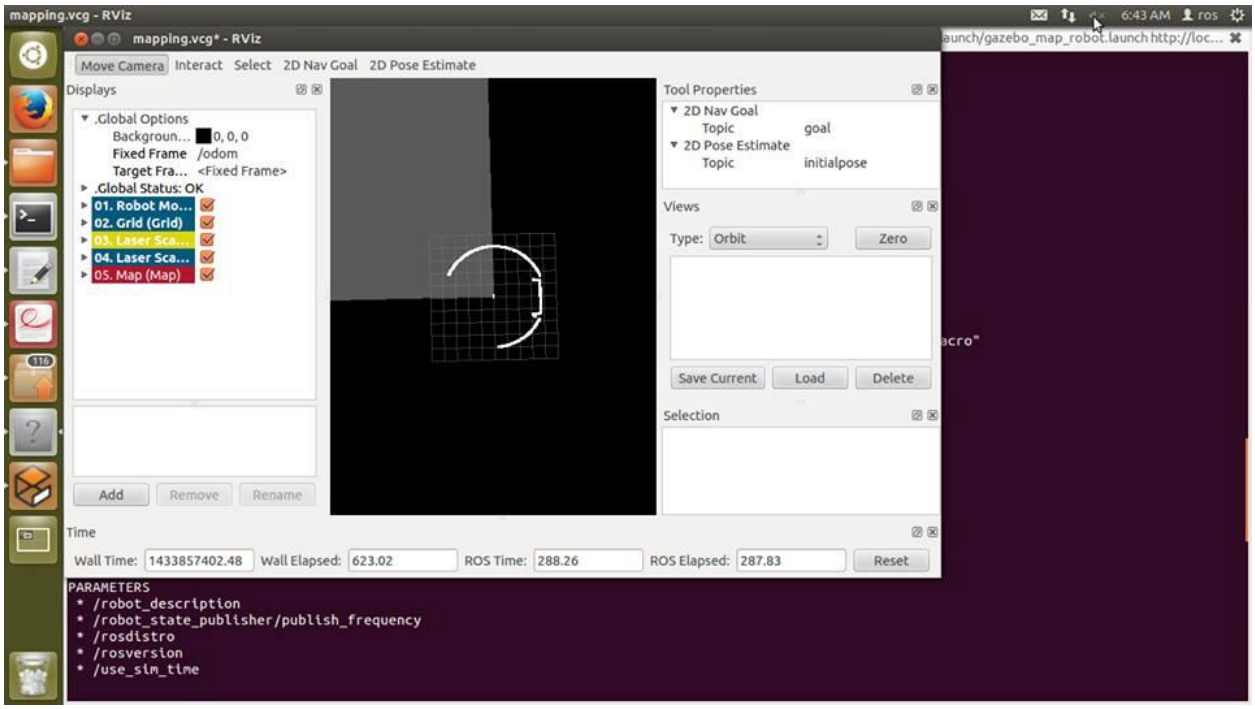
Hokuyo Specs

- detectable range is 20mm to 5600mm
- 100msec/scan
- 5V operating voltage
- 240° area scanning range with 0.36° angular resolution



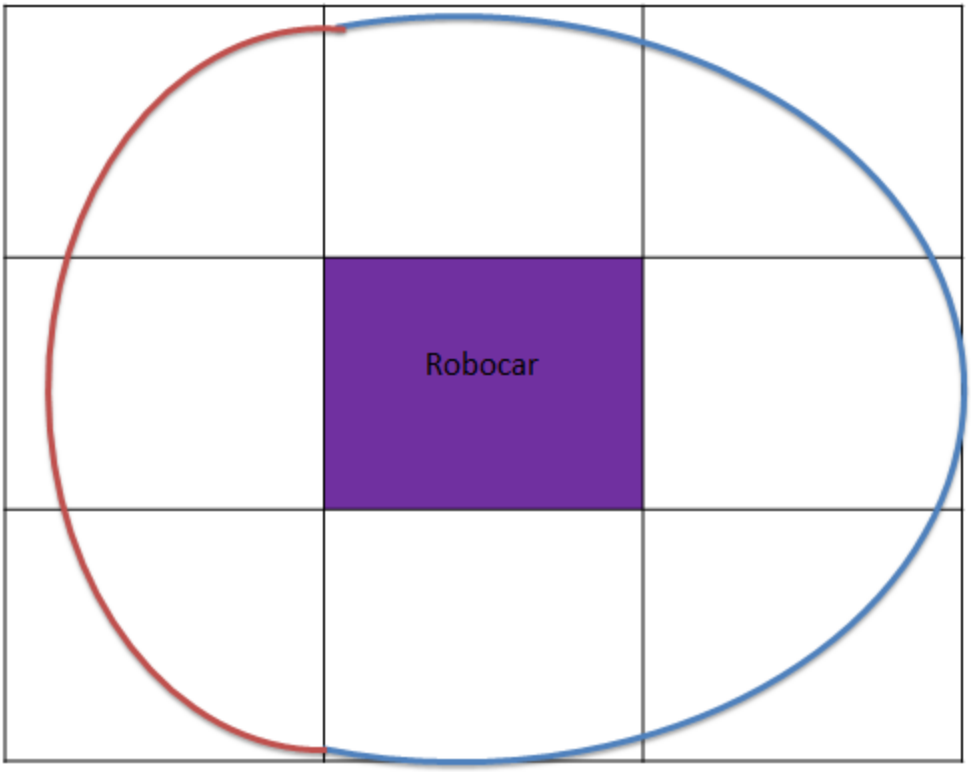
And as a result that the Sensors area scanning range is 240° we used two Hokuyo range finders to cover all the surrounding area.

Sensors Data



Using one sensor

As we have said before, we need two sensors to get a 360-degrees-data around the Robo-car. We divide the 360 degrees among a 3x3-matrix, where cell (1, 1) represents the Robo-car itself. But by using two sensors, there exists overlapping on cells. We can use this overlapping to get more accuracy, but it takes more computational-time.



To get data from a sensor, the sensor publishes the result on a topic, where each sensor publishes to its topic. The next step is to make a node which subscribes these topics and get a result. This node is named Control_movement. We divide data from two sensors on the remaining 8 cells of the matrix.

Finally, the Control_movement node publishes the 3x3 matrix to the new topic to send it to the Path-planning module.

Now that we have the model of our robot written in the URDF file and know how to use the Sensors data, we have to set the environment that the robot will navigate , fortunately, Gazebo provides a variety of Environments and we just picked one. Now to watch the model in 3D we wrote a launch file in the launch folder of the package and in the file we specified what files we need to launch with which simulator then we should call the launch file from the command prompt.

Xacro:

When Writing the URDF file before adding the sensors, the code became large, imagine if we start to add sensors and any other new feature on the model, the file will start to increase and the maintenance of the code will start to become more complicated, so we used Xacro.

Xacro is an XML language. The main feature of Xacro is its support for Macros, where Macros are used to make tasks using the application less repetitive and allow a programmer to enable code reuse.

With Xacro, we construct shorter and more readable XML files by using macros that expand to larger XML expressions. This package is most useful when working with large XML documents such as robot descriptions. It is heavily used in packages such as the URDF.

And for Multi-Robot simulation we must include each robot's URDF file within the launch file and its controlling nodes.

Now that we construct our Robot we need to move it. We already talked about the algorithms used in path planning (A* & D*) and about some approaches used in the navigation.

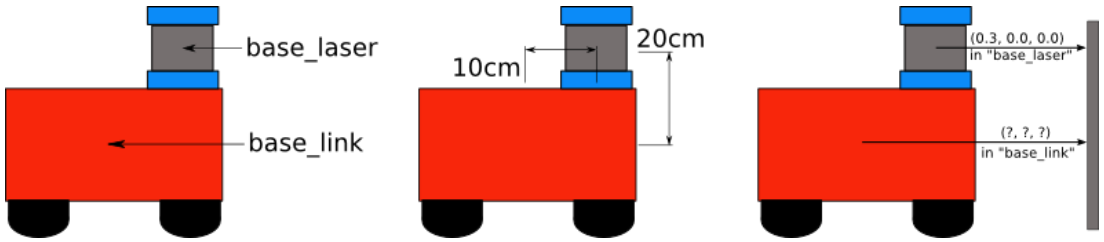
5.2.1.3 Robot Transform Tree

Many ROS packages require the transform tree of a robot to be published using the tf software library. At an abstract level, a transform tree defines offsets in terms of both translation and rotation between different coordinate frames.

To make this more concrete, consider our robot that has a mobile base with two rangefinders mounted on top of it. Now let's define two coordinate frames: one corresponding to the center point of the base of the robot and one for the center point of one rangefinder. Let's also give them names for easy reference. We'll call the coordinate frame attached to the mobile base "base_link" and we'll call the coordinate frame attached to the laser "laser_base_link1".

At this point, let's assume that we have some data from the laser in the form of distances from the laser's center point. In other words, we have some data in the "laser_base_link1" coordinate frame.

Now suppose we want to take this data and use it to help the mobile base avoids obstacles in the world. To do this successfully, we need a way of transforming the laser scan we've received from the "laser_base_link1" frame to the "base_link" frame. So we need to define a relationship between the "laser_base_link1"and "base_link" coordinate frames.



In defining this relationship, assume we know that the laser is mounted 10cm forward and 20cm above the center point of the mobile base. This gives us a translational offset that relates the "base_link" frame to the "laser_base_link1" frame. we know that to get data from the "base_link" frame to the "laser_base_link1" frame we must apply a translation of (x: 0.1m, y: 0.0m, z: 0.2m), and to get data from the "laser_base_link1"

frame to the "base_link" frame we must apply the opposite translation (x: -0.1m, y: 0.0m, z: -0.20m).

We could choose to manage this relationship ourselves, meaning storing and applying the appropriate translations between the frames when necessary, but this becomes a real pain as the number of coordinate frames increase.

Luckily, we don't have to do this work ourselves. Instead we'll define the relationship between "base_link" and “laser_base_link1” once using tf and let it manage the transformation between the two coordinate frames for us.

To define and store the relationship between the "base_link" and “laser_base_link1” frames using tf, we need to add them to a transform tree. Conceptually, each node in the transform tree corresponds to a coordinate frame and each edge corresponds to the transform that needs to be applied to move from the current node to its child.

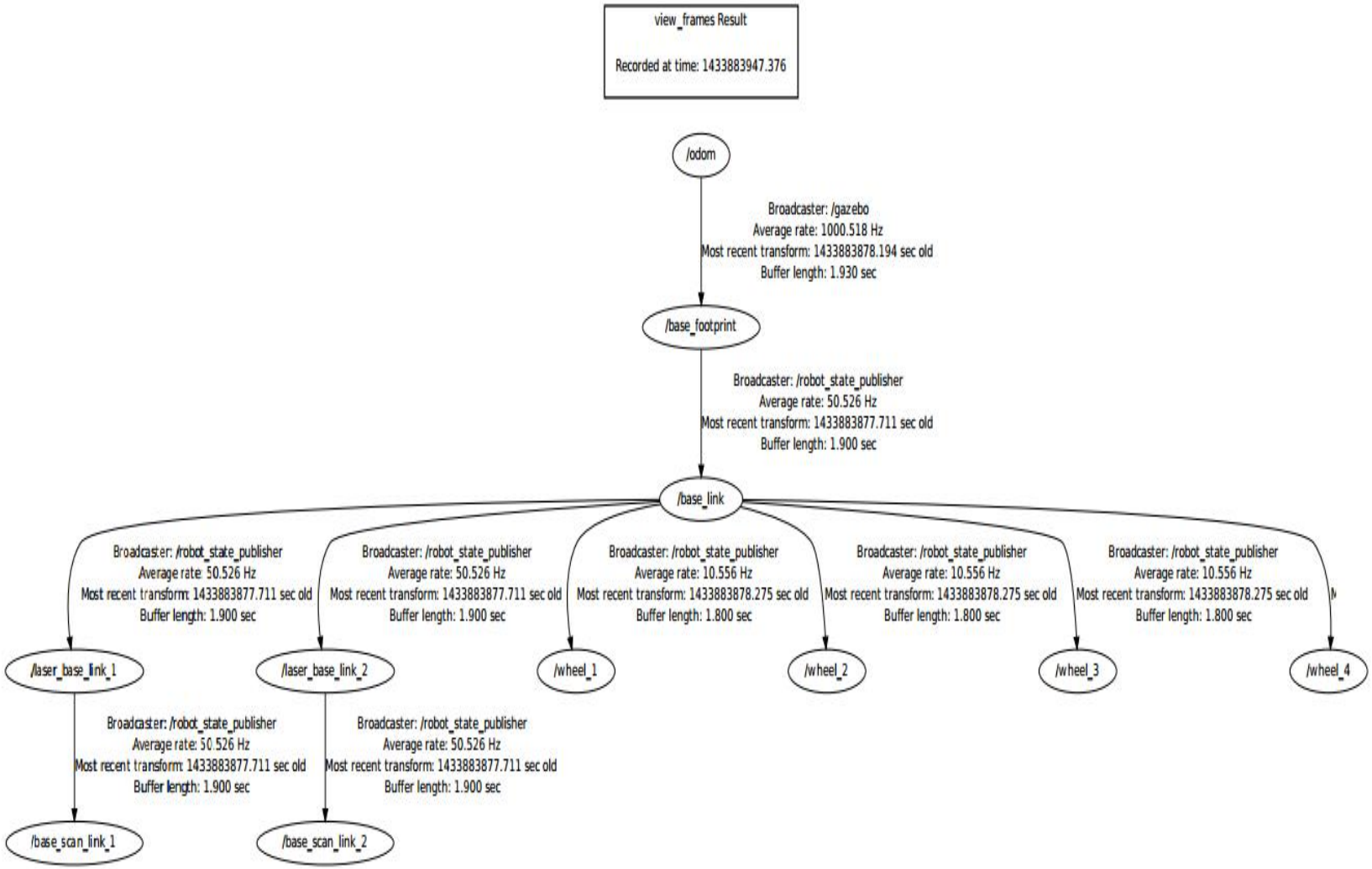
Tf uses a tree structure to guarantee that there is only a single traversal that links any two coordinate frames together, and assumes that all edges in the tree are directed from parent to child nodes

To create a transform tree for our robot, we'll create two nodes, one for the "base_link" coordinate frame and one for the “laser_base_link1” coordinate frame. To create the edge between them, we first need to decide which node will be the parent and which will be the child. This is important because tf assumes that all transforms move from parent to child.

We chose the "base_link" coordinate frame as the parent because other pieces/sensors will be connected to it. This means the transform associated with the edge connecting "base_link" and “laser_base_link1” should be (x: 0.1m, y: 0.0m, z: 0.2m).

With this transform tree set up, converting the laser scan received in the “laser_base_link1” frame to the "base_link" frame is as simple as making a call to the tf library.

The Figure Shows our robot transformation tree , which is generated using the ROS command rosrun tf view_frames.


view_frames Result
Recorded at time: 1433883947.376
/odom
Broadcaster: /gazebo
Average rate: 1000.518 Hz
Most recent transform: 1433883878.194 sec old
Buffer length: 1.930 sec
/base_footprint
Broadcaster: /robot_state_publisher
Average rate: 50.526 Hz
Most recent transform: 1433883877.711 sec old
Buffer length: 1.900 sec
/base_link
Broadcaster: /robot_state_publisher
Average rate: 50.526 Hz
Most recent transform: 1433883877.711 sec old
Buffer length: 1.900 sec
Broadcaster: /robot_state_publisher
Average rate: 50.526 Hz
Most recent transform: 1433883877.711 sec old
Buffer length: 1.900 sec
Broadcaster: /robot_state_publisher
Average rate: 10.556 Hz
Most recent transform: 1433883878.275 sec old
Buffer length: 1.800 sec
Broadcaster: /robot_state_publisher
Average rate: 10.556 Hz
Most recent transform: 1433883878.275 sec old
Buffer length: 1.800 sec
Broadcaster: /robot_state_publisher
Average rate: 10.556 Hz
Most recent transform: 1433883878.275 sec old
Buffer length: 1.800 sec
/laser_base_link_1
/laser_base_link_2
/wheel_1
/wheel_2
/wheel_3
/wheel_4
Broadcaster: /robot_state_publisher
Average rate: 50.526 Hz
Most recent transform: 1433883877.711 sec old
Buffer length: 1.900 sec
Broadcaster: /robot_state_publisher
Average rate: 50.526 Hz
Most recent transform: 1433883877.711 sec old
Buffer length: 1.900 sec
/base_scan_link_1
/base_scan_link_2

Robots odometry :

During the navigation we need to receive data from the robot odometry where the odometry is the distance of something relative to a point.
In our case, it is the distance between “base_link” and a fixed point in the frame “odom”.

Odometry message in ROS include more than the pose of the robot. Besides the odometry, the Robot’s linear and angular velocity and other stuff are sent by robot.

The next figure shows the structure of the message:

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string child_frame_id
geometry_msgs/PoseWithCovariance pose
  geometry_msgs/Pose pose
    geometry_msgs/Point position
      float64 x
      float64 y
      float64 z
    geometry_msgs/Quaternion orientation
      float64 x
      float64 y
      float64 z
      float64 w
  float64[36] covariance
geometry_msgs/TwistWithCovariance twist
  geometry_msgs/Twist twist
    geometry_msgs/Vector3 linear
      float64 x
      float64 y
      float64 z
    geometry_msgs/Vector3 angular
      float64 x
      float64 y
      float64 z
  float64[36] covariance
```

As the odometry is the displacement between two frames, it is necessary to publish the transform of it and it helps us in localization of the robot.

CONCLUSION & FUTURE WORK

Project Conc.

We Studded the Project Co-Swarm ’13, the concepts they used, their implementation and the problems they faced.

one of our targets was to solve their big problem which was the malfunction of the Real Robots “Robo-Car” in the Robotics lab by building a cost less efficient Simulated Robot using ROS as an operating system and Gazebo as a Simulator, and another target was to accomplish one of their future work which is using Dynamic environment and D*lite in path planning.

Our journey through that Project wasn’t a piece of cake and our huge problem especially in building the Robot is that we didn’t find people that worked with ROS or build their Simulated robots before, and when facing a Problem we communicate with people outside Egypt that worked with ROS or Gazebo in one way or another, we even contacted one of “Learning ROS for robotics programming” writers “Enrique Fernandez” but he couldn’t help because of his tight time , Finally we were on our own but he still helped us a lot with his book, we really do recommend it for learning ROS, it’s a good start.

We managed to build a Simulated Robot with no cost with big Similarity to that in the Robotics lab but more efficient, we also wrote the D*lite algorithm.
We are in the integration phase between the code and the simulated robot but we are facing some problems in simulating multi-Robot, we still have time till the Final Seminar….

Future Work:

- Monte-Carlo localization (Markov localization).

REFERENCES

References:

- **Frontier-Based Exploration Using Multiple Robots** – Brain Yamauchi – 1998.
- **A Frontier-Based Approach for Autonomous Exploration** – Brain Yamauchi – 1997.
- **Comparative Study of Algorithms for Frontier based Area Exploration and Slam for Mobile Robots** - Dayanand V ,Rahul Sharma K and Gireesh Kumar – 2013
- **A comparison of path planning strategies for autonomous exploration and mapping of unknown environments** – Miguel Juliá – 2012.
- **Lifelong Planning A***- Sven Koenig, Maxim Likhachev , and David Furcy.
- **D* Lite** - Sven Koenig, MaximLikhachev .
- **An Incremental Algorithm for a Generalization of the Shortest-Path Problem** - G. Ramalingam and Thomas Reps.
- **Survey of Swarm Robotics Techniques** - Angie Shia.
- **Swarm Robotics Algorithms** - Don Miner.
- **A Survey and Comparison of Commercial and Open-Source Robotic Simulator Software**– AaronStaranowicz andGian Luca Mariottini – 2011 .
- **Learning ROS for robotics programming** – Enrique Fernandez & Aaron Martinez – Book – September 2013.
- **Scanning Laser Range Finder URG-04LX-UG01** **data sheet** .