People's Democratic Republic of Algeria
Ministry of Higher Education and Scientific Research

University M'Hamed BOUGARA – Boumerdes

Institute of Electrical and Electronic Engineering

Department of Power and Control / Electronics

Final Year Project Report Presented in Partial Fulfilment of
the Requirements for the Degree of

MASTER

In **Control / Electronics**

Option: **Control / Computer Engineering**

Title:

Kinect-based path planning and execution for autonomous mobile robot using ROS: Case of RobuTER

Presented by:

- **GUIR Dallel (Control)**
- **SAIGHI Souhila (Computer Engineering)**

Supervisor:

Dr. BOUSHAKI Razika

Mr. HENTOUT Abdelfeteh

Registration Number:……/2017

Acknowledgement

First and foremost, We are thankful to God almighty, for showing heavenly blessing upon us, as without that nothing would have been possible.

We would like to express my felt gratitude to our supervisors

Mr. HENTOUT ABDELFETAH from CDTA and Mss. BOUSHAKI RAZIKA from INELEC for their wisdom, guidance and support.

Also, we would like to thank CDTA for opening their doors to us and offering their help to make this work happen.

A special thanks to our families and friends for their support and all INELEC staffs.

Dedication:

I would like to dedicate this work:

To my dear father and my lovely mother

To my brothers and my sister Manel

To all my family especially my Grandfather

To all my dear friends especially my best friend Souhila.

Also to my future husband.

Dallel

Dedication:

I would like to dedicate this work:

To my dear father and my lovely mother

To my brothers and my sisters.

To all my family especially Maissa.

To all my dear friends especially my best friend Dallel.

Souhila

Abstract

This project presents preliminary results of the application of two-Kinect cameras system on a two wheeled indoor mobile robot for off-line path planning and execution. In our approach, the robot makes use of depth information delivered by the vision system to accurately model its surrounding environment through image processing techniques. In addition, a Rapidly-exploring random tree is implemented to generate a collision-free path linking an initial configuration of the mobile robot (Source) to a final configuration (Target). After that, Piecewise Cubic Hermite Interpolating Polynomial is used to smooth the generated optimal path. Finally, a comparison between RRT and Genetic algorithm has been done.

Contents

List of figures

Chapter 1: Introduction to mobile robot and navigation

Chapter 2: Path planning approches

Chapter 3: Implementation

Chapter 04: Experimental validation

Appendix

List of Tables

Chapter 02:

Chapter 04:

List of acronym

IR : Infrared.

ICC : Instantaneous Center of Curve.

MPRT : Mobile Robot Programming Toolkit

ROS : Robot Operating System.

RVIZ : Robot Visualisation.

RGB : Red, Green, Blue.

OpenCV : Open Source Computer Vision.

OpenNI : Open Natural System.

STAIR : Simple Two-Dimensional Robot Simulator.

YARP : Yet Another Robot Platform.

Introduction

Path generation and execution is one of the most important tasks for autonomous mobile robots. Indeed, the robot has to perform certain interrelated activities [27] including (i) task planning (generation of operations plans), (ii) environment modeling and multi-sensory fusion, (iii) path planning, (iv) localization of the robot inside its environment, and (v) path execution and tracking.

Mobile robot may operate in different modes; one of these modes is the autonomous mode including intelligent systems those have a capacity to acquire and apply knowledge in an "intelligent" manner and have the capabilities of perception, reasoning, learning, and making inferences (or decisions) to complete a given task. Hence making smarter robots is a problem that has faced the scientific community for several years, now, to reach a goal position starting from a known initial point with some desired criteria is a complicated task. So the problem has been solved with soft computing methods.

For a mobile robot to perceive its environment and localizes itself within it, vision has become a standard sensory tool. Especially with the advancement of image processing techniques, which facilitates the extraction of the useful information from images captured by cameras.

The objective of our project is to deal with the development of a strategy to tackling the collision-free trajectory planning problem for mobile robots evolving in indoor environment with obstacles, using ROS; where the generated feasible path connect an initial location to an imposed final location. The robot makes use of kinect cameras to model its environment by applying soft computing techniques and image processing tools for perceiving the environment and map building.

Rapidly-exploring random trees used to generate a feasible path joining the initial and final position. Finally, a comparative summary between RRTs and genetic algorithm has been done.

1.1. Introduction

Mobile robots are the objects which move around in their environment and are not fixed to one physical location. They can be controlled by Bluetooth, wireless network of PC, a wireless remote control microcontroller.

Robot navigation means the robot ability to determine its own position in its reference frame and then to plan a path toward some goal location. In order to navigate in its environment, the robot requires representation, i.e. a map of the environment and the ability to interpret that representation [1].

Navigation can be defined as the combination of the three fundamental competences [5]:

- Self-localization.
- Path planning.
- Map-building and map-interpretation (map use).

The range of potential applications for mobile robots is enormous. It includes [3]:

- Medical services: Service robots.
- Automatic cleaning of (large) areas.
- Agricultural: Fruit and vegetable picking, fertilization, planting…
- Forests: Cleaning, fire preventing, tree cutting…
- Hazard environments.
- Construction and demolishing.
- Military.

1.2. Navigation

Leonard and Durrant-Whyte [2] summarized the general problem of mobile robot navigation by three questions:

- Where am I?
- Where am I going?
- How do I get there?

In order to tackle these questions, the robot has to:

- Handle a map of its environment.
- Self-localize itself in the environment.
- Plan a path from its location to a desired location.

Therefore, the robot has to have a model of the environment, be able to perceive, estimate its relative state and finally plan and execute its movement.

An autonomous robot navigation system has traditionally been hierarchical; it consists of a dynamical control loop with four main elements (Figure 1) [4]:

- *Perception* as obtaining and interpreting sensory information.
- *Mapping/localization* involving the construction of a spatial representation by using the information perceived from its sensors and estimating the robot position within the spatial map.
- *Cognition* the strategy to find a path toward a goal location.
- *Motor control* where motor actions are determined and adapted to environmental changes.

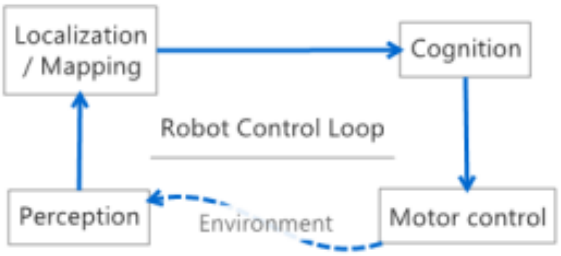


Fig. 1-1. Autonomous navigation problem

The *classical control paradigm* (Horizontal/Functional decomposition) is based on the sequence *Sense* → *Think* → *Act* [2] as shown by Figure 1-2.

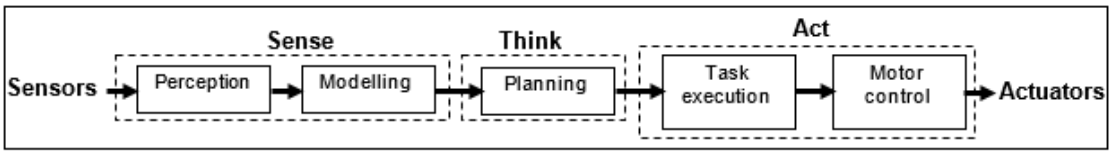


Fig. 1-2. Functional decomposition of mobile robot control system [5]

1.2.1. Perception

The first action in the control loop is perception of the robot itself and its environment, which is done through *proprioceptive* and *exteroceptive* sensors. Proprioceptive sensors capture information about the self-state of the robot; whereas, exteroceptive sensors capture information about the environment.

The types of sensors being used on mobile robots shows a big variety. The most relevant ones can be briefly listed as encoders, gyroscopes, accelerometers, sonars, laser range finders, beacon-based sensors and vision sensors.

In theory, navigation can be realized using only proprioceptive sensors (odometry). It is basically calculating the robot position based on the rotation of wheels and/or calculating orientations using gyroscopes/accelerometers. But in real world settings, odometry performs poorly over time due to unbounded growth of integration errors caused by uncertainties. It is also possible to navigate using only exteroceptive sensors. One such realization of this approach is the *Global Positioning System* (*GPS*) which is being successfully used in vehicle navigation systems [2].

1.2.2. Localization

The goal for an autonomous robot is to be able to construct (or use) a map or floor plan and to localize itself in it.

The problem of robot localization consists of answering the question: *Where am I?*. From the robot's point of view, this means that the robot has to find out its location relative to the environment. And, this poses difficult challenges because of the inaccuracy and incompleteness of the sensors and actuators. The robot must also have a representation of its belief regarding its position on the map. Where the design questions for belief representation are: *Does the robot identify a single unique position as its current position?*, or *Does the robot describe its position in terms of a set of possible positions?* If multiple possible positions are expressed in a single belief, *How are those multiple positions ranked?* [2].

Markov localization addresses the global localization problem, the position tracking problem, and the kidnapped robot problem. Where it tracks the robot's belief state using an arbitrary probability density function to represent the robot's position.

Kalman filter localization tracks the robot's belief state typically as a single hypothesis with normal distribution. It addresses the position tracking problem. The Kalman filters are designed to operate efficiently on Gaussian distributions which are defined by the mean plus a standard deviation parameter σ [2].

Other localization methods include the use of passive objects in the environment such as landmark-based navigation, positioning beacon systems, and route-based localization strategies [2].

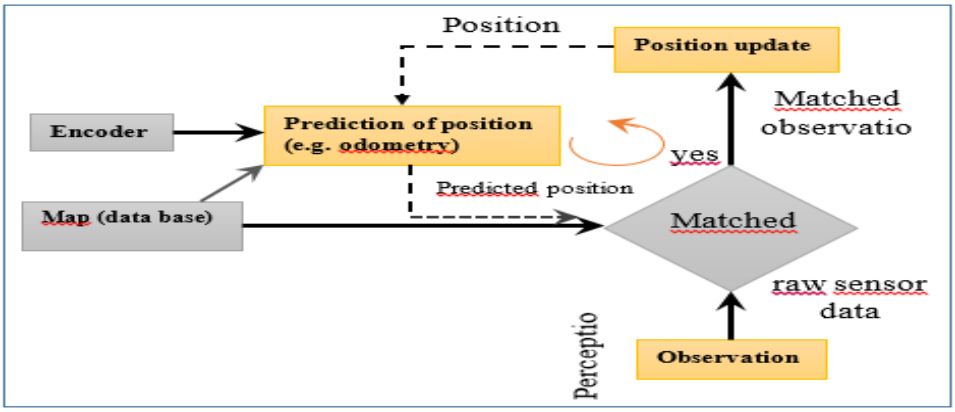


Fig. 1-3: General schematic for mobile robot localization [2]

1.2.3. Path planning

Path planning can be defined as searching a suitable path in a map from one place to another, without colliding with any obstacles. The Robot motion planning can be basically divided into two main categories: (*i*) local path planning and (*ii*) global path planning:

- In global path planning, the environment is known in advance and the terrain is static or the obstacles are known in advance. Hence, the path planning algorithm is able to make a complete map of the environment form the start point to the goal even before the robot starts motion.
- On the other hand, in local path planning, the environment is completely unknown to the mobile robot; i.e. the environment is dynamic and unstructured or the obstacles are not known in advance. In such a situation, the robot needs to gather information about the environment in real time and update its control laws so as to achieve this [6].

Various techniques have been exercised in path planning; they are classified into two categories: (*i*) classical and (*ii*) heuristic methods.

1.2.3.1. Classical methods

In classical methods, either a solution would be found or it would be proven that such a solution does not exist. The main disadvantage of such methods is their computationally intensiveness and their inability to cope with uncertainty. Such disadvantages make their usage brittle in real-world applications. This is due to the natural characteristics of such applications which are being unpredictable and uncertain [7]. Some examples are as follows:

- Cell decomposition.
- Roadmap.
- Potential field.

1.2.3.2. Heuristic methods

The abovementioned classical approaches suffer from many drawbacks, such as high time complexity in high dimensions, and trapping in local minima, which makes them inefficient in practice. In order to improve the efficiency of classical methods, many algorithms have been developed to solve the various encountered problems including:

- Probabilistic Roadmaps (PRM).
- Genetic algorithm.
- Rapidly exploring random tree (RRT).
- Particle Swarm Optimization (PSO).

The major advantage of heuristic methods is the high-speed in the implementation [8].

1.2.4. Motion control

A mobile robot needs locomotion mechanisms that enable it to move unbounded throughout its environment. But, there are a large variety of possible ways to move; so, the selection of a robot's approach to locomotion is an important aspect of mobile robot

design. In locomotion, the environment is fixed and the robot moves by imparting force to the environment.

Locomotion and manipulation thus share the same core issues of stability, contact characteristics and environmental type [2]:

- *Stability*: number and geometry of contact points, center of gravity, static/dynamic stability, inclination of terrain.
- *Characteristics of contact*: contact point/path size and shape, angle of contact, friction.
- *Type of environment*: structure, medium (e.g. water, air, soft or hard ground).

In mobile robotics, we need to understand the mechanical behavior of the robot both in order to design appropriate mobile robots for tasks and to understand how to create control software for an instance of mobile robot hardware. Kinematics is the most basic study of how mechanical systems behave. In addition, the process of understanding the motions of a robot begins with the process of describing the contribution each wheel provides for motion.

1.3. Conclusion

This chapter provides a brief description to autonomous mobile robots and navigation. All the necessary terms used throughout this work, including the different block diagrams required for successful autonomous navigation starting from perception and map building, followed by the localization problem, then different path planning techniques.

2.1. Introduction

Path planning is one of the fundamental problems in mobile robotics. As mentioned by Latombe [9], the capability of effectively planning its motions is *"eminently necessary since, by definition, a robot accomplishes tasks by moving in the real world".* Especially in the context of autonomous mobile robots, path planning techniques have to simultaneously solve two complementary tasks. On one hand, their task is to minimize the length of the trajectory from the starting position to the target location; on the other hand, they should maximize the distance to obstacles in order to minimize the risk of colliding with an object.

Various approaches have been introduced to implement path planning for a mobile robot [10]. The approaches are according to environment, type of sensors, robot capabilities, etc. In this chapter, we will introduce some path planning algorithms and discuss their advantages and limits.

2.2. Path planning algorithms

Existing methods on path planning can be classified into *off-line* and *on-line* planning approaches [11] [12]. *Off-line planning approaches* generate the entire path to the *Target* before motion begins; These approaches use complete information about the workspace, where an optimization criterion can be used for searching the optimal collision-free trajectory. They are most useful for repeatable tasks in static environments where optimality is essential (industrial applications, etc.). Whereas in *on-line planning approaches*, the trajectory to the *Target* is calculated incrementally during motion. These methods use incomplete information of the environment and build the trajectory step by step. Thus, this type enables fast collision detection and trajectory planning [13]. These approaches are required in applications where obstacles are detected during motion, the computation time required for a global solution delays the task execution, or simply as an alternative to a computationally expensive off-line search [14]. The following diagram represents the classification of the methods:

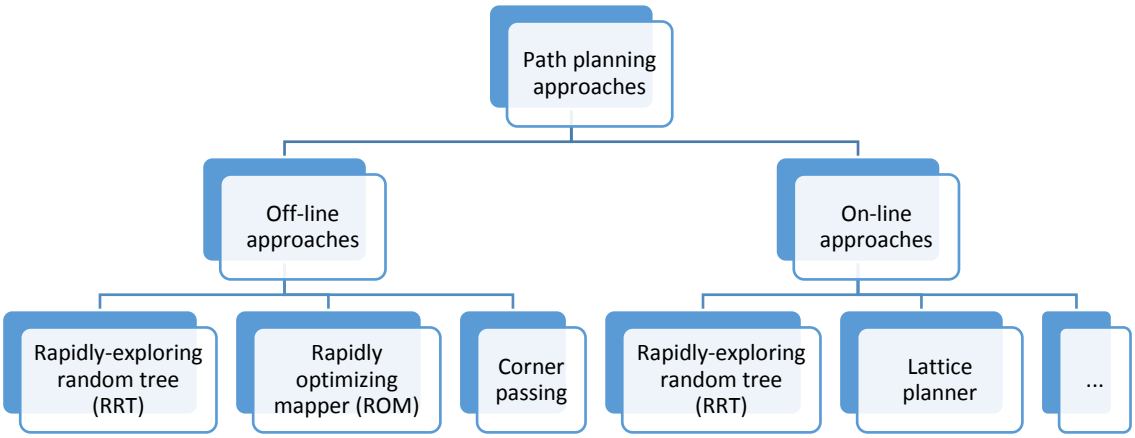


Fig 2-1. Classification of existing methods on path planning

In what follows, we only present some off-line and on-line path planning approaches for mobile robots.

2.2.1. Corner passing method

Corner passing method is based on binary image, which is created from obstacles. First, a line is drawn from the starting point toward the target; if there is an obstacle in the path, two paths from the starting point would be considered to the outermost corner of the object and the same process for each path should be repeated until reaching the destination without any collisions with obstacles. The distance of each path would be computed and the path which has the shortest distance is the optimal one. This method requires a lot of computing because at each stage the outermost corners of the object should be calculated, but it gives the most optimal path [15].

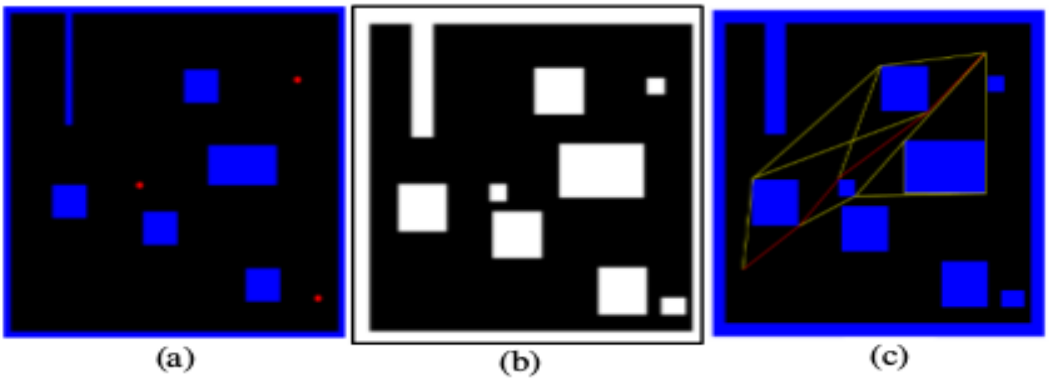


Fig 2-2. Functioning of corner passing method: (a) image of obstacles, (b) binary shape of obstacles, (c) planned path using the method (the red path is the optimal one) [15]

2.2.2. Rapidly optimizing mapper (ROM)

Rapidly optimizing mapper (ROM) is a solution for a holonomic off-line point robot in a static environment. ROM is able to plan a collision free route from the starting point to the goal configuration using series of different methods and parameters to analyze the workspace and compute the shortest collision-free trajectory.

The strategy of ROM focuses on fewer numbers of obstacles (roadblock obstacles) instead of all available obstacles in the environment in order to achieve a better performance and construct an optimal trajectory [16].

The Rapidly optimizing mapper path planner [17] consists of four different and distinct phases as follows:

- *Initial phase*: computing sets of primitive variables related to the SD (standoff distance), DOT **(**Degree of Traverse**),** and DOST **(**Degree of surface traversal).
- *Workspace analyzer*: analyzing the workspace to determine roadblock obstacles, and roadblock obstacle surface scanning.
- *Graph (Roadmap) builder*.
- *Shortest path computation unit*: analyzing the graph constructed and using the Dijkstra shortest path algorithm.

2.2.3. Lattice planner method

Lattice planner consists of a set of states, connected by edges. To construct the set of states, each dimension in the planning domain is discretized into cells of finite size [18] [19]. Typically, metric dimensions are divided into small grids; it is represented as a series of connected states in a lattice-based graph constructed using motion primitives and efficient search algorithms.

A lattice-based graph is a graph constructed using short motion primitives as edges that end up at the center of cells. Motion primitives are short, kinematically feasible motions which form the basis of movements that can be performed by the robot platform [20].

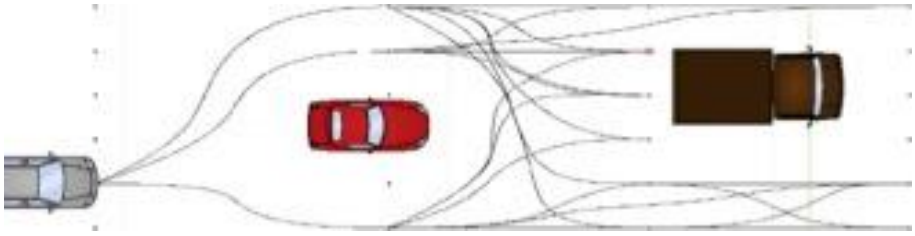


Fig 2-3. Lattice planner for on-road driving

2.2.4. Rapidly-exploring random tree

A Rapidly-exploring Random Tree (RRT) is a data structure and algorithm that is designed for efficiently searching non-convex high-dimensional spaces [21]. RRTs are constructed incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree. RRTs are particularly suited for path planning problems that involve obstacles and differential constraints (non-holonomic or kino-dynamic). RRTs can be considered as a technique for generating open-loop trajectories for non-linear systems with state constraints [22].

RRT constructs a tree using random sampling in search space. The tree starts from an initial state and expands to find a path toward the goal state. The tree gradually expands as the iteration continues. During each iteration, a random state is selected from configuration space (Figure 2-4) [23].

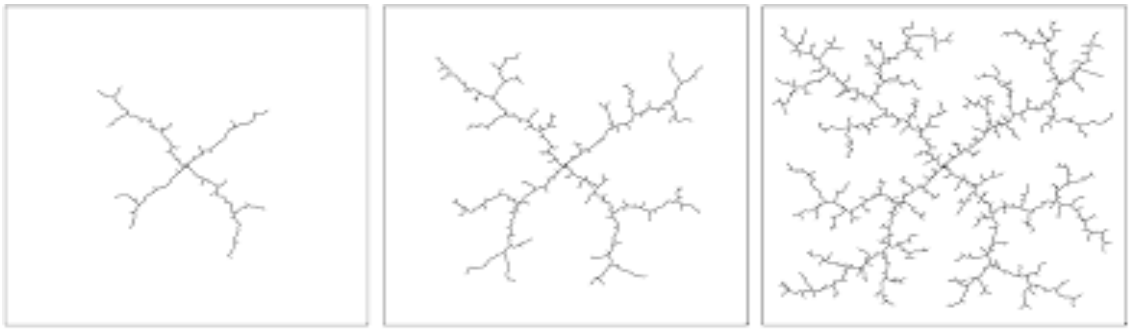


Fig 2-4. Incremental build of a rapidly exploring random tree (RRT)

2.3. Advantages and disadvantages of each method

The main advantages and drawbacks of the approaches we studied in this chapter can be outlined as follows:

Methods	Advantages	Disadvantages
Rapidly-exploring random trees	-Quick search of free space. -Advanced decision techniques are applied for collision checking. -Optimality in the path is guaranteed in newer implementations such as RRT*. -Real-time feasibility.	-Each node of the tree needs to be checked for collisions while the tree is expanding.
Rapidly optimizing mapper (ROM)	-Optimality. -Use few numbers of obstacles.	-Very close to obstacles.
Corner passing method	-Easy algorithm. -Optimality in the path guaranteed.	-Requires a lot of computing. -Computation complexity. -Pure off-line method.
Lattice planner	-Low computational power needed. -Smoothness and optimality of the path are guaranteed.	-Time inefficiency with the calculation of a path. -May lead to exhaustive sampling or oscillations.

Table 2-1. Advantages and disadvantages of the studied path planning approaches

2.4. Conclusion

In this chapter, we have discussed different path planning algorithms for autonomous mobile robot; furthermore, we described their advantages and drawbacks. This allowed us to select the best algorithm to work with in this project. At the end, we adopted the Rapidly-exploring random trees for the implementation of our path planner.

3.1. Introduction

Mobile robots navigation includes different interrelated activities described previously in the first chapter. The perception has been done using the data acquired by two Kinect cameras, which is then preprocessed to get an environment model. The mobile robot position is estimated using the data obtained from the encoders.

The path planning is performed by the fastest and usable method *Rapidly-exploring Random Tree* algorithm which is implemented using *Robot Operating System*.

3.2. Robot Operating System

The *Robot Operating System (ROS)* is a flexible framework for writing robot software. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms [24]. *ROS* was built from the ground up to encourage collaborative robotics software development. It was designed specifically to collaborate and build upon each other work. Software in the ROS Ecosystem [25] can be separated into three groups:

- Language- and platform-independent tools used for building and distributing ROS-based software.
- ROS client library implementations such as roscpp, rospy, and roslisp.
- Packages containing application-related code which uses one or more ROS client libraries.

3.3. Environment perception

The Kinect has two sensors, a color sensor and a depth sensor. To enable independent acquisition from each of these devices, they are treated as two independent devices in the *image acquisition*. Those devices return four data streams:

- The image stream contains color data in RGB format with resolution of 640x480, frame rate of 30 frames per second.

- The depth stream returns depth information in pixels, with resolution of 640x480, frame rate of 30 frames per second with a valid range of 50cm to 400cm.

The skeletal stream is returned by the depth sensor and returns data about the skeletons. There is also an audio stream, but those are unused in our work.

The management of the Kinect camera and its functionality is ensured by the package *openni-launch*, which contains launch files and nodes for using OpenNI-compliant devices (Open Natural Interaction)[1].

Fig 3-1. Acquired RGB color and depth images (Depth in mm).

3.3.1. Data preprocessing

The objective of this phase is to obtain a binary safe map where the robot can evolve without colliding possible obstacles inside its environment.

In order to perform this phase, we need to use the **Open Source Computer Vision** (**OpenCV**) which is a library used for image processing. It is mainly used to do all the operations related to images.

The technique that we have used for generating a binary image by using static cameras is **Background Subtraction (BS).** Background Subtraction calculates the foreground mask performing a subtraction between the current frame and a background model, containing the static part of the scene.

The background modeling consists of two main steps: (*i*) Background Initialization and (*ii*) Background Update. In the first step, an initial model of the background has computed; while in the second step that model has updated in order to

[1] *OpenNI* is an open-source framework for "natural interaction" - using your hands and body to interact.

adapt to possible changes in the scene. The following function has been used to implement this data processing:

cv::BackgroundSubtractorMOG

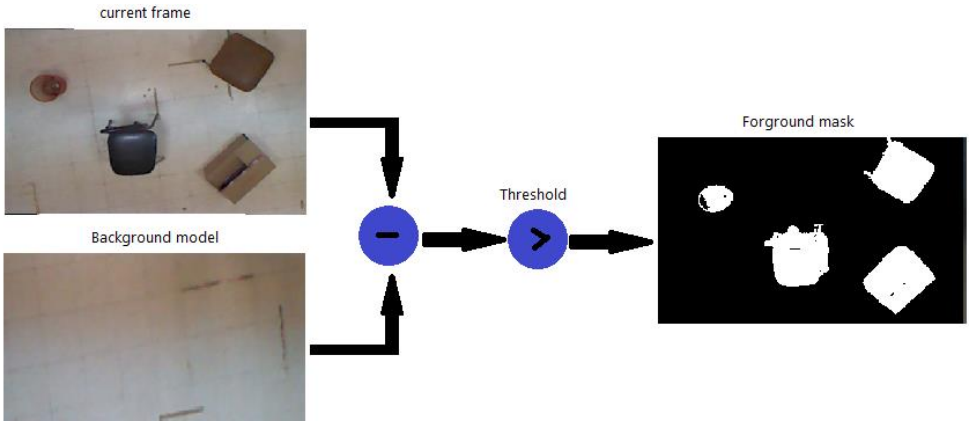


Fig 3-2. Example of background subtraction

However, before using OpenCV we need to interface it with ROS using **CVBridge** and **image_transport** package. The first package is used to convert between ROS Image messages and OpenCV images; while the second one provides transparent support for transporting images in low-bandwith compressed format.

```
cv_bridge::CvImageConstPtr image;
image=cv_bridge::toCvCopy(msg ,sensor_msgs::image_encodings::TYPE)
```

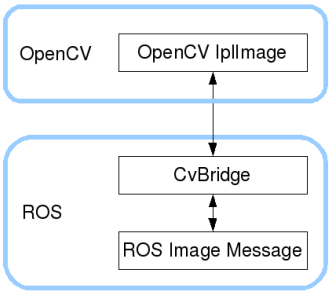


Fig 3-3. Interfacing ROS with OpenCV

Using distributed threshold, an environment model is obtained in the form of a binary (logical) map that could be used by the robot; it represents the obstacles and the free space environment.

3.3.2. Data processing and feature extraction

For the mobile robot to navigate successfully without collisions, and due to the uncertainties of its sensors, a safe region could be added to the binary map using a disk

mask. Thus, we obtain a safe map representing three areas: (*i*) forbidden area (obstacles), (*ii*) dangerous area (near the obstacles), and (*iii*) safe area.

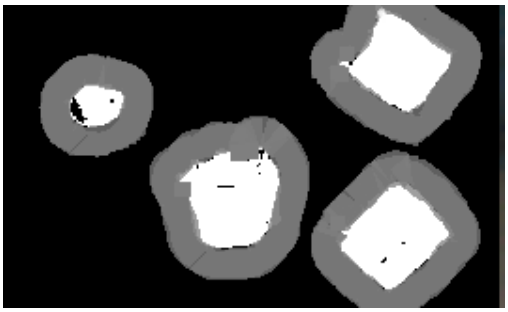


Fig 3-4. Safe map representing the three areas.

3.4. Visualization in RVIZ

RVIZ is a 3D visualizer for displaying sensor data and state information from *ROS*. Using *rviz,* we can visualize the current configuration on a virtual model of the robot, etc. We can also display live representations of sensor values coming over ROS topics including camera data, laser distance measurements, etc.

In order to visualize our binary safe map on RVIZ (Figure 3-5), we follow the two steps described below:

- Find each non-zero pixel coordinate *(x, y)* using *findNonZero* function described in the line code:

  ```
  cv::Mat nonZeroCoordinates;
  cv::findNonZero(image, nonZeroCoordinate);
  ```

- Display the coordinates found using *visualization_msgs/Marker*[2].

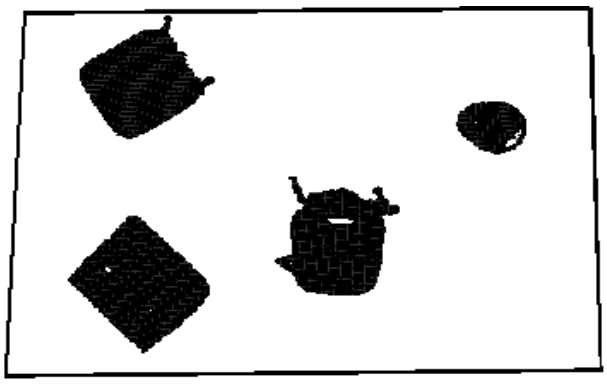


Fig 3-5. Visualization on RVIZ.

[2] The marker message is used to send visualization "markers" such as boxes, spheres, arrows, lines, point, etc. to a visualization environment such as *rviz*.

3.5. Kinematic analysis of the mobile base (odometry localization)

The kinematic analysis of the mobile robot needs to focus on the following main reference frames:

- $R_A = (O_A, \overrightarrow{x_A}, \overrightarrow{y_A}, \overrightarrow{z_A})$: Absolute reference frame.
- $R_B = (O_B, \overrightarrow{x_B}, \overrightarrow{y_B}, \overrightarrow{z_B})$: Mobile base reference frame.

During its motion, the mobile robot odometry is calculated in real time. Indeed, a relationship between the absolute reference frame of the plane and the local reference frame of the robot is established in order to specify its position and orientation angle on the plane.

The axes X_i and Y_i in figure 3.6(a) define an arbitrary inertial basis on the plane as the global reference frame from some origin O:{X_i, Y_i}. To specify the position of the robot, choose a point P on the robot chassis as its position reference point. The basis {X_R, Y_R} defines two axes relative to P on the robot chassis and is thus the robot's local reference frame. The position of P in the global reference frame is specified by coordinates x and y, and the angular difference between the global and local reference frames is given by θ. The position of the robot can be described as a vector with these three elements. Note the use of the subscript I to clarify the basis of this position as the global reference frame (eq. 3.1)

$$\xi_I = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \tag{3.1}$$

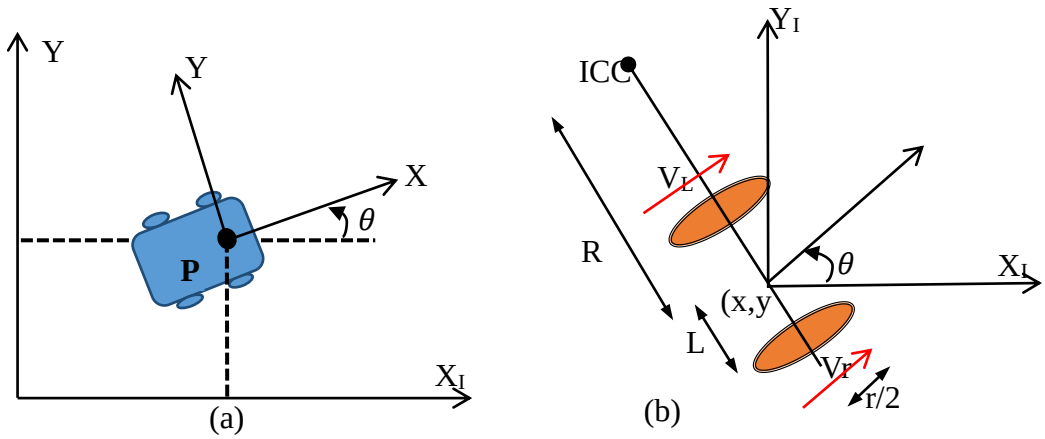


Fig 3-6. (a) The global reference frame. (b) The robot wheels Kinematics.

To describe robot motion in terms of component motions, we map motion along the axes of the global reference frame to motion along the axes of the robot's local

reference frame. The mapping is a function of the current pose of the robot. This mapping is accomplished using the rotation matrix:

$$R(\theta) = \begin{bmatrix} \cos(\Theta) & \sin(\Theta) & 0 \\ -\sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}, R^{-1}(\theta) = \begin{bmatrix} \cos(\Theta) & -\sin(\Theta) & 0 \\ \sin(\Theta) & \cos(\Theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

The differential mobile robot has two wheels, each with diameter r. P is centered between the two drive wheels; each wheel is at a distance l from P. Given r, l, P, and the speed of each wheel, $\dot{\varphi}_r$ and $\dot{\varphi}_l$, a forward kinematic model would predict the robot's overall speed in the global frame.

The Instantaneous Center of Curvature,

$$ICC = [x - R\sin\theta\,, y + R\cos\theta] \tag{3.3}$$

Noting that the right velocity $V_r = \omega(R + l)$, and left velocity $V_l = \omega(R - l)$ where:

$$R = l\,\frac{V_r + V_l}{V_r - V_l}$$

$$\omega(Angular\ velocity) = \frac{V_r - V_l}{2\,l}$$

$$V(Linear\ velocity) = R\omega = \frac{V_r + V_l}{2} \tag{3.4}$$

Thus, the component motions are given by:

$$\begin{bmatrix} \dot{X}\mathrm{I} \\ \dot{Y}I \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{V_r + V_l}{2}\cos\theta \\ \frac{V_r + V_l}{2}\sin\theta \\ \frac{V_r - V_l}{2l} \end{bmatrix} \tag{3.5}$$

Approximating $\dot{\mathrm{X}}_\mathrm{I}(t) \approx \frac{\Delta X}{\Delta T}$, and all the other derivatives, while ΔT is very small, the curve between two positions is approximated by a line-segment with constant angle $\theta + \frac{\Delta\theta}{2}$. ΔS_r and ΔS_l are the traveled distances for the right and left wheels, respectively.

$$\begin{bmatrix} \Delta X_I \\ \Delta Y_I \\ \Delta\theta \end{bmatrix} = \begin{bmatrix} \Delta S\cos(\theta + \Delta\theta/2) \\ \Delta S\sin(\theta + \Delta\theta/2) \\ (\Delta S_{r-}\,\Delta S_l)/2l \end{bmatrix} = \begin{bmatrix} \frac{\Delta S_r + \Delta S_l}{2}\cos(\theta + \Delta\theta/2) \\ \frac{\Delta S_r + \Delta S_l}{2}\sin(\theta + \Delta\theta/2) \\ \frac{\Delta S_r - \Delta S_l}{2l} \end{bmatrix}$$

$$\xi_I(k+1) = \xi_I(k) + \Delta\xi_I = \begin{bmatrix} X_I \\ Y_I \\ \theta \end{bmatrix} + \begin{bmatrix} \frac{\Delta S_r + \Delta S_l}{2}\cos(\theta + \Delta\theta/2) \\ \frac{\Delta S_r + \Delta S_l}{2}\sin(\theta + \Delta\theta/2) \\ \frac{\Delta S_r - \Delta S_l}{2l} \end{bmatrix} \quad (3.6)$$

3.6. Path planning using the Rapidly-exploring Random Tree (RRT) algorithm:

RRTs [26] were proposed as both a sampling algorithm and a data structure designed to allow fast searches in high-dimensional spaces in motion planning. RRTs are progressively built toward unexplored regions of the workspace from an initial configuration. The basic RRT construction algorithms are described below:

- **Algorithm 1**: BUILD(): Construct an RRT that eventually creates a path between q_{init} and q_{goal}.

  Input: Initial configuration q_{init}, goal configuration q_{goal}, number of nodes I_{max} incremental distance ε; **Output**: Path $q_{init}\ldots q_{goal} \subseteq$ RRT tree τ**{**

  ```
  τ.INIT(q_init);
  for i=1 to Imax{
      q_rand←BASED_RANDOM_CONFIG();
      if (EXTEND(τ, q_rand, ε)=Solved) {
          // Return found solution
          return τ.Path(q_init, q_goal);
      }
  }
  // No solution found within I_max nodes max
  return No Solution;
  }
  ```

- **Algorithm 2**: EXTEND(): Extend τ towards q_{rand} with distance ε and check

  Input: random node q_{rand}, goal node q_{goal}, RRT tree τ, incremental distance ε; **Output**: Status**{**

  ```
  q_near←NEAREST_NEIGBOUR(q_rand ,τ);
  q_new←NEW _CONFIG(q_near, q_rand, ε);
  if (q_new!=NULL){
      τ.ADD_NODE(q_new);
      τ.ADD_EDGE(q_near, q_new);
      if (CHECK_FINISHED(q_new, q_goal)){
          //q_new connects to q_goal, so we have a complete path
          τ.ADD_EDGE(q_new, q_goal);
          return Solved;
      }
      else{
          if (q_new=q_rand){
              q_rand reached
          }
  ```

else{

extended towards q_{rand}

}

}

}

At every step, a random *q_rand* configuration is chosen and for that configuration the nearest configuration already belonging to the tree *q_near* is found. For this, a definition of distance is required (in motion planning, the Euclidean distance is usually chosen as the distance measure). When the nearest configuration is found, a local planner tries to join *q_near* with *q_rand* with a limit distance. If *q_rand* was reached, it is added to the tree and connected with an edge to *q_near*. Otherwise, the configuration *q_new* obtained at the end of the local search is added to the tree in the same way as long as there was no collision with an obstacle during the search. This operation is called the *Extend step*, illustrated in Figure 3-7. This process is repeated until some criteria is met, like a limit on the size of the tree.

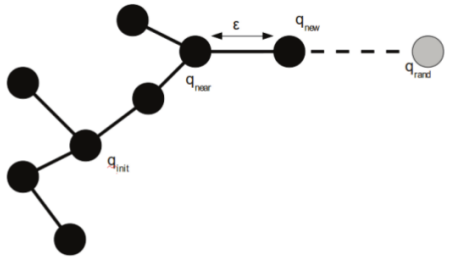


Fig 3-7. Extend phase of an RRT.

Once the RRT has been built, multiples queries can be issued. For each query, the nearest configurations belonging to the tree to both the initial and the goal configuration are found. Then, the initial and final configurations are joined to the tree to those nearest configurations using the local planner and a path is retrieved by tracing back in the tree structure.

3.6.1. Adding new node

In the beginning, the graph is empty. We have only the starting and goal positions. The aim is to create random tree from the starting position to the goal position. First, we should create a random node and add it to the graph. A simple iteration in performed; each step attempts to extend the RRT by adding a new vertex that is biased by a randomly-selected state. The algorithm selects the nearest vertex already in the RRT to the given state by calculating the Euclidean distance.

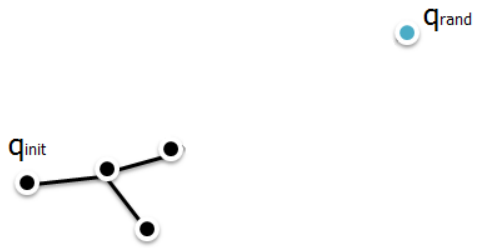


Fig 3-8. Adding new vertex.

3.6.2. Checking the step size

Another important process is checking the distance between new vertex and the vertex that is the closest one in RRT; this is done because the distance of the new vertex might be larger than the step size that is given. The lines between nodes cannot be larger than the step size value. If the distance is not larger than the step size, we can go forward to the next step. Otherwise, we should create a new vertex which is step size away from the closest vertex; then, go the next step.

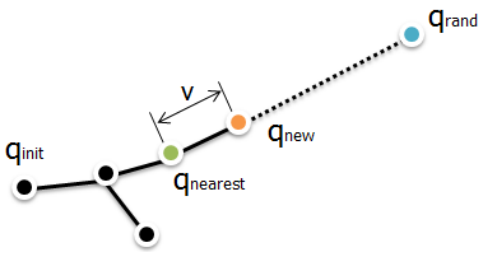
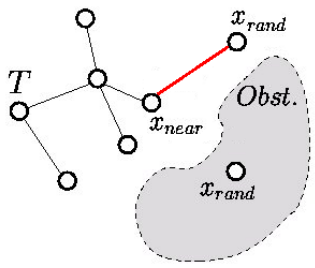


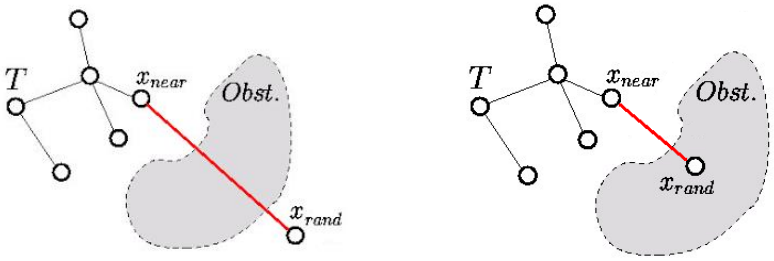
Fig 3-9 Checking the distance between *q_near* and *q_rand*

3.6.3. Checking the intersection

During the creation of the tree, the obstacles must be avoided by checking whether a given state lies inside the obstacle or not. Furthermore, each edge of the RRT will correspond to a path that lies entirely in free region (Figure 3-10).



(a) Outside the obstacle, there is not an intersection



(b) Outside the obstacle (c) Inside the obstacle, but there is an intersection

Fig 3-10. Checking the intersection between *q_near* and *q_rand*.

3.7. Implementation of RRT in ROS

In the previous section, we have seen how to get the image of our environment from a Kinect. Now, in this part we will see how to implement the algorithm of RRT described previously using ROS.

First, we need to create a workspace where a new package will be build. Once we get the package, two nodes will be created: (*i*) the first for the visualization of the environment and (*ii*) the second for the creation and visualization of the tree. Each node has been written in C++ code.

The visualization is done in RVIZ with the following parameters:

- Frame_id=”/path_planner”
- marker_topic=”path_planner_rrt”

Before starting the creation of the two nodes we have to construct two header files *obstacles.h* and *rrt.h*. These two files will contain the definitions of functions, global structure and variables, which will be imported or used into C++ program by using the pre-processor.

- *obstacles.h* contains one function *getObstacleArray()* and one variable.
- *rrt.h* comprises the structure of *rrtNode* and eleven different functions needed for the construction of the tree.

After the creation of the header files, we will start the implementation of the nodes:

- *env_node*: this node will include only one *cpp* file; the following flowchart describe its code (Figure 3-11):

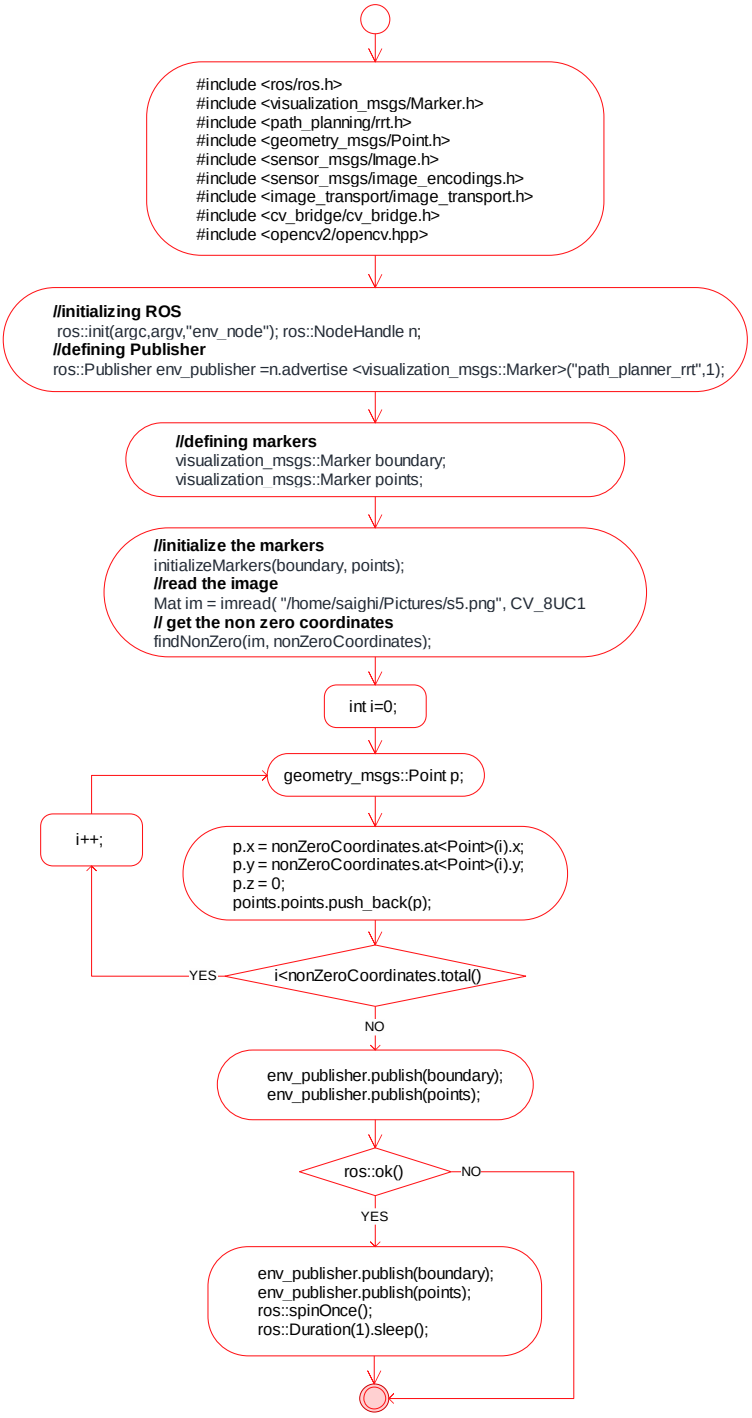
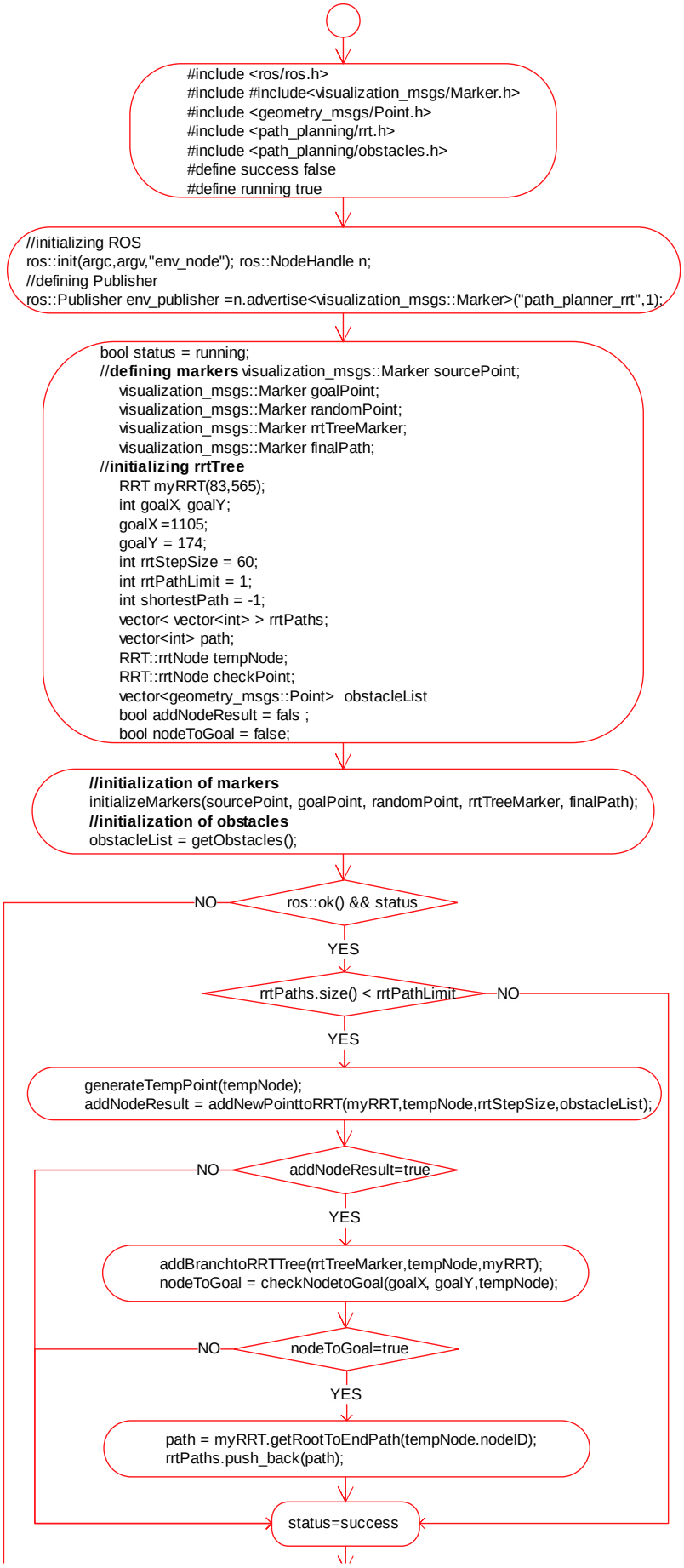


Fig. 3-11. Flowchart describing the *env_node*

- *rrt_node*: this node will include four *cpp* file:
  - *obstacle.cpp* contains the position of the obstacles.
  - *rrt.cpp*.
  - *rrt_node*: the following flowchart describes its code (Figure 3-12).


#include <ros/ros.h>
#include #include<visualization_msgs/Marker.h>
#include <geometry_msgs/Point.h>
#include <path_planning/rrt.h>
#include <path_planning/obstacles.h>
#define success false
#define running true
//initializing ROS
ros::init(argc,argv,"env_node"); ros::NodeHandle n;
//defining Publisher
ros::Publisher env_publisher =n.advertise<visualization_msgs::Marker>("path_planner_rrt",1);
bool status = running;
//defining markers visualization_msgs::Marker sourcePoint;
visualization_msgs::Marker goalPoint;
visualization_msgs::Marker randomPoint;
visualization_msgs::Marker rrtTreeMarker;
visualization_msgs::Marker finalPath;
//initializing rrtTree
RRT myRRT(83,565);
int goalX, goalY;
goalX =1105;
goalY = 174;
int rrtStepSize = 60;
int rrtPathLimit = 1;
int shortestPath = -1;
vector< vector<int> > rrtPaths;
vector<int> path;
RRT::rrtNode tempNode;
RRT::rrtNode checkPoint;
vector<geometry_msgs::Point> obstacleList
bool addNodeResult = fals ;
bool nodeToGoal = false;
//initialization of markers
initializeMarkers(sourcePoint, goalPoint, randomPoint, rrtTreeMarker, finalPath);
//initialization of obstacles
obstacleList = getObstacles();
ros::ok() && status
NO
YES
rrtPaths.size() < rrtPathLimit
NO
YES
generateTempPoint(tempNode);
addNodeResult = addNewPointtoRRT(myRRT,tempNode,rrtStepSize,obstacleList);
addNodeResult=true
NO
YES
addBranchtoRRTTree(rrtTreeMarker,tempNode,myRRT);
nodeToGoal = checkNodetoGoal(goalX, goalY,tempNode);
nodeToGoal=true
NO
YES
path = myRRT.getRootToEndPath(tempNode.nodeID);
rrtPaths.push_back(path);
status=success

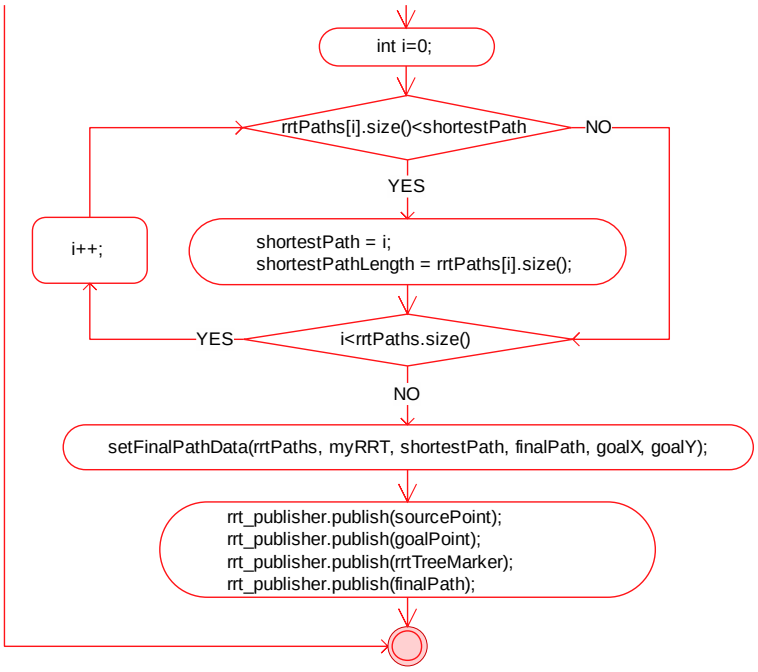


Fig. 3-12. Flowchart describing the *rrt_node*

Now, we have written our nodes, we need to build them. We open up the *CMakeLists.txt* file and add the following lines to the bottom of the file (where *path_planning* is the name of the package):

```
add_executable(talker src/talker.cpp)
target_link_libraries(env_node ${catkin_LIBRARIES})
add_dependencies(env_node path_planning_cpp)
add_executable(rrt_node src/listener.cpp)
target_link_libraries(rrt_node ${catkin_LIBRARIES})
add_dependencies(rrt_node path_planning _cpp)
```

Next, we save the file and build the package using *Catkin_make* line. Finally, we run the two nodes created with *RVIZ* (Figure 3-13):

- `$ rosrun rviz rviz`
- `$ rosrun path_planning env_node`
- `$ rosrun path_planning rrt_node`

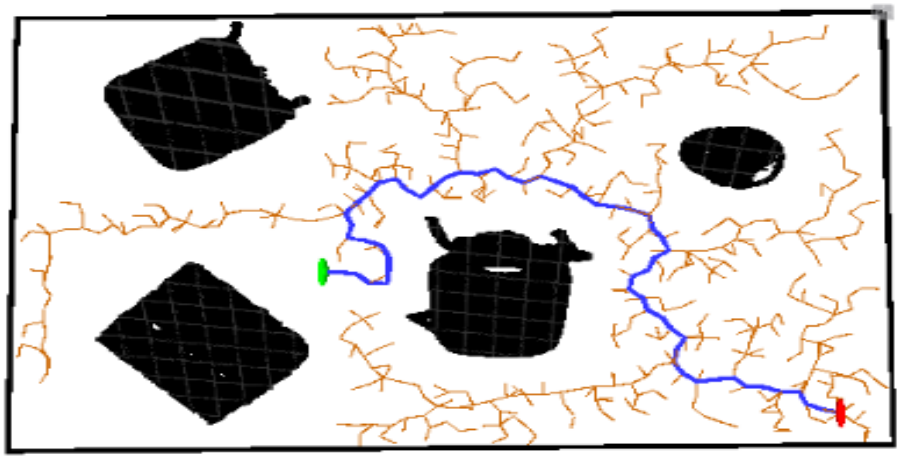


Fig 3-13. Execution of the nodes and visualization on RVIZ

3.8. Conclusion

Image acquisition and image processing is mainly used to implement real-time perception and localization; unfortunately, we could not apply it on localization because of a technical problem found on the RobuTER. The *Rapidly-exploring Random Tree* algorithm is implemented to generate a fast safe path.

4.1 Introduction

The techniques and strategies described in the previous chapters are applied on the differential drive industrial mobile robot RobuTER described in appendix C. It is available at the Division of Computer-Integrated Manufacturing and Robotics (DPR) where we did our implementations.

As shown in Fig 4-1, two Kinect cameras are fixed on the roof of the experimental workroom; each Kinect is placed at a height of 3500mm. They are placed in such a manner with the same axis to visualize both the environment (ground, obstacles, etc.) and the mobile robot (RobuTER). The Kinect cameras are connected to a host PC in order to acquire and process images of the scene.

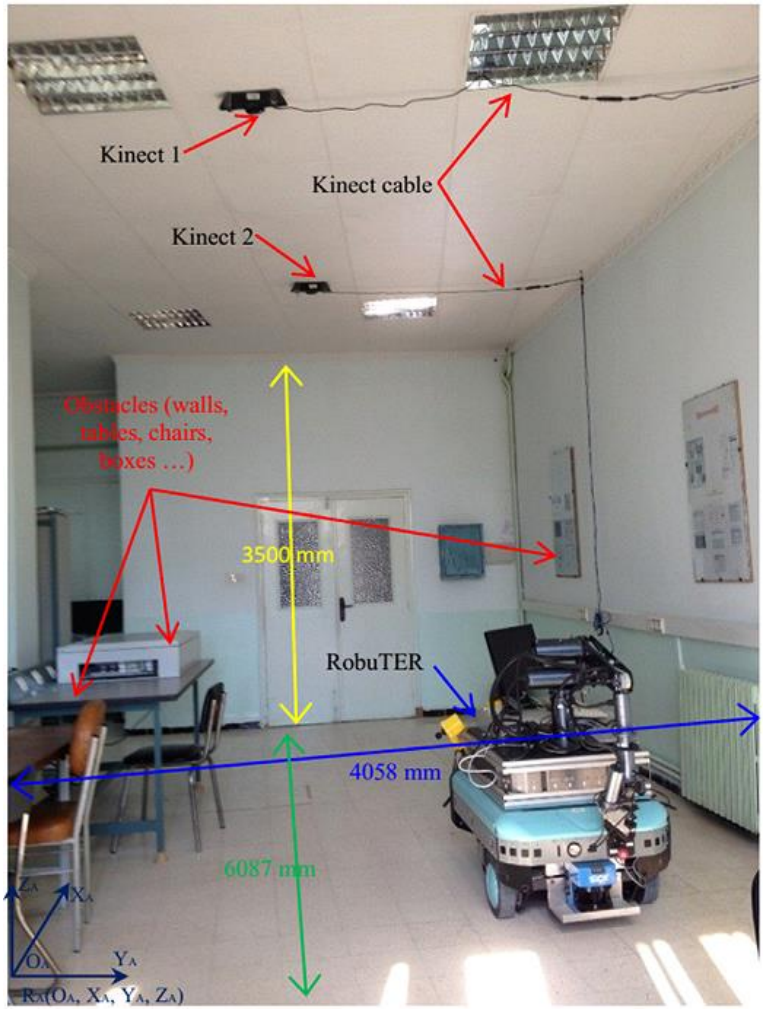


Fig 4-1. Experimental robotic testbed.

In order to compare the results of our work and what it was done previously, we have chosen the same environment with the same conditions as in [27].

4-2. Environment modeling

The visual perception of the robot environment is carried out using two Microsoft Kinect cameras system Version 1. The Kinect cameras have the following parameters:

Sensitivity	**1**
Linearity	linear
Measurement range	[500 mm, 4000 mm] 3500 mm
Error	1 mm
Noise	Modeled (8 pixels); Non-modeled
Resolution	640x480 and 30 frames per second
Type of output	uint<640x480x3> for RGB uint<640x480> for depth

Table 4-1. Kinect characteristics.

The pictures taken from the first Kinect camera (RGB and depth images) are shown below:

Fig 4-2. RGB and depth images acquired by the first Kinect.

The images delivered by the second Kinect camera (RGB and its depth images) are shown as follows:

Fig 4-3. RGB and depth images acquired by the second Kinect.

The Kinect sensor acquisition and the perception techniques give an exact binary map model for the environment with a sufficient resolution of 960x640.

Fig 4-4. Binary map of the overall environment.

And here is the binary safe map:

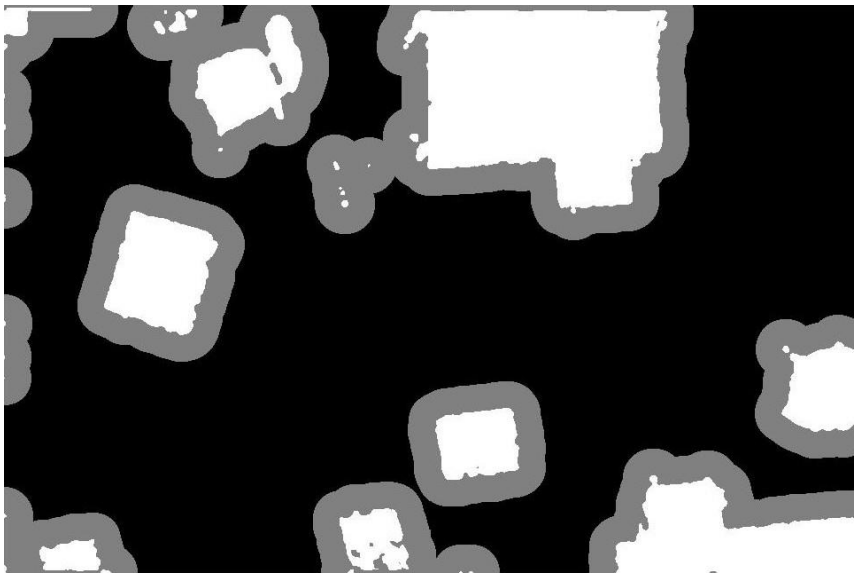


Fig 4-5. Binary safe map of the overall environment

4-3. Visualization on RVIZ

The environment has been visualized on *Rviz* using nonzero function to apply the algorithm on it and find the feasible paths. After executing the environment node on *ROS* and adding the marker, we got (Fig. 4-6):

```
$ rosrun path_planning env_node
```

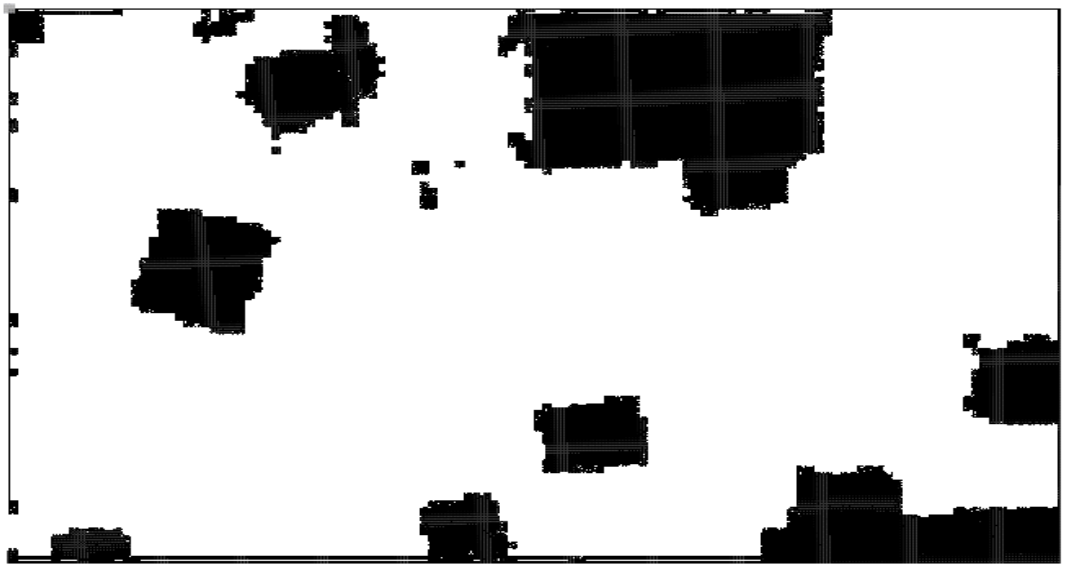


Fig 4-6. Visualization of the environment on RVIZ

4-4. Trajectory planning

The obtained binary map gives an almost continuous description of the environment; this is well suited to be used by the rapidly-exploring random trees.

The mobile robot RobuTER, with dimensions of 1200mm×680mm, has to move from $Source(x, y)_{init}=(74mm, 1953mm)$ toward $Target(x, y)_{fin}=(5501mm, 1308mm)$ with a maximum linear speed of *150mm*/s.

RRT chooses the first feasible path and creates its nodes randomly. Consequently, we have to execute the algorithm many times with different step sizes; finally, we have to choose the best one referring to the execution time, smoothing and optimality.

We have selected three main step sizes: 8 pixel, 35 pixels, and 60 pixels which correspond to 50.72mm, 221.9mm and 380.4mm, respectively.

4-4-1. Step size=08 pixels

The obtained path after executing the algorithm with step size of 8 pixels is shown by Figure 4-7. The RRT path consists of 144 segments. The path is generated in around 15sec; its total length is l=7303.68mm.

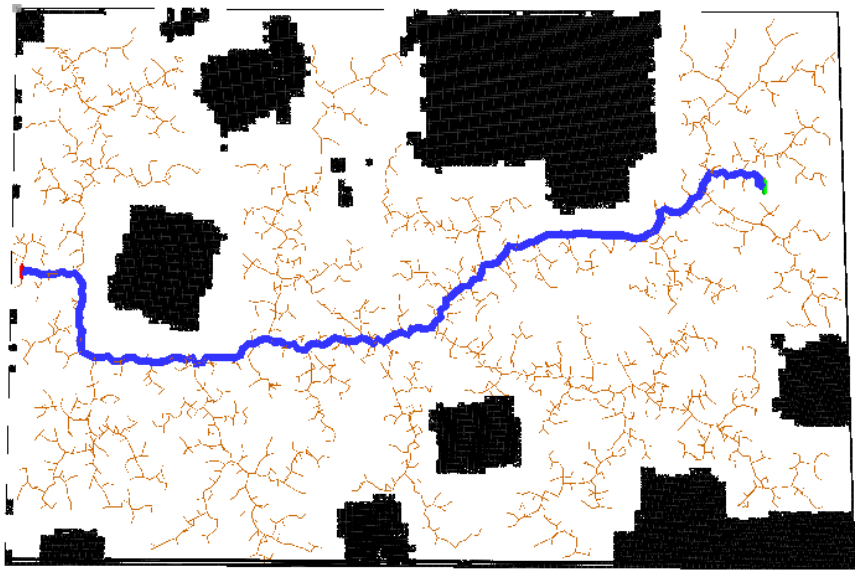


Fig 4-7. RRT planning with a step size of 8 pixels

4-4-2. Step size=35 pixels

After execution with 35 pixels, we got the path shown in Figure 4.8. The path consists of 33 segments. It is generated in 2.16 sec; its total length *l*=7100.8mm.

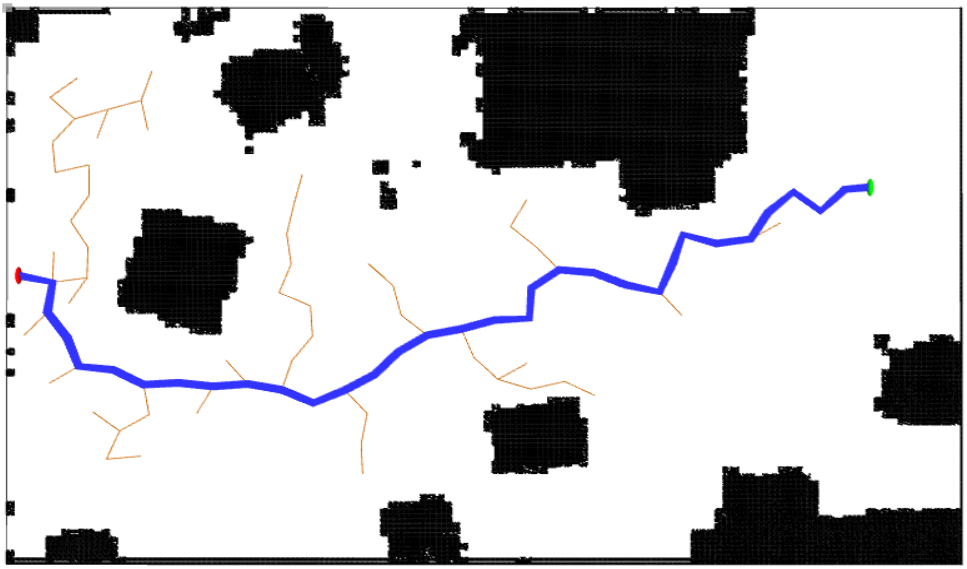


Fig 4-8. RRT planning with a step size of 35 pixels

4-4-3. Step size=60 pixels

The resulted path is given by Figure 4-9. Using a step size of 60 pixels, the RRT path consists of 18 segments. The path is generated in less than 2sec; its total length *l*=6847.2mm.

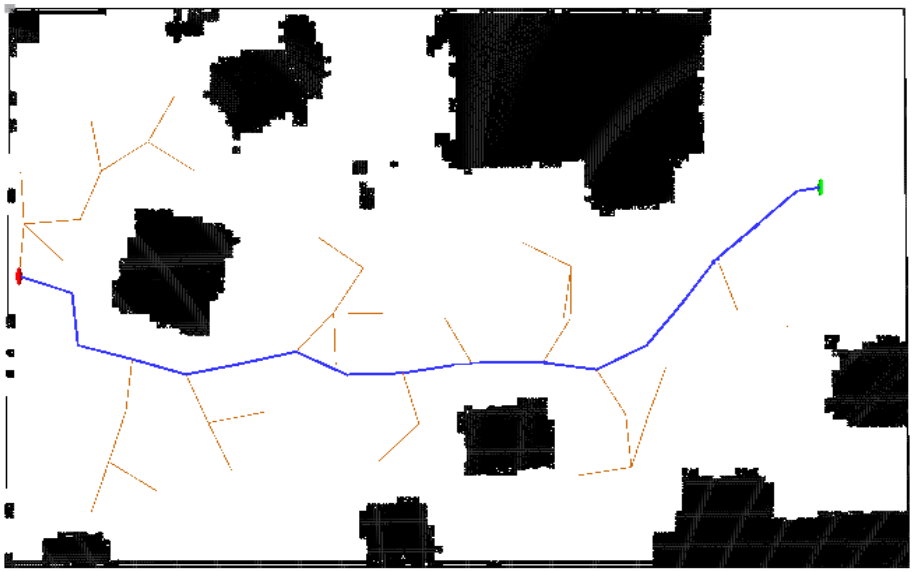


Fig 4-9. RRT planning with a step size of 60 pixels

The following table presents the summary of the obtained results:

Step size	8 pixels	35 pixels	60 pixels
Length	7303.68	7100.8	6847.2
N° of segments	144	33	18
Run time	15	2.4	1.9

Table 4.2. Summary of executions

We can clearly see that executing the RRT with a step size=60 pixels gives the shortest path comparing with 8 and 35 pixels. In addition, it is much faster and gives less number of segments. Consequently, we opted for this step size in what follows.

4-5. Path smoothing

We can see that the generated paths are zigzag lines; so, to get more efficient results we have used *Piecewise Cubic Hermite Interpolating Polynomial (PCHIP)* in order to smooth our path as shown below in Figure 4-10:

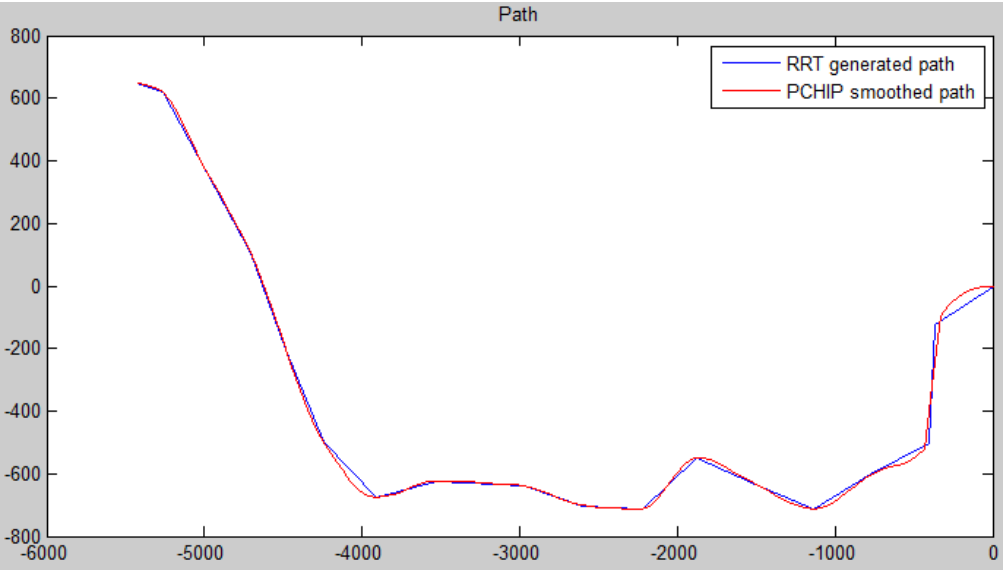


Figure 4-10. RRT generated path and its PCHIP smoothed path

4-6. Path execution

The obtained path in the previous section must be carried out by the differential mobile robot; thus, $V(t)$ and $\omega(t)$ are evaluated resulting in unique $V_r(t)$ and $V_l(t)$ from equation (3.4) stated in chapter 3.

The required $V_r(t)$ and $V_l(t)$ are sent from the computer (acting as a client) to the mobile robot periodically using one socket; the embedded PC of the mobile robot on the other side is acting as a server, where the data is received, decoded, and then executed.

The graphs of variation of the right wheel, left wheel and linear velocities sent to the mobile robot are shown below in Figure 4-11:

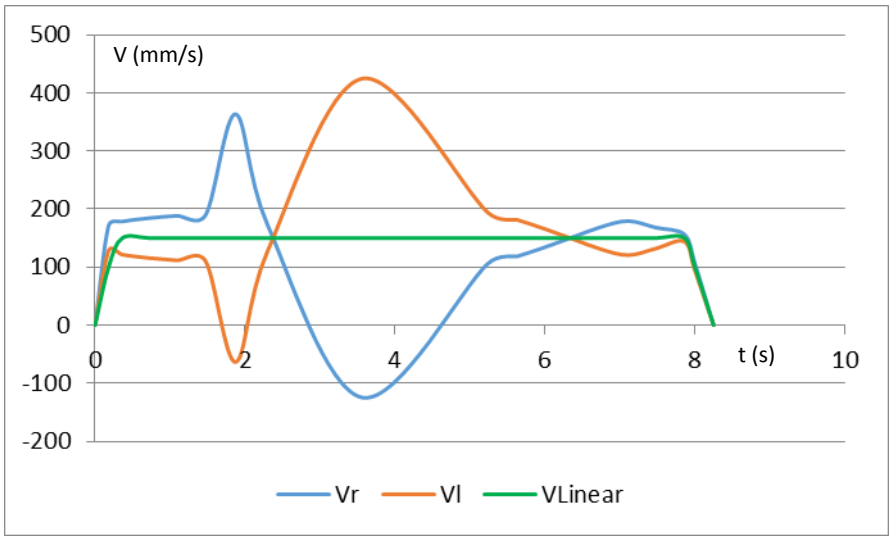


Figure 4-11. Variation of the robot right wheel and left wheel velocities for path execution

4-7. Comparison

We have mentioned before that the same work has been done using another algorithm in the same environment and same conditions [27]. In this part, we will do a comparison between the results of the two algorithms.

Table 4.3 shows the results of executing the RRT and GA algorithms on the same environment.

Parameter	RRT	GA [1]
Run time (s)	2	7
Path length (mm)	6847.2	6020
Number of segments	18	18

Table 4.3 Summary of comparative analysis

From Table 4.3, we noted firstly that RRT takes longer distance from initial point to the goal comparing with genetic algorithm. Secondly, to generate the path connecting the two initial and goal position, RRT is three times faster that the GA algorithm. Finally, both algorithms generate paths with the same number of segments.

Another comparison is done regarding the calculation times for different workspaces with different dimensions and number of obstacles. For each case, we considered different numbers of Kinect cameras with various Source-Goal positions. Table 4.4 summarizes the average for 10 different runs of RRT and GA algorithms.

Resolution (accuracy, area size)	**Number of obstacles**	**Time generations(s) of feasible path**	
		GA [1]	RRT
640×480 (01 Kinect camera)	05	5.09	0.80
	10	5.59	1.04
	15	6.46	1.25
640×960 (02 Kinect cameras)	10	6.33	1.54
	15	8.03	1.78
	20	8.47	2.11
640×1280 (04 Kinect cameras)	15	9.62	3.03
	20	10.6	3.2
	25	10.96	3.51

Table 4.4 Summary of the average calculation time of 10 different runs with different Source-Goal positions

From Table 4.4, it is clear that RRT is better and more efficient algorithm compared to GA proposed in [27]. The run time of RRT in the different environments is around three times faster than GA.

4-8. Conclusion

This chapter is a combination of implementations of the techniques described in chapters two and three. In addition, a general description of the implementation is given, and some path planning results have been shown with their development steps. At the end a comparative summary between RRT and GA has been done.

By using RRT and especially with the fast perception technique used in this work, we can pass from off-line planning to on-line planning.

Conclusion

The aim of this project was to implement a kinect-based path planning and execution for autonomous mobile robot, so the robot should be able to achieve several tasks by itself including the perception of its surrounding environment and finding and executing its path.

At the beginning we have stated some generalities about autonomous navigation for mobile robot and the different techniques performed for path planning strategy. Robot operating system is used as framework to write the mobile robot software because of its simplicity and the packages on it, C++ is integrated with ROS ensuring fast and reliable data transferring, while RVIZ is used for visualizing and showing the work.

Initially, some image acquisition and processing techniques were implemented, allowing the mobile robot to model its environment by constructing the binary map through image processing techniques, the rapidly-exploring random trees has implemented to generate the suitable path starting from an initial position going to the end position with less execution time possible, ensuring that the robot will go safely and rapidly. This is accompanied by showing illustrative examples. The result of this algorithm has shown that it is fast, reliable and safe.

After comparing this work with what it has been done previously with genetic algorithm we found that RRT is three times rapid then the other algorithm.

As a further work, we would develop our work to be evaluated on dynamic rapidly changing environment, RRTstar and RRTstar-smart can be used to implement a more intelligent robot that has the ability of recognition of people and objects, reasoning, learning, and making inferences.

Appendix -A-

A.1. Introduction to Robot Operating System ROS

Robot Operating System (ROS) is a trending robot application development platform that provides various features such as message passing, distributed computing, code reusing, and so on.

The ROS project was started in 2007 with the name Switchyard by Morgan Quigley as part of the Stanford STAIR robot project. The main development of ROS happened at Willow Garage. Here are some of the reasons why people choose ROS over other robotic platforms such as Player,YARP, Orocos, MRPT, and so on :

- ***High-end capabilities***: ROS comes with ready to use capabilities, for example, SLAM (Simultaneous Localization and Mapping) and AMCL (Adaptive Monte Carlo Localization) packages in ROS can be used for performing autonomous navigation in mobile robots and the MoveIt package for motion planning of robot manipulators.
- ***Tons of tools***: ROS is packed with tons of tools for debugging, visualizing, and performing simulation. The tools such as rqt_gui, RViz and Gazebo are some of the strong open source tools for debugging, visualization, and simulation. The software framework that has these many tools is very rare.
- ***Support high-end sensors and actuators***: ROS is packed with device drivers and interface packages of various sensors and actuators in robotics. The high-end sensors include Velodyne-LIDAR, Laser scanners, Kinect, and so on and actuators such as Dynamixel servos. We can interface these components to ROS without any hassle.
- ***Inter-platform operability***: the ROS message-passing middleware allows communicating between different nodes. These nodes can be programmed in any language that has ROS client libraries. We can write high performance nodes in C++ or C and other nodes in Python or Java. This kind of flexibility is not available in other frameworks.
- ***Modularity:*** One of the issues that can occur in most of the standalone robotic applications are, if any of the threads of main code crash, the entire robot application can stop. In ROS, the situation is different, we are writing different nodes for each process and if one node crashes, the system can still work. Also, ROS provides robust methods to resume operation even if any sensors or motors are

- ***Concurrent resource handling***: Handling a hardware resource by more than two processes is always a headache. Imagine, we want to process an image from a camera for face detection and motion detection, we can either write the code as a single entity that can do both, or we can write a single threaded code for concurrency. If we want to add more than two features in threads, the application behavior will get complex and will be difficult to debug. But in ROS, we can access the devices using ROS topics from the ROS drivers. Any number of ROS nodes can subscribe to the image message from the ROS camera driver and each node can perform different functionalities. It reduce the complexity in computation and also increase the debug-ability of the entire system.
- ***Active community***: When we choose a library or software framework, especially from an open source community, one of the main factors that needs to be checked before using it is its software support and developer community. There is no guarantee of support from an open source tool. Some tools provide good support and some tools don't. In ROS, the support community is active. The ROS community has a steady growth in developers worldwide.

A.2. Understanding the ROS file system level

Similar to an operating system, ROS files are also organized on the hard disk in a particular fashion. In this level, we can see how these files are organized on the disk. The following graph shows how ROS files and folder are organized on the disk:

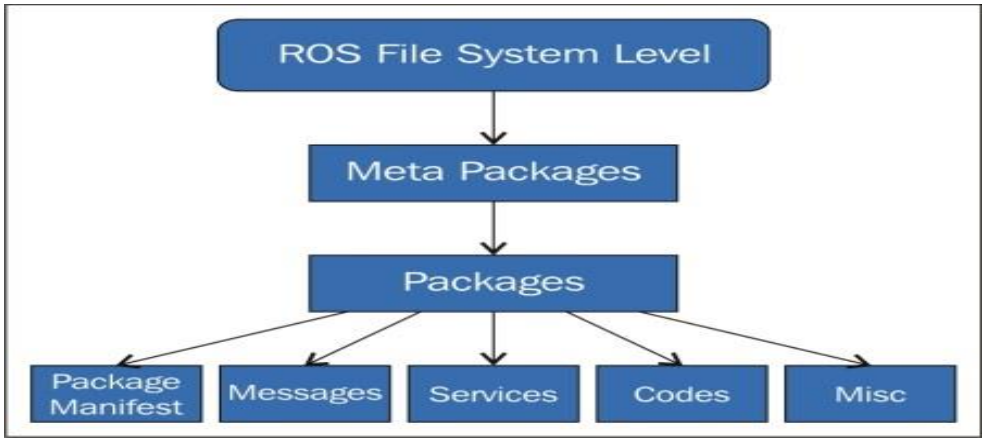


Fig A-1. The ROS file system level

Similar to an operating system, an ROS program is divided into folders, and these

- ***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

runtimeprocess (nodes), configuration files; and so on.

- ***Manifests:*** Manifests provide information about a package, license information, dependencies, compiler flags, and so on. Manifest are managed with a file called *manifests.xml*.
- ***Stacks***: When you gather several packages with some functionality, you will obtain a stack. In ROS, there exists a lot of these stacks with different uses, for example, the navigation
- ***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, stored in my_package/srv/MyServiceType.srv, define the request and response data structures for services in ROS.

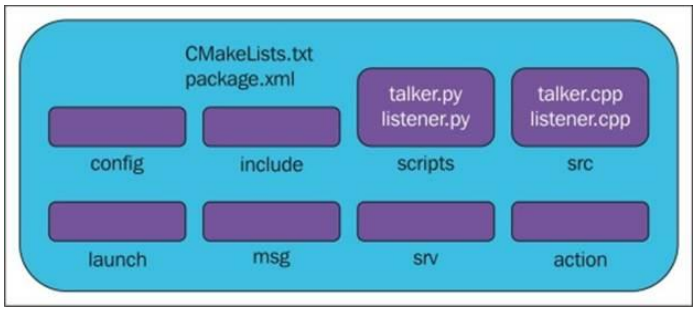


Fig A-2. Structure of typical package

A.3. Understanding the ROS computational Graph level

ROS creates a network where all the processes are connected. Any node in the system can access this network, interact with other nodes, see the information that they are sending, and transmit data to the network.

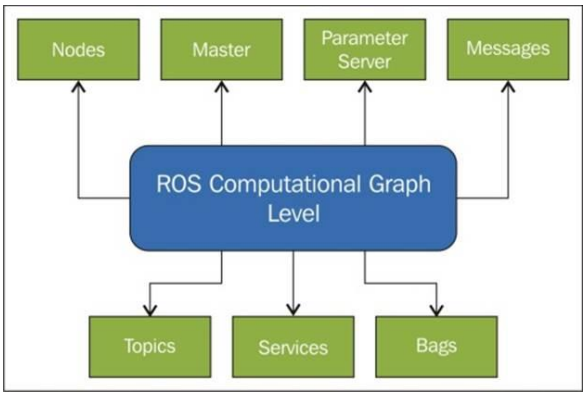


Fig A-2. ROS computational Graph level

- ***Nodes***: ROS nodes are a process that perform computation using ROS client libraries such as roscpp and rospy. One node can communicate with other nodes using ROS Topics, Services, and Parameters.
- ***Topic***: Chanel between two or more nodes, nodes communicate by publishing and/or subscribing to the appropriate topics
- ***Services***: ROS uses a simplified service description language for describing ROS service types. This builds directly upon the ROS msg format to enable request/response communication between nodes. Service descriptions are stored in .srv file in the srv/ subdirectory of a package.
- ***Parameters:*** The Parameter Server gives us the possibility to have data stored using keys in a central location. With this parameter, it is possible to configure a nodes while it's running or to change the working of the nodes.

Appendix -B-

B.1. Installing ROS Indigo:

In this section, you will see the steps to install ROS Electric on your computer.

We assume that Ubuntu repository was successfully installed.

B.1.1. Configure your Ubuntu repositories

First, you must check that your Ubuntu accepts restricted, universal, and multiversal repositories

B.1.2. Setup your sources.list

Setup your computer to accept software from packages.ros.org.

```
$sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

B.1.3. Set up your keys

. It is important to add the key because with it we can be sure that we are downloading the code from the right place and no body modified it.

```
$sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

B.1.4. Installation:

Before doing something, it is necessary to update all the programs used by ROS. We do it to avoid incompatibility problems. Type the following command in a shell and wait:

```
$sudo apt-get update
```

There are many different libraries and tools in ROS; the one installed and used in this worf is **desktop- full duplex**

```
$sudo apt-get install ros-indigo-desktop-full
```

B.1.5. Initialize rosdep

rosdep enables you to easily install system dependencies for source you want to compile and is required to run some core components in ROS.

```
$sudo rosdep init
$rosdep update
```

B.1.6. Environment setup

```
$echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$source ~/.bashrc
```

B.1.7. Getting rosinstall

```
$sudo apt-get install python-rosinstall
```

B.2. Steps to install the Kinect in ROS:

In the following we will take a look on the process of installing and running the Kinect. Firstly, we will install OpenNI and Kinect driver.

Installing dependencies:

```
$sudo apt-get install g++ python libusb-1.0-0-dev freeglut3-dev
$ sudo apt-get install doxygen graphviz mono-complete
$ sudo apt-get install openjdk-7-jdk
```

Intalling OpenNI:

```
$ git clone https://github.com/OpenNI/OpenNI.git
$ cd OpenNI
$ git checkout Unstable-1.5.4.0
$ cd Platform/Linux/CreateRedist
$ sudo chmod +x RedistMaker
$ ./RedistMaker
$ cd ../Redist/OpenNI-Bin-Dev-Linux-[xxx]
$ sudo ./install.sh
```

Installing Kinect driver

```
$ git clone git://github.com/ph4m/SensorKinect.git
$ cd SensorKinect/Platform/Linux/CreateRedist
$ sudo chmod +x RedistMaker
$ ./RedistMaker
$ cd ../Redist/Sensor-Bin-Linux-x64-v*
$ sudo ./install.sh
```

Now, we install ***openni_launch*** which includes launch files to open an OpenNI device and load all nodelets to convert raw depth/RGB/IR streams to depth images, disparity images, and (registered) point clouds.

```
$ sudo apt-get install ros-indigo-openni-camera ros-indigo-openni-launch
```

To run Kinect on ROS:

```
$ roslaunch openni_launch openni.launch
```

To visualise Kinect data

```
$ rosrun rviz rviz
```

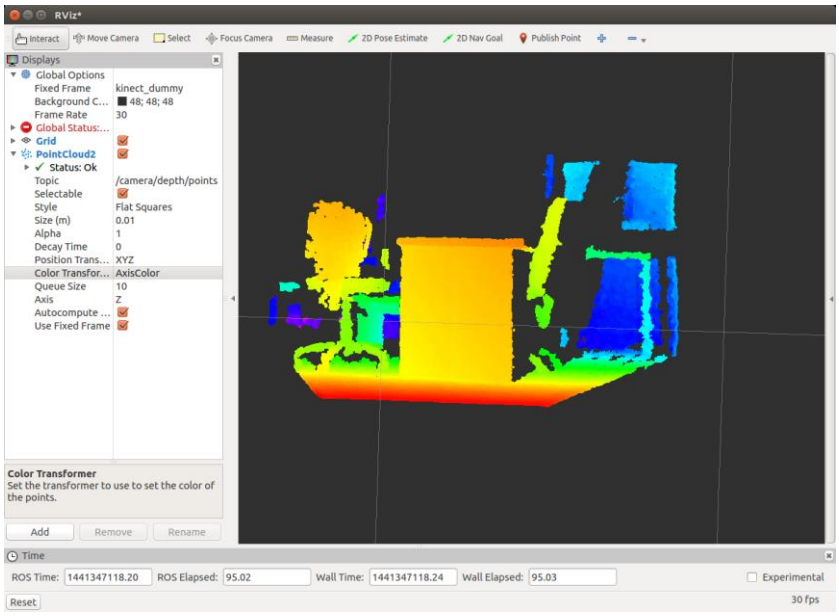


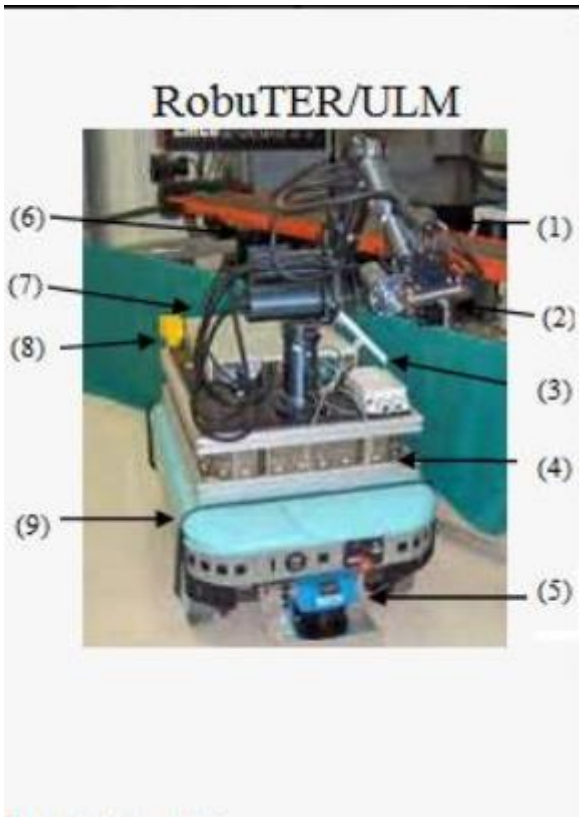
Fig B-1. The kinect test in the Rviz simulator

Appendix -C-

C.1. General description of RobuTER

RobuTER is a robotic mobile platform that is available in the Center of Development of Advanced Technologies (CDTA) of Algiers. It is a rectangular non-holonomic robotic mobile platform, developed by the French company Robosoft.

The robot base consists of a platform with two wheels and a load capacity of 15 kg. The wheels are 250 mm in diameter, and have a torque of 22 Nm nominal per wheel. They are driven by DC electric motors and enable it to reach a nominal speed of 2.6 m/s. The direction of RobuTER is given by the differential speed of the two wheels. The two wheels are placed at the front of the platform to provide stability.



(1)- Camera.
(2)- Efforts sensor.
(3)- Wireless Video Transmission.
(4)-Ultrasonic sensor.
(5)-LMS.
(6)-Manipulator Arm.
(7)-Embedded PC.
(8)- Joystick.
(10)- Mobil base.

Fig C-1. The architecture of the experimental robotic system

C-2. Software Architecture

RobuTER's embedded PC runs Linux RedHat 9.0 Operating System, which has the advantages of being free, open sourced, licensed under the GPL – General Public License, and being very well documented and featuring all necessary tools for development onboard of the RobuTER itself.

The development of applications for RobuTER is based on the Robosoft Development Toolchain. This development is based on the SynDEx CAD environment.

References

[1] "Robot path planning using interval analysis" Student: Hadi Jaber IASE 2012 Supervisor: Dr. Luc Jaulin ENSTA Bretagne/OSM

[2] "Mobile Robots Navigation" *Edited by Alejandra Barrera*

[3] "**Mobile Robots: towards New Application"s** *Edited by Aleksandar Lazinica, ISBN 978-3-86611-314-5, 600 pages, Publisher: I-Tech Education and Publishing, Chapters published December 01, 2006*

[4] *"Technical Report on Autonomous Mobile Robot Navigation" Ali Gürcan Özkil Marts 2009*

[5] *"An Investigation of Hybrid Maps for Mobile Robots" P. Buschka, , Örebro: Örebro universitetsbibliotek, 2005.*

[6] *"A survey on path planning techniques for autonomous mobile robots" by Leena.N , K.K.Saju, Cochin University of Science and Technology, India*

[7] *"Review of classical and heuristic-based navigation and path planning approaches" Adham Atyabi, David M.W. Powers*

[8] *"Classic and Heuristic Approaches in Robot Motion Planning – A Chronological Review" by Ellips Masehian, and Davoud Sedighizadeh*

[9] "Robot Motion Planning" J.C. Latombe. Kluwer Academic Publishers, Boston, MA, 1991. ISBN 0-7923-9206-X.

[10] *INTERNATIONAL JOURNAL OF SYSTEMS APPLICATIONS, ENGINEERING & DEVELOPMENT Issue 2, Volume 5, 2011, Buniyamin N., Wan Ngah W.A.J., Sariff N., Mohamad Z.*

[11] *"Off-Line and On-Line Trajectory" PlanningZ. Shiller (B) Department of Mechanical Engineering and Mechatronics, Ariel University, Ariel, Israel Springer International Publishing Switzerland 2015*

[12] *"Industrial robot navigation and obstacle avoidance employing fuzzy logic", P.G.*

Zavlangas, S.G. Tzafestas, J. Intell. Robot. Syst. 27 (1–2) (2000) 85–97.

[13] *"A trajectory planning of redundant manipulators based on bilevel optimization" R. Menasri, A. Nakib, B. Daachi, H. Oulhadj, P. Siarry, Appl. Math. Comput. 250 (2015) 934–947.*

[14] "On computing the global time optimal motions of robotic manipulators in the presence of obstacles " Shiller Z, Dubowsky S (1991) IEEE Trans Robot Autom 7(6):785–797

[15] *Research Journal of Recent Sciences Vol. 3(5), 110-115, May (2014) Mohsen AhmadiMousavi 1 , BehzadMoshiri 2 , Mohammad Dehghani 3 and HabibYajam Tehran, IRAN.*

[16] *"An Analysis for a Novel Path Planning Method" Kamkarian P, Hexmoor H (2015)* Adv Robot Autom 4: 130

[17] https://www.omicsgroup.org/journals/a-robotic-path-planner-contender-2168-9695-1000131.php?aid=60230&view=mobile

[18] http://library.isr.ist.utl.pt/docs/roswiki/Events(2f)CoTeSys(2d)ROS(2d)School(2f)SBPL_Lab.html

[19] State Lattice with Controllers: Augmenting Lattice-Based Path Planning with Controller-Based Motion Primitives.

[20] http://sbpl.net/node/53.

[21] https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree

[22] "A Performance Comparison of Rapidly-exploring Random Tree and Dijkstra's Algorithm for Holonomic Robot Path Planning"; S. M. LaValle, Planning *algorithms, New York: Cambridge University Press, 2006.*

[23] IJCSNS International Journal of Computer Science and Network Security , VOL.16 No.10,October2016

[24] http://www.ros.org/about-ros/

[25] "Browsing packages for indigo". ROS.org. ROS. Retrieved 21 February 2016.

[26] S. M. LaValle and J. J. Kuffner. Randomized kinodynamic planning. In International Conference on Robotics and Automation, pages 473–479, 1999.

[27] Optimal path planning and execution for mobile robots using genetic algorithm and adaptive fuzzy-logic control. A. Bakdi et al. / Robotics and Autonomous Systems 89 (2017) 95–109