Grid-based Map Building and Navigation Algorithms for Mobile Robots

Chui Ching Yee

M.Sc. in Electronic and Information Technology

Full-Time

2007-08

First supervisor: Dr. Bala Amavasai

Second supervisor: Dr. Lyuba Alboul

This thesis is submitted in partial fulfilment
of the requirements for the degree of
Masters of Science
in Electronic and Information Technology

Acknowledgements

This dissertation would not be accomplished as predicted without the cooperation of various parties. Words of thankfulness are thus forwarded to those whom I owed utmost respect and appreciation.

First of all and foremost, I would like to convey my deepest gratitude to Dr. Bala Amavasai, my project supervisor. His kind encouragements, constructive comments and patience have proven to be the pillar to achieve the goals of my project. The sacrifice and effort he put in to enable the project successfully work is truly sincere.

In spite of that, I was also deeply touched by the willingness of Mr. Joan Saez-Pons, MMVL researcher, to give invaluable insight related to the Player/Stage simulator. No forgetting my friends who had helped me from time to time. A special compliment goes to them for their generous contribution on ideas and inspirations to make this project a success.

Finally yet importantly, I wish to express heartiest appreciation to my parents for providing encouragement in fabricating my project. Thanks to those who had contributed to the success of this project.

Contents

Chapter 1 Introduction **1**
1.1 Background 1
1.2 Motivation 2
1.3 Block Diagram and Description 3
1.4 Deliverables/Objectives 3
1.5 Dissertation Foundation 4
1.5.1 Schedule 4
1.5.2 Equipments and cost required 5
1.5.3 Technical limitations 5
1.5.4 Potential hazards 5
1.6 Dissertation Guidelines / Structure 6
1.7 Summary 6

Chapter 2 Player/Stage Robot Simulator **7**
2.1 Robot Simulator 7
2.1.1 Introduction 7
2.1.2 Existing robot simulators 8
2.2 Player/Stage Robot Simulator 8
2.3 Methodology of using Player/Stage 10
2.3.1 Configuration and executable files 10
2.3.2 Client programming steps 11
2.3.3 Client program compilation linkages 12

Chapter 3 Literature Review and Relevant Theory **13**
3.1 Map Building 13
3.1.1 Introduction 13
3.1.2 Historical overview 13
3.1.3 Categories of map representation 14
3.1.4 Metric Map building approaches 16
3.1.5 Inherent problems of map building 17
3.1.6 Graphics library for Metric Map – PNGwriter 18

3.1.7 Sensing model used for map building 19

3.2 Autonomous Navigation System 22

3.2.1 Introduction 22

3.2.2 Obstacle avoidance system 22

3.2.3 Path exploration algorithm 24

3.2.4 Path finding algorithm 26

3.3 Control Algorithm for Map Building with Autonomous Navigation 27

Chapter 4 Methodology/ Algorithm 30

4.1 Specifications and Assumptions 30

4.2 Usage of Player/Stage Proxy Class 31

4.3 Map Building Algorithm 32

4.3.1 Map building architecture 32

4.3.1.1 Computation of position of laser reading 33

4.3.1.2 Interpretation of laser reading into knowledge 35

4.3.1.3 Determination of occupancy value 36

4.3.1.4 Map building using PNGwriter Graphic Library 36

4.3.2 Map building algorithm 37

4.4 Autonomous Navigation Algorithm for Map Exploration 38

4.4.1 Autonomous navigation architecture 38

4.4.1.1 Obstacle avoidance algorithm 38

4.4.1.2 Path exploration algorithm using modified DFS paradigm 41

4.4.1.3 A* path finding algorithm 46

4.4.2 Autonomous navigation algorithm 49

4.5 Control Architecture – Sense-Plan-Act (SPA) Approach 50

Chapter 5 Map Benchmarking Suite 52

5.1 Cross Correlation 52

5.2 Map Score 54

5.3 Map Score of Occupied Cells 54

Chapter 6 Empirical Evaluation 56

6.1 Obstacle Avoidance Algorithm 56

6.1.1 Ideal case 56

6.1.2 Case of asynchronous lower layer functions 57
6.1.3 Case of asynchronous higher layer functions 59
6.1.4 Discussion 60
6.2 A* Search Algorithm for Path Finding 60
6.2.1 Evaluation of A* search algorithm 61
6.2.2 Discussion 67
6.3 Modified DFS Algorithm for Path Exploration 68
6.3.1 Evaluation of Modified DFS algorithm 68
6.3.2 Discussion 71
6.4 Quality of Map Model Using Laser Scanner 71
6.4.1 Evaluation of map model quality 71
6.4.2 Discussion 73

Chapter 7 Conclusion and Further Work 74
7.1 Conclusion 74
7.2 Further Work 75
7.2.1 Sensor deployment 75
7.2.2 Self-localisation technique 76
7.2.3 Map building algorithm 76
7.2.4 Simultaneous Localization and Mapping (SLAM) 76
7.2.5 Path finding algorithm 77
7.2.6 Distributed map building using multirobot system 77

References 78

Appendices 93
Appendix A: *.world* configuration file 94
Appendix B: *.cfg* configuration file 95
Appendix C: *.cc* configuration file 96

List of Figures

Figure 1.1 Block diagram of robot system 3
Figure 2.1 Block diagram of control system without Player 9
Figure 2.2 Block diagram of control system using Player 9
Figure 2.3 Interaction Relationship between robot and client via Player/Stage 10
Figure 3.1 Breadth-first Search (BFS) 25
Figure 3.2 Depth-first Search (DFS) 25
Figure 3.3 Sense Plan Act (SPA) Architecture 28
Figure 3.4 Subsumption Architecture 28
Figure 4.1 Map building architecture 32
Figure 4.2 Computation of position of laser reading 33
Figure 4.3 Computation of θ_j with (a) zero θ_{robot} , (b) non-zero θ_{robot} 34
Figure 4.4 Information extraction from laser reading 35
Figure 4.5 Autonomous navigation architecture 38
Figure 4.6 Obstacle detection area 39
Figure 4.7 Determination of navigable points for path planning 41
Figure 4.8 Path generated using (a) DFS algorithm, (b) modified DFS algorithm 42
Figure 4.9 Modified DFS algorithm for complex and cross-link paths exploration 43
Figure 4.10 State machine of path planning algorithm using modified DFS 44
Figure 4.11 G scoring and arrow interpretation 47
Figure 4.12 Structure of robot control system using SPA paradigm 50
Figure 5.1 A typical corridor (a) ideal map, (b) generated map with curved obstacle (source from *[44]*) 54
Figure 5.2 A typical corridor with 'shadow' (source from *[44]*) 55
Figure 6.1 Ideal obstacle detection and avoidance 56
Figure 6.2 Series of obstacle avoidance algorithm empirical results for (a) ≈0, (b) ≈500, (c) ≈1000, (d) ≈1500, (e) ≈2000, (f) ≈2500, (g) ≈3000, (h) ≈3500, lines of lower layer functions 57
Figure 6.3 Series of obstacle avoidance algorithm empirical results for (a) 1, (b) 2, (c) 3, (d) 4, higher layer functions 59

Figure 6.4 A* search algorithm evaluation from node (10,10) to (10,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 61

Figure 6.5 A* search algorithm evaluation from node (10,10) to (150,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 62

Figure 6.6 A* search algorithm evaluation from node (10,10) to (80,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 62

Figure 6.7 A* search algorithm evaluation from node (80,10) to (80,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 62

Figure 6.8 A* search algorithm evaluation from node (10,10) to (10,153) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 63

Figure 6.9 A* search algorithm evaluation from node (10,10) to (10,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 63

Figure 6.10 A* search algorithm evaluation from node (80,70) to (10,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 63

Figure 6.11 A* search algorithm evaluation from node (10,10) to (70,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 64

Figure 6.12 A* algorithm evaluation from (10,10) to unreachable node (40,35) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 67

Figure 6.13 Modified DFS algorithm evaluation (a) original map (b) map model 68

Figure 6.14 Steps of path exploration and navigation using modified DFS Algorithm 69

Figure 6.15 Problem of Modified DFS 71

Figure 6.16 Grid-based map building algorithm (a) original map, (b) map model built (1st test), (c) map model built(2nd test) 71

Figure 6.17 Grid-based map building algorithm (a) original map, (b) map model built (1st test), (c) map model built(2nd test) 72

Figure 6.18 Grid-based map building algorithm (a) original map, (b) map model built (1st test), (c) map model built(2nd test) 72

List of Tables

Table 1.1 Work Plan 4

Table 3.1 Comparison of 3 types of map categories 15

Table 3.2 Advantages and drawbacks for a variety of sensors 20

Table 3.3 Comparison of three obstacle avoidance systems 24

Table 6.1 A* search algorithm evaluation for various terrain using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 64

Table 6.2 A* search algorithm evaluation for unreachable goal using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method 67

Table 6.3 Steps of path exploration and navigation using modified DFS algorithm 70

Table 6.4 Quality of map model evaluated using Cross Correlation, Map Score and Map Score with Occupied Cells Methods 72

Chapter 1
Introduction

1.1 Background

In an industrial fire, there are many hazardous circumstances that fire-fighters and rescuers will face, such as explosions, airborne chemical contamination, building collapse, senses impairment due to thick smoke and other uncertain situations. These not only endanger the fire-fighters and rescuers, but also impede the rescue of casualties due to time delay.

Time is critical, especially at search and rescue incidents. Generally, fire-fighters initially need to lay out guidelines and mark out a route to the fire or casualties, along with a safe route back to outside. Yet, this can lead to tragedies. In one case two fire-fighters died at Gillender Street, London in 1992 when they lost track of their exit route due to thick smoke when their air apply ran out. *[1]*

With the purpose of improving safety assessment whilst saving time , two miniature explorer robot systems named “ViewFinder” and “Guardians” are being developed by Sheffield Hallam University and several European partners including South Yorkshire Fire and Rescue Services to assist in search and rescue *[2]*. “ViewFinder” explorer robots will firstly enter the dangerous building before

fire-fighters and rescuers to map safe paths for the fire crew to access to fire and casualties respectively. Whereas, “Guardians” which will be sent in after, will help assess human safety by detecting fires and planning escape routes, which are then reported back to fire-fighters. Hence, path planning and risk assessment precedes fire-fighting and rescuing operation, and are then continually reviewed while the operations are on-going.

Both of these explorer robots will address key issues related to map building, autonomous robot navigation, multi-robot system, communication system, human-robot interfaces, and safety assessment.

1.2 Motivation

The existence of the “ViewFinder” explorer robot demonstrates a real life application of map building and shows how important map building is. In the event of fire, the ability to build maps under unknown environment is paramount for reliable localisation and real time navigation. Relying on original building ichnography may lead to unreliably measure due to dynamic environment.

Autonomous navigation systems without the “revisiting” problem, path planning algorithms and obstacle avoidance are all essential elements to glean all information within short enough time, so to fasten search and rescue job under safety mode. Without a map of environment, a robot can neither plan a path, nor search casualties effectively as it may retrace its step repeatedly.

As a result, an idea was proposed according to this “ViewFinder” concept – develop an autonomous navigation mobile robotic system with map building capability.

1.3 Block Diagram and Description

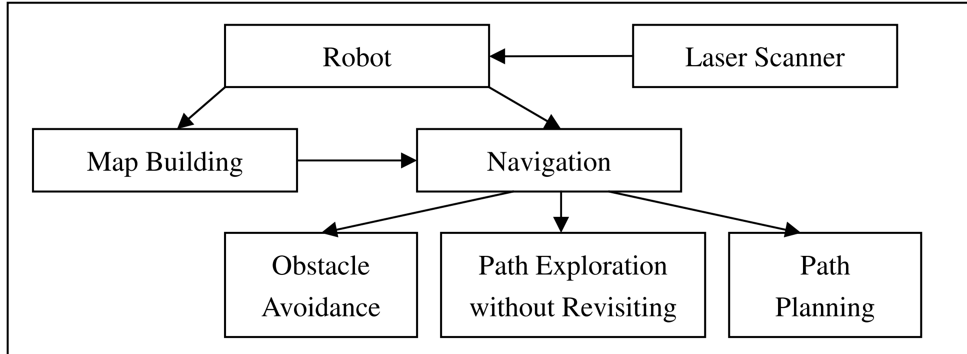


Figure 1.1 Block diagram of robot system

With reference to *Figure 1.1*, this dissertation will focus on map building and autonomous navigation for laser-guided robot in an unknown 2D and static environment for sake of simplicity. Obstacle avoidance, path exploration without revisiting and path planning will be considered as well in navigation.

However, this dissertation addresses the issue of a warehouse search in a limited space with no real smoke, hence safety assessment will be excluded. To fasten the progress of programming and ease troubleshooting, the Player/Stage robot simulator is used instead of creating real robot hardware.

1.4 Deliverables/Objectives

The deliverables from or objectives of this dissertation are:

1. the simulated robot possesses map building capability with obstacle detection and avoidance using laser sensor,
2. the robot is able to navigate or explore path without retracing the same place,
3. the design of path planning algorithm to find shortest path to destination desired,
4. use of Player/Stage to simulate robot client program, and
5. characterise the critical limitations of algorithms programmed.

1.5 Dissertation Foundation

During the implementation of this project, several inevitable constraints are firstly needed to be taken into consideration before the project commenced in order to make this project a success. These constraints are time taken, equipments and cost required, technical limitations, and potential hazards.

1.5.1 Schedule

Due to time constraint of 6 months and having 2nd semester study from October until January, a systematic work planning (as shown on *Table 1.1*) with well time management is a must.

Task	**June**	**July**	**Aug**	**Sept**	**Oct**	**Nov**	**Dec**	**Jan**
Research on Player/stage								
Familiar in using Linux								
Research on map building								
Research on sensor								
Programming and testing of map building								
Research on navigation algorithm								
Research on revisiting problem								
Programming and testing of navigation								
Research on path planning								
Programming and testing of path planning								
Research on multi-robot communication system								
Research on distributed map building								
Dissertation writing								

Table 1.1 Work Plan

1.5.2 Equipments and cost required

The work in this dissertation is carried out on a laptop running at 1.86GHz with 512 MB of RAM. The software simulator is Player/Stage which is performed via Kubuntu Linux OS. Thus, there is no expenditure for implementing this project.

1.5.3 Technical limitations

Due to the robot simulator and operating system being used is relatively new to author, long period of time was needed in doing research and familiarise with both Player/Stage and Kubuntu Linux. Since Player/Stage simulator is open source and still under development, some technical limitations are encountered as well:

- Less information related about Player/Stage
- Some essential and useful functions of Player/Stage are still in developing stage
- Difficulty in fine tuning the algorithm due to time restriction

1.5.4 Potential hazards

Awareness of safety issues will prevent unnecessary physical or mental harm. Working with laptop for long period will cause vision harm, waist strain, finger cramp and suffer a dull mentality as well. Some precautionary measures are recommended as following:

- Positioning monitor and chair to a comfortable pose
- An erect sitting posture is paramount
- Taking short break at regular intervals to prevent strain and cramp, whilst refresh mind

1.6 Dissertation Guidelines / Structure

Chapter 2 discusses Player/Stage simulator in details. Chapter 3 discusses the literature review or relevant theory of current work and picks novelty algorithm solution. Based on these well proven theories, algorithms development and methodology are attempted to be defined at Chapter 4. Chapter 5 carries out extensible suite of benchmarks which allow for the empirical evaluation of map building paradigm. For proof-of-concept, some simulation-based analyses has been performed, Chapter 6 investigates the results as well as critical analysis of the results obtained. Chapter 7 concludes the dissertation and recommends further work to improve the functionalities of the system.

1.7 Summary

This chapter highlights the source and importance of the dissertation relevant to one of European Funded Research Project – “ViewFinder”. It also identifies work plan, equipment setup and safety issues, followed by layout of the following chapters.

Chapter 2
Player/Stage Robot Simulator

2.1 Robot Simulator

2.1.1 Introduction

Robot simulation is an important element in robotics research for testing control schemes to be ported onto real robots. The robot simulator provides model of a real robot and its environment, which in particular provides the benefit of:

1. evaluating, predicting and monitoring the behavior of robot
2. reducing testing and development time
3. avoiding robot damage and operator injure due to control algorithm failure, which indirectly reducing robot repair cost and medical cost
4. fastening error finding in control algorithm implemented
5. offering data access that are hard to be measured on real mobile robot
6. allowing testing on various kind of mobile robot without need of significant adaptation of the implemented algorithm
7. easily switching between simulated robot and real one
8. providing high probability of getting success when implemented on real robot if the algorithm tested in simulation is proved to be successful

Yet, robot simulators are unable to calculate the real measurement due to system error and non-system error such as angle drift, sensor noise.

2.1.2 Existing robot simulators

In the past, several robot simulators have been developed, such as Player/Stage, Khepera and SIMROBOT. All of these robot simulators can simulate one or more robot in 2D environment.

The Stage Simulator provides various sensor models such as sonar, laser range finder, pan-tilt-zoom camera and odometry. It supports several programming languages including C, C++, Java, Tcl and Python. For Khepera Simulator, infrared sensor is the only type of implemented sensor. The control algorithm can be written in C/C++. MATLAB interface is also available. SIMROBOT is a robot simulator for MATLAB. It provides 2 types of implemented sensor, which are sonar and laser range finder.

Player/Stage is probably the most widely used. Most of the major intelligent robotics journals and conferences regularly publish papers featuring real and simulated robot experiments using Player, Stage and Gazebo *[6]*.

2.2 Player/Stage Robot Simulator

The Player Project *[6]* (formerly called “Player/Stage Project”, or “Player/Stage/Gazebo Project”) was founded in 2000 by Brian Gerkey, Richard Vaughan and Andrew Howard. It creates Free Software that enables research in robot and sensor systems. The Player software runs on POSIX-compatible operating system, including Linux, Solaris, BSD and Mac OSX (Darwin). A port to Microsoft Windows is still under planning stages.

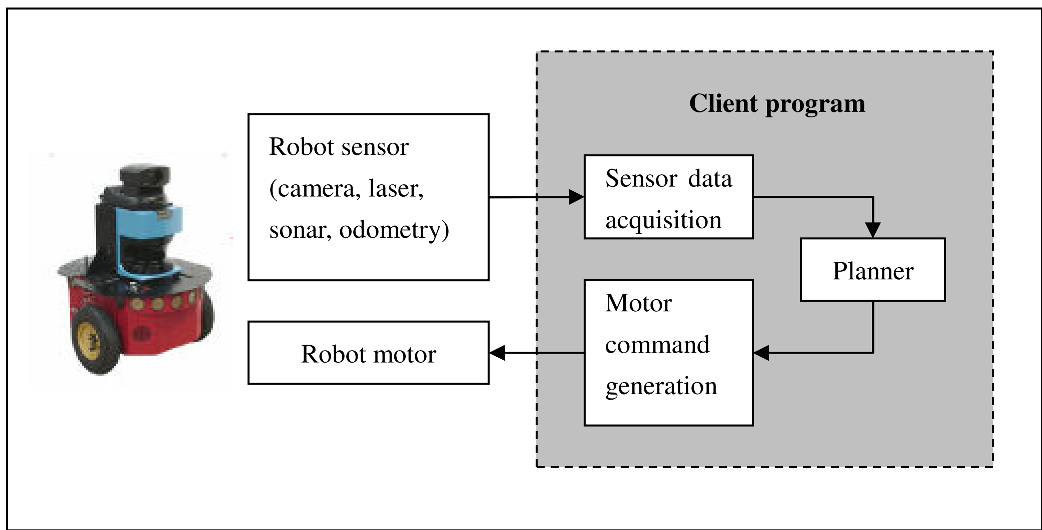


Figure 2.1 Block diagram of control system without Player

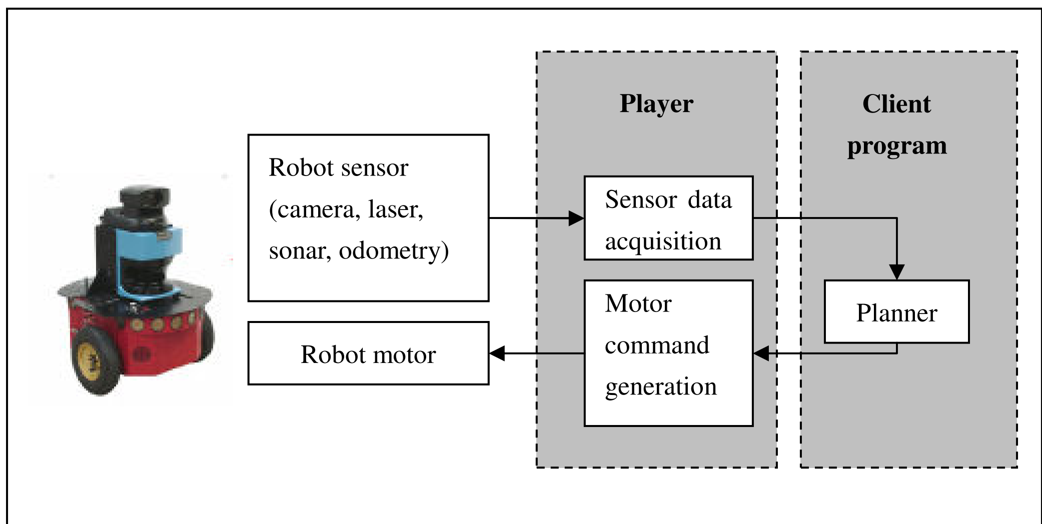


Figure 2.2 Block diagram of control system using Player

The comparison between *Figure 2.1* and *Figure 2.2* shows the use of Player ease the implementation of client program. Refer to *Figure 2.3*, Player *[7,9]*, which is Hardware Abstraction Layer (HAL) for robot device, provides flexible interface to various sensors and actuators hardware. Player supports multiple devices on the same interface, enabling distributed and collaborative sensing and control. It provides code repository and transport mechanism, allowing data exchange among drivers. Common used transport mechanism is client/server TCP socket-based transport.

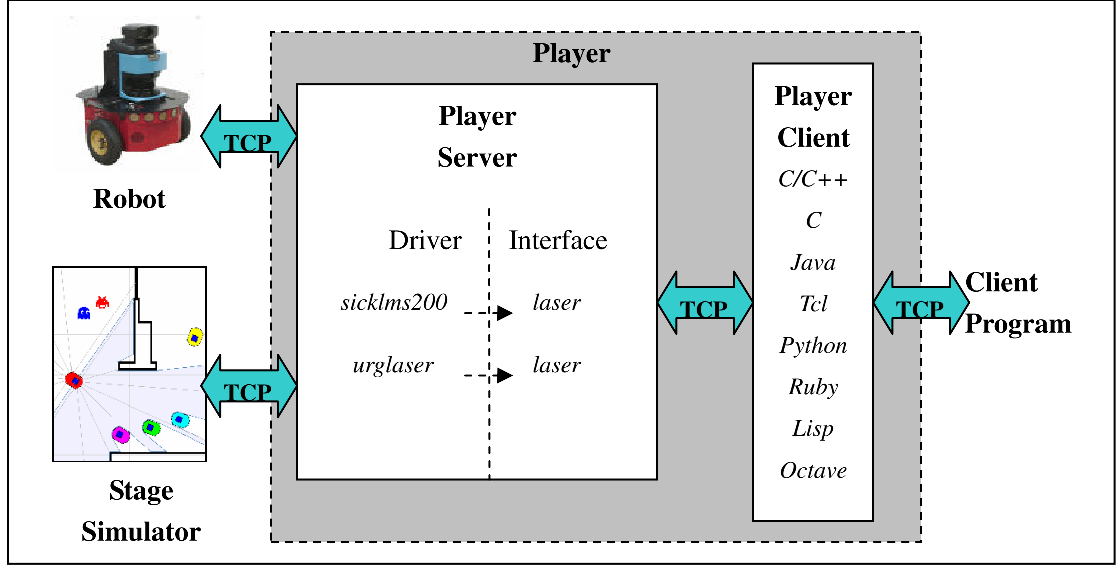


Figure 2.3 Interaction Relationship between robot and client via Player/Stage

There are three concepts of Player defines interface specification supported by more drivers:

1. interface – a specification of how to interact with certain class of sensor, actuator and algorithm, for example, laser
2. driver – is a software which making it appear to be same as any other entity in its class, for an instance, sicklms200, urglaser
3. device – is a driver bound to interface, for example, laser interface supported by both sicklms200 and urglaser

Stage *[8]*, which is 2D robot platform simulator, simulates a population of robots moving in and sensing a 2D bitmapped environment.

2.3 Methodology of using Player/Stage

2.3.1 Configuration and executable files

A common way to use Player is to run the Player server on robot, then to access robot's device with client program *[10]*. Using Player with Stage requires 2

configuration file, which are *.world* file (stage configuration file) and *.cfg* file (player configuration file). *.world* file defines simulated world with virtual device inside it, whereas *.cfg* file map the virtual device to Player device and instantiate device to access and control robot. (refer to *Appendix A and Appendix B*)

Another two important files used are *.png* file and *.cc* file. The *.png* is a graphical of the environment or world. *.cc* file (refer to *Appendix C*) is C++ based executable file to execute client program into simulated robot.

2.3.2 Client programming steps

```
int main (int argc, char **argv)
{
        // establish connection
        PlayerClient robot(gHostname, gPort);
        // instantiate device
        LaserProxy lp(&robot,0);
        PositionProxy pp(&robot,0);

        While(1)
        {
                // data acquisition
                robot.Read();
                // process data
                speed = …..; // calculate new speed and turnrate based on laser data
                turnrate = …;
                // send motor command
                pp.SetSpeed(speed, turnrate);
        }
}
```

Pseudo Code 2.1 Client programming steps

In writing a client program *[17]*, firstly, the user needs to establish a connection to Player server to instantiate simulated robot using PlayerClient proxy. Then, appropriate device proxies (such as LaserProxy *[15]*, PositionProxy *[16]*) are created

to instantiate devices of the instantiated robot. By utilizing the member functions of these proxies, user can acquire data, process the data, and lastly send appropriate command. (refer to *Pseudo code 2.1*)

2.3.3 Client program compilation linkages

For Kubuntu Linux operating system, the compilation linkage for client program is $ g++ `pkg-config --cflags playerc++` -o *filename filename*.cc `pkg-config --libs playerc++`. Firstly, run the *.cfg* file, then simply type command $*./filename* to execute the corresponding client program.

Chapter 3
Literature Review and Relevant Theory

3.1 Map Building

3.1.1 Introduction

Map building is the process of generating a model of the surrounding area for autonomous robot motion in unknown environment. An accurate model of the robot's surrounding environment facilitates fast-timing and reliable completion of a variety of complex tasks, such as path planning, path exploration.

3.1.2 Historical overview

In the 1980s and early 1990s, the field of robot mapping *[21, 72]* was widely divided into metric and topological approaches. An early representative of metric mapping algorithm, known as "occupancy grid mapping algorithm" developed by Elfes and Moravec *[47, 95]*, which represents metric maps by fine-grained grids that model the occupied and free space of the environment. This approach has enjoyed enormous popularity, such as *[55, 57, 96, 97, 99, 100, 119]*. Examples of topological approaches include *[51, 102, 103, 106, 107, 119]*.

Since the 1990s, the field of robot mapping has been dominated by probabilistic techniques, which basically use Bayes theorem to model a maximum likelihood map based on data. There are series families of probabilistic approaches, such as Kalman Filter (use Gaussians to estimate the robots pose) *[55, 104, 108]*, Expectation Maximization *[111, 118]*, Object Maps [81, 110].

Incremental techniques are designed to work in real time easier than probabilistic methods. Incremental methods include occupancy grids *[47, 48, 49]*, and DOGMA. The basic principle of occupancy grids is to calculate the binary occupancy of a location (x,y) and incrementally update each grid cell. DOGMA (Dynamic Occupancy Grid Mapping Algorithm) is an extension of occupancy grid approach that operates in dynamic environment.

Simultaneous localization and mapping (SLAM) *[84 ,114]* is a breakthrough in modern robot mapping. The robots start in an unknown pose and incrementally mapping an unknown environment while simultaneously using this map model to update its location. Significant progress has been made towards the solution of the SLAM problem *[86, 104, 108]*, such as SLAM extension from 2D to 3D *[114, 116]*, online SLAM for dynamic environment *[85]*, FastSLAM *[50]*, topological SLAM *[51]*, multiple-hypothesis approach for underwater robot *[87],* multirobot SLAM *[53]*.

3.1.3 Categories of map representation

Map model representation can be categories into Metric (grid based) map, Topological map and Metric Topological map. Metric map, which is cell-based structure, records the properties of each cell. The cell is generally represented as a square of grid. Topological map is graph-based structure that only records the existence of recognisable places and the paths between them without distance information. Metric Topological maps are similar to topological maps, but provide

additional distance information about the path between locations. Comparison of these 3 types of maps is shown as *Table 3.1*.

	Metric (grid based) Map	**Topological Map**	**Metric Topological Map**
Characteristics	- cell-based structure - store information of obstacle and spatial relationship	- graph-based structure - no geometry relation between path	- graph-based structure - possesses geometry relation between path
Examples	map divided into evenly size square with obstacle shaded black (occupied) and free space blank	nodes represent places, edge is navigable path between places	the distance between place A and place B is 10 metre
Pros	- easy to construct - useful in map matching - can dissimilar identical places or objects - enable estimation of robot's and obstacle's pose	- require less storage - less computation time - faster path planning using Dijkstra Algorithm *[34]*, but path chose may not the shortest	- require less storage - less computation time - path planning algorithms more optimal compare to Topological Map
Cons	- require huge storage - large computation time - path planning may not efficient, but the path chose may shorter than that of Topological Map	- harder to construct - not valid for map matching - perceptual aliasing in recognizing identical place	- harder to construct - not valid for map matching - perceptual aliasing in recognizing identical place

	- sensitive to noise	- cannot estimate the pose of robot and obstacle	- cannot estimate the position of robot and obstacle

Table 3.1 Comparison of 3 types of map categories

This dissertation is primarily concerned with the acquisition of the Metric Map, due to its usefulness, popularity and ease of construction. However, the size of grid for Metric Maps depends on the clock pulse and accuracy of sensor used. Higher resolution comes at a higher computational time, but it helps to solve various hard problems.

3.1.4 Metric Map building approaches

Many metric mapping techniques use occupany grid approach coupled with probabilistic approach, in order to handle uncertainty during estimating map and robot pose. Several paradigms currently used are Probabilistic Occupancy Grid theory put forward by Moravec and Elfes *[47, 95]*, Bayesian based Occupancy Grid methods by Matthies and Elfes *[48]*, and MURIEL method (Multiple Representation, Independent Evidence Log) by Kurt Konolige *[49]*.

Probabilistic representation is a method for measuring a probability value using numbers in the range [0, 1] to record the occupancy status (unknown, empty, or occupied), with 0 representing an absolutely empty area, 1 indicating an absolutely occupied cell, and 0.5 representing unknown area. The advantage of this is that when the number 0.5 is put into Bayes equation *[48]*, meaning that nothing new about the environment, no change is made to the cell value.

Since this dissertation only focuses on initial stage of map building, thus the probabilistic mapping techniques will not be discussed. However, in order to ease improvement of grid value accuracy in future, three values of 0, 0.5 and 1 will be

used to indicate grid occupancy status: emptied, unknown and occupied respectively.

3.1.5 Inherent problems of map building

In mapping an environment, some difficulties are encountered:

1. Inaccurate estimation of robot's and object's poses

   The goal of robot mapping is for an autonomous robot to be able to render map and localize itself in it. However, robot may suffer errors in odometry such as angular drift and wheel slippage, eventually render a large error-prone map. This need compensated with self-localization techniques, such as dead reckoning *[19, 43]*, Monte-Carlo Localization *[46]*, landmark based matching algorithm.

2. Difficulty in translating sensor reading into knowledge about the environment

   Without vision system, it is difficult to interpret sensors reading as objects in real world, such as stair, wall, human, table.

3. Dimensionality of environment

   For purpose of map updating, all information about the environment needed to be stored. Thus, more memory capacity and computation time is required for larger surface area of two-dimensional (2D) map. Since three-dimensional (3D) map is much more tangled, it goes without saying that huge amounts of data and complex algorithm are needed.

4. Dynamic environment

   Under dynamic environment, such as opening door, movable human and objects, the map that a robot built may no longer be valid after a period of time. For example, a robot facing a closed door that previously was open. This can be explained by two hypotheses, either the door status changed, or that the robot is not where it believes to be.

5. Path exploration during mapping

   The task of generating robot motion in the pursuit of building a map is commonly referred to as robotic exploration. Exploring robots in the unknown environment have to cope with partial and incomplete models, and “revisiting” problem. Hence, any viable exploration strategy has to be able to accommodate contingencies that might arise during map acquisition.

3.1.6 Graphics library for Metric Map – PNGwriter

During Metric Map acquisition, a graphic library is used for sake of image creation. It generally deals with a rectangular table (M*N pixels) with each pixel has its own colour. Once the information or value for the cells (small evenly size square in grid) are stored or updated, graphics library will translate these values into specific colour to indicate the property of the cells (occupied or emptied). Normally, black colour is used to represent obstacle (cell occupied), and white colour is used to represent free space (cell emptied).

Existing graphics libraries are GD Graphics Library *[22]*, JavaScript Vector Graphics Library *[23]*, PNGwriter Graphic Library *[24]*, PGPLOT Graphics Subroutine Library *[25]* and others.

In this dissertation, PNGwriter Graphic Library *[24]* is used for Metric Map creation in map building due to its simplicity and portability. PNGwriter Graphic Library is an easy-to-use graphics library that plots a high quality PNG image pixel by pixel from C++ program. It runs under Linux, Unix, Mac OS X and Windows.

Due to its characteristic, during map building using Player/Stage robot simulator, PNGwriter provides the benefits of:

1. enabling fast image creation, as it can directly create the Metric Map model from C++ source code used by Player,

2. creating reusable image file, as PNGwriter can create, read and update the output image, and
3. ease in map matching between map model created by PNGwriter and original map used by Stage simulator, as both of these maps are in PNG format

3.1.7 Sensing model used for map building

In modeling map in an unknown environment, sensor plays an essential role in retrieving data on properties of the environment, enabling robot to perceive the world. Sensors commonly brought to bear this task include sonar *[44, 48, 95, 98]*, laser *[58, 116]*, infrared *[98]*, stereo vision *[48, 58, 76, 98, 110]*, wheel encoder and touch sensors, each has its pros and cons as shown in *Table 3.2*.

For sonar sensors, the distance relationship between robot and surrounding object is measured via acoustic pulse emitted. Using total time elapsed (t) for emitting acoustic pulse and receiving echo, the range (D) can be calculated using formula $D = (v*t)/2$, where v is the speed of sound. Speed of sound (v) is proportional to temperature (T), that is $v = 20*\sqrt{T}$.

The operating principle of lasers is same as sonar, it emits a short pulse of light (laser). The time elapsed (t) between emission and detection is used to determine distance (D) using the speed of light (c), $D = (c*t)/2$. It is able to emit laser beams with around 0.5° spread, which are much narrower than sonar beams of $25^{\circ} - 30^{\circ}$.

Stereo vision is an optimal sensing method that uses two cameras placed in different positions to capture images, detect object, analyse profile of object and determine the distance to the object.

Unlike the three sensors described formerly, wheel encoder (also called odometry) does not provide information about environment, it only determines

distance and angle raveled by the robot itself, thus able to estimate the position and orientation of robot. The distance (D) is measured by multiplying the number of revolutions of the wheels (n) by perimeter per revolution (p), i.e. D = n*p.

A touch sensor is a simple on-off switch, it is activated when the robot hits an obstacle. It is usually used in object avoidance system as last line of defense.

Robot sensor type	Advantages	Drawbacks
Sonar sensor	- relatively low cost - fast computational time due to less or no process of determine obstacle position - able volumetric sensing	- inaccurate and noisy, as roughness surface causes scattering reflections or angle of reflection is too large that acoustic pulse reflected is away from receiver - specular reflections give rise to erroneous readings - arrays of sonar sensors can experience crosstalk, which one sensor detects the reflected beam of another sensor - unable to determine the exact position of objects
Laser sensor	- less chance of specular reflections due to its shorter wavelength - high accuracy - able to determine exact	- more expensive - only able to detect objects in plane - require higher processing power and memory

	position of object relative to robot - gives detailed description of the field of view	
Stereo vision	- able to detect objects that sonar and laser may miss - able to identify and dissimilar objects - suit for 3D environment	- higher processing as more information needed for each object
Wheel encoders	- useful in navigating robot - able to estimate the position and orientation of robot	- measurement of distance and angle travelled is inaccurate due to wheel slippage or angular drift, which leads to error in estimating robot position and orientation
Touch sensor	- no algorithm needed - high reliability	- not suitable for map building, as it only gives signal when hit object

Table 3.2 Advantages and drawbacks for a variety of sensors

Sonar, laser and stereo vision are non-contact sensing model, thus they are less reliable compare to touch sensor due to environment effects. Environmental factors such as humidity and temperature affect the output of the sonar systems. Natural light interference is a problem in laser scanners because it can interfere with the readings. Stereo vision is influenced by brightness and degree of visibility.

The Stage Simulator provides various sensor models such as sonar, laser range finder, pan-tilt-zoom camera and odometry. Regardless of environment effects, laser

sensor is an optimal sensing for 2D map building due to its accuracy in providing information about surrounding environment and its ability of determining exact position of objects relative to robot. Odometry device is used for navigation and localisation, it helps in estimating object pose in map model when combining with laser reading data.

3.2 Autonomous Navigation System

3.2.1 Introduction

Motion is ubiquitous in both the real world and synthetic environments. In the field of unmanned robotic systems, autonomous navigation system is obligatory for industrial robot. Current industrial robot lack flexibility and autonomy, as these robots perform pre-programmed sequences of operation in highly constraint environment, and are not able to operate in new environment or to face unexpected situation. This is inevitable problem, as the autonomous navigation system is typically designed according to demands and environment limitations.

In this dissertation, the navigation system is designed for sake of map acquisition in 2D indoor unknown environment, thus effective path exploration, shortest path finding and obstacle avoidance must be well-designed. Path exploration and path finding are challenging navigating problem, which is often solved sub-optimally via simple heuristics.

3.2.2 Obstacle avoidance system

One of the challenges in designing intelligent autonomous mobile robots is reliable obstacle avoidance. Obstacle avoidance *[56, 58, 76, 96]* plays an important role in prevent both robot and object hit from damage. Obstacle avoidance can be

divided into two parts, obstacle detection and avoidance control.

There are 2 sensors that widely used in detecting and avoiding obstacle, which are sonar *[44]* and laser *[58, 116]* sensors. Numerous methods for obstacle avoidance have been suggested, for example, stationary sonar sensors, a rotating sonar sensor and laser scanner system. As only 2D environment is focused, vision-based system *[76]* using camera is not necessary.

For stationary sonar sensors system with decentralized locating of several sonar sensors, only the region in which the obstacle lies can be determined. Conversely, neither the exact obstacle pose nor obstacle size can be determined.

The rotating sonar system compensates the weakness of stationary sonar sensors system. It gives more accurate position for the obstacle. However, it is relatively costly due to additional drive mechanism and requires complex programming to control the drive mechanism. The motor has to rotate slowly so that the transducer has enough time for the acoustic pulses transmission. Also, the vibration of the drive motor causes data noise. Another disadvantage is the error in width detection because of low angular accuracy. Both stationary and rotating sonar systems suffer to problem of reflection and scattering of sound waves.

Laser scanners are found to be more reliable, it provides high position and angular accuracy due to its high resolution of 0.25 degrees and tightly focused beam. It is also able to detect multiple obstacles.

Table 3.3 shows the comparison of these three systems. Obviously, laser scanner system is the best choice as obstacle avoidance system in this dissertation, due to its benefits over sonar system.

Criterion	Stationary sonar	Rotating sonar	Laser scanner
Position of obstacle	Region in which it lies can be determined	Approximately estimation	Exact position can be determined
Orientation of obstacle	No	Approximately estimation	Yes
Distance accuracy	Low	Low	High
Angular accuracy	-	Low	High
Obstacle size	Cannot be determined	Cannot be determined	Can be determined
Multiple obstacle detection	No	No	Yes
Data noise	High	High	Low
Environment effect	High	High	Low

Table 3.3 Comparison of three obstacle avoidance systems

3.2.3 Path exploration algorithm

Two major difficulties in path exploration is the need to cope with the large amount of uncertainty environments and revisiting problem. Robots, which perform pre-programmed sequences of operation, are lack of flexibility and are not able to operate in new environment or to face unexpected situation. This may leads to inefficient operation and time delay, especially in uncertainty or dynamic environment.

In addition, during path exploration, the mobile robot will never know itself that it was retracing its steps and revisit the same terrain. As a consequence, it will not be able know if it have build the whole indoor map completely. Revisiting will exhaust more time and result to ineffective map building.

Many algorithms done *[92, 93]* are on-line exploration and navigation.

Hoffmann, Icking, Klein and Kriegel *[94]* have described a competitive on-line strategy for polygon exploration. Important work done by Fox, Ko, Konolige and Stewart *[69]* developed a hierarchical Bayesian approach to address with revisiting problem.

A systematic search strategy is a significant solution to encounter the exploration problems. Due to unknown state spaces, uninformed search algorithm *[27, 28, 29, 30, 42]* is suit for solving revisiting problem. There are two uninformed search strategies that widely used in 2D search scheme:

1. Breadth-first Search

   Breadth-first Search (BFS) is a graph search algorithm that begins at a given vertex, which is at level 0 and explores all vertices at level 1, then explores all vertices at level 2, and so on. Refer to *Figure 3.1*, the path generated is A – B – C – B – D – E – F.

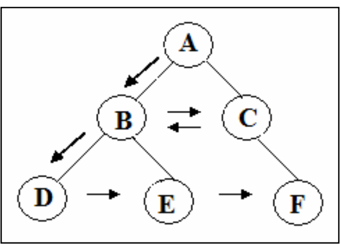


Figure 3.1
Breadth-first Search

2. Depth-first Search

   Depth-first Search (DFS) is another way of traversing, which starts at a root and explores as far as possible along each branch before backtracking. It goes deeper and deeper until hits a node that has no children, then backtracking to the most recent node it has not explored. It is easy to program as a recursive routine. Refer to Figure 2.4., the path generated is A – B – D – B – E – B – A – C – F.

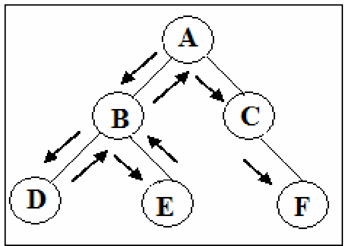


Figure 3.2 Depth-first Search (DFS)

Both BFS and DFS use marks to keep track of the vertices that have already been visited, and not visit them again. And both BFS and DFS are used to search until goal is found. However, target or object finding is not concerned in path or map

exploration. Yet, these search approaches still can be applied theoretically in exploration and navigation system to cope with revisiting problem.

For simplicity, DFS is a more suitable, optimal and effective algorithm to be used as path exploration algorithm for map building. In consideration of effectiveness, DFS will be modified, the detail will be discussed in Chapter 4.

3.2.4 Path finding algorithm

Once the navigable paths are found through path exploration algorithm, the robot can move to the navigable paths. To make the robot to be useful and intelligent, path finding algorithm is required to plan a safety and shortest route from source (initial state) to destination (goal state) without possible obstacle collision.

Informed/Heuristic Search Algorithms *[27, 31, 60]* is widely used in path finding. It uses heuristic function *[62]* that estimates "distance" (cost) from the current node/state to goal to guide search.

The A* Search *[67, 82, 83]* is common and widely used Informed Search Algorithm for path planning. A* Search uses the known cost combined with an estimate heuristic to choose a node to expand. It incrementally searches a sequence of state transitions that leads a robot from its initial point to desired goal. A* is well for static and deterministic environment.

Hierarchical Pathfinding A* (HPA*) *[40]* is a better version of A* using "divide and conquer" technique. HPA* possesses Pre-processing Phase prior to Pathfinding Phase, that divides the large grid into smaller clusters and builds a subgrid connectivity graph.

To find exact the shortest path, Dijkstra's algorithm *[34]* is the best choice. It

searches by expanding out equally in every direction, determines the distances between one point to all other points, and eventually chooses the exact shortest path. Dijkstra's algorithm usually ends up exploring a much larger area before the goal is found. This generally makes it slower than A*.

The D* algorithm (Dynamic A*) *[63, 64]* is most widely used for path re-planning at dynamic environment, due to its efficient use of heuristic and incremental updates. It repairs or re-plans the path once new information is discovered.

Stentz has developed Focussed D* *[65]* that repeatedly determines a shortest path from the current robot coordinates to the goal coordinates while the robot moves along the path. It is able to replan faster than planning from scratch.

Lifelong Planning A* (LPA*) *[67]* generalizes both A* and a version of DynamicSWSF-FP. It is an incremental heuristic search method that repeatedly determines shortest paths between two given vertices as the edge costs of a graph change. This algorithm reconstructs only the areas affected by the changes to the environment's state.

A* algorithm is selected to be implemented in this project, since environment concerned is static and deterministic, and A* provides faster computation and uses smaller data storage than that of Dijkstra's algorithm.

3.3 Control Algorithm for Map Building with Autonomous Navigation

In autonomous robotics system, control algorithm is a vital architecture that enables distributed and collaborative sensing and control. Three different approaches

have been proposed:

1. Traditional approach – Sense Plan Act (SPA) Architecture

   In the traditional approach *[32]*, perception, planning and execution follow each other in exact order (*Figure 3.3*). This theoretically enables robots to deal with complex problems, yet its top-down behaviour makes it slow deliberative and inflexible, and not suited for fast changing environments.

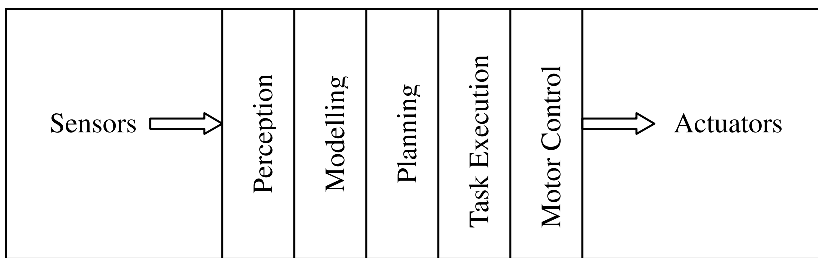


Figure 3.3 Sense Plan Act (SPA) Architecture

2. Subsumption / Reactive architecture

   Rodney Brooks *[74]* criticised the weakness of traditional approach, to provide deliberative architecture, Brooks created Subsumption Architecture (refer to *Figure 3.4*), which advocates layering of behaviors architecture *[35, 52]*. All layers running in parallel with little interaction between them.

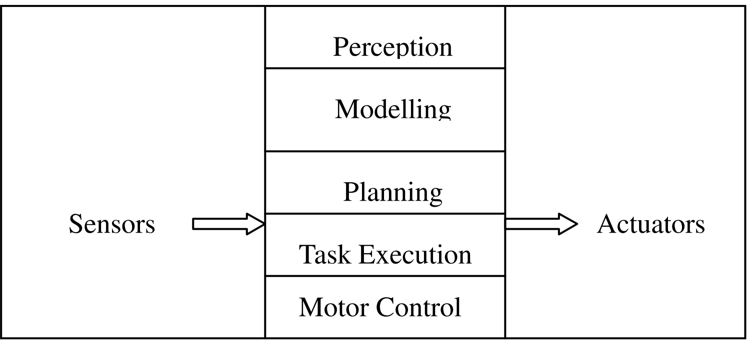


Figure 3.4 Subsumption Architecture

However, this approach is inflexibility at run time with relatively slow

response, significant processing power is required to maintain accurate up-to-date all the time. Thus, it is well with low-level behaviours, such as obstacle avoidance and wall following, but not suit for higher-level functions such as learning or planning. This approach has been worked by some researchers *[75, 77,117]*.

3. Hybrid architecture

   To compensate the lack of higher functions, Hybrid architecture was proposed. It applies behaviour-based, reactive system for low-level control, and a central planning device for higher-level behaviours such as planning and mapping. For an instance, SSS *[36]* is a hybrid 3-layer architecture applied by Jonathan H. Connell. An interesting development of Hybrid architecture is the use of a deliberative layer to perform higher-level functions like mapping and navigation. This layer can be implemented through neural networks or genetic algorithms, among others.

To ease construction, Sense Plan Act Architecture is concerned in this project. It enables easy implementation of mapping, path planning and obstacle avoidance algorithms in static environment.

Chapter 4
Methodology/ Algorithm

4.1 Specifications and Assumptions

The specifications of the robot simulation presented in this dissertation is given below:

1. Player/Stage server environment will be configured to simulate the Pioneer robot inside a series of 2D industrial warehouse terrains, each with square area of 16x16m^2.
2. Only odometry device and laser scanner device are required.
3. The laser scanner device is able to scan a planar field of 180° with 0.5^o resolution, which consists of 361 samples of reading data (up to 4 meters).
4. The origin location of robot in environment is known.
5. The map model built is a grid of 160x160 cells (represents 16x16 m^2), with internal of 0.1m for each x-axis and y-axis.

Due to the nature of real world imperfections, some assumptions has been made in this simulation, as given below:

1. The operation of the robot is ideal, i.e. there is neither systematic nor non-systematic error which leads to measurement error, hence, localisation techniques are not addressed

2. The environment is under normal temperature with good visibility and good condition that do not impair the accuracy of the laser reading
3. High precision laser and odometry devices do not deal with inaccuracies and errors in reading

Assumptions made about the environment:

1. Static (unchanging)
2. Observable (can sense its initial and current state)
3. Discrete (world carved up into towns)
4. Deterministic (no unexpected events)

Other assumptions made during execution:

1. The rate of environmental change is zero and only a static environment is addressed
2. The sequences are not overly complete so they do not perform computations that take a long time which cause further time delay
3. Whole processes complete within one clock cycle of the robot execution.

4.2 Usage of Player/Stage Proxy Class

In this project, the simulated Pioneer robot is implemented with a p2os *[12]* controller to provides position2d *[14]* and laser *[13]* interfaces. Player/Stage provides a C++ client library for a variety of devices of Player server, each associated with appropriate and ready-to-use proxy classes.

In this robot simulation, for the laser scanner device, sicklms200 driver *[11]* with laser interface *[13]* is used, which is associated with LaserProxy *[15]* class for data acquisition. The Position2dProxy *[16]* class is associated with position2d interface *[14]* to return odometry data, and accepts velocity commands.

4.3 Map Building Algorithm

4.3.1 Map building architecture

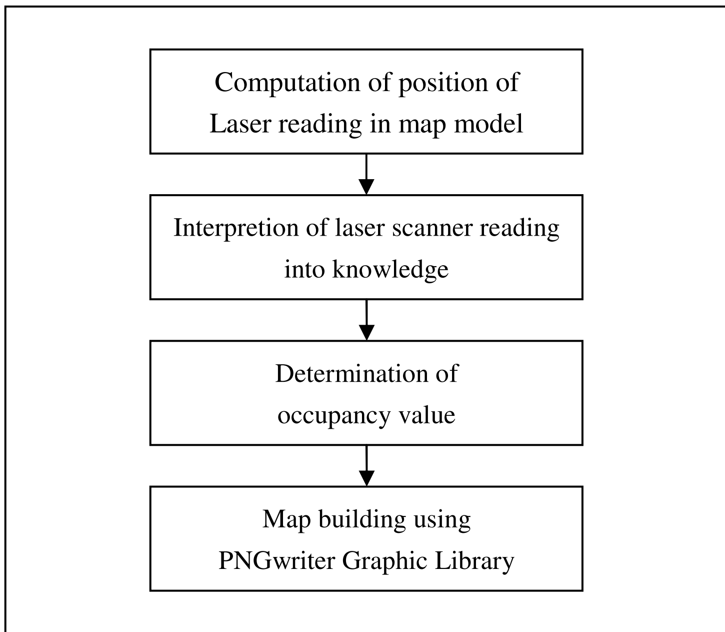


Figure 4.1 Map building architecture

The map building architecture is described herewith. *Figure 4.1* decribes map building architecture possesses 4 major algorithms:

1. Computation of the position of laser reading in map model

   First and foremost, the laser scanner reading data, with total of 361 samples, must be translated into Cartesian [x,y] form. This will provide an outline of the perimeter of the laser scanning area.

2. Interpretion of laser scanner reading into knowledge

   All the 361 positions (cells in map model) must then be evaluated and interpreted into useful knowledge. The knowledge is information that define the property of the corresponding cell, whether the cell is occupied (indicates obstacle in real world) or emptied (indicates free space in real world).

3. Determination of occupancy value using Probability Occupancy Grid Theory

   Based on Probability Occupancy Grid Theory, cells are given certain occupancy value of 0, 0.5 and 1, representing the property of the cell.

4. Map building using PNGwriter Graphic Library

   Finally, with the occupancy values computed, the simulated environment model can be created using the PNGwriter Graphic Library.

4.3.1.1 Computation of position of laser reading

As the origin location of the robot is known, intuitively, with odometry data, the location of robot can be computed and defined in [x,y,θ] form, representing the position and orientation of robot in map model. Then, by combining the laser reading, the position of the laser reading at corresponding time can be computed.

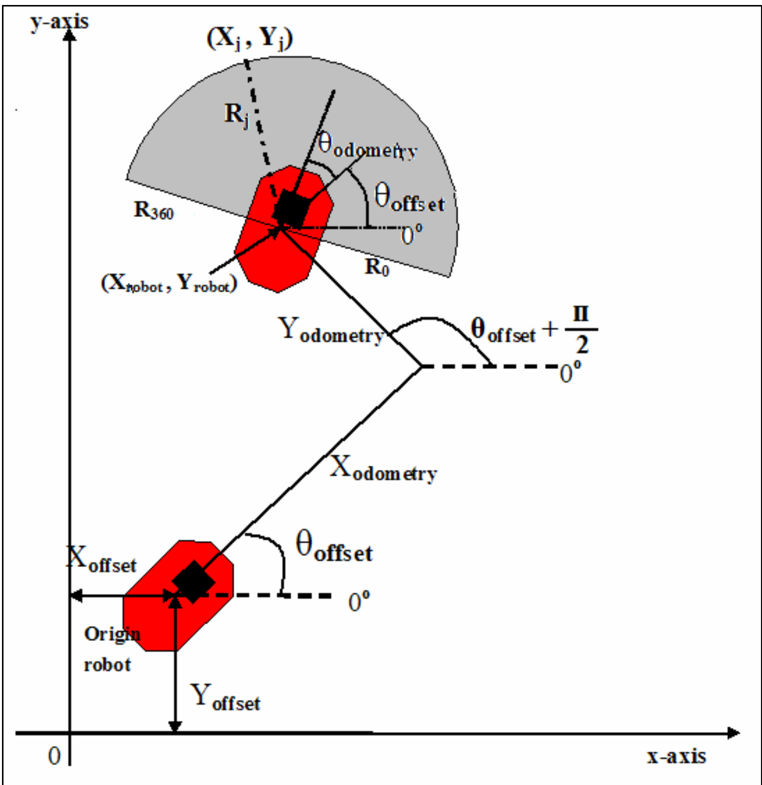


Figure 4.2 Computation of position of laser reading

Refer to *Figure 4.2*,

Given that origin location of robot in map model = (X_{offset}, Y_{offset}, θ_{offset})

After a duration of time, the new location of robot = (X_{robot}, Y_{robot}, θ_{robot})

based on odometry data ($X_{odometry}$, $Y_{odometry}$, $\theta_{odometry}$),

$X_{robot} = X_{offset} + X_{odometry}*\cos(\theta_{offset}) + Y_{odometry}*\cos(\pi/2 + \theta_{offset})$ ---*Equ 4.1*

$Y_{robot} = Y_{offset} + X_{odometry}*\sin(\theta_{offset}) + Y_{odometry}*\sin(\pi/2 + \theta_{offset})$ ----*Equ 4.2*

$\theta_{robot} = \theta_{offset} + \theta_{odometry}$ --- *Equ 4.3*

To calculate the position of laser reading R_j, first, the sample number of 0-360 must be convert to [-90°, 90°],

notice from *Figure 4.3(a)*, θ_0 (sample no. 1) = - 90°

θ_{180} (sample no. 181) = 0°

θ_{360} (sample no. 361) = 90°

thus, it can be deduced that θ_j (sample no. j+1) = j/2 - 90°

then, from *Figure 4.3(b)*,

θ_j (sample no. j+1) = (j/2 - 90°)*π/180 ° + θ_{robot} (in radian) ----*Equ 4.4*

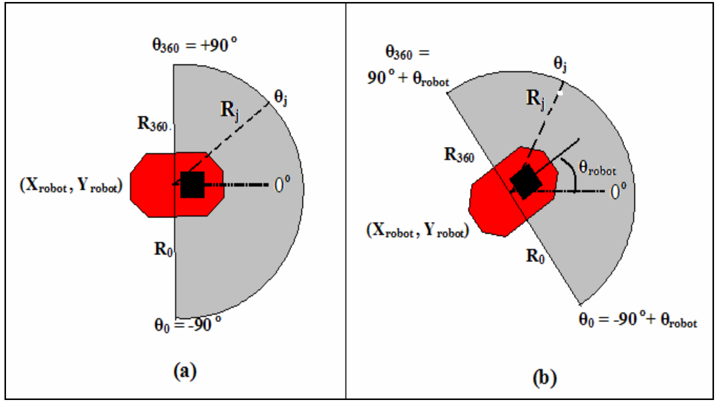


Figure 4.3 Computation of θ_j with (a) zero θ_{robot} , (b) non-zero θ_{robot}

Using *Equ 4.4*, the position of laser reading (X_j, Y_j) in *Figure 4.2* can be found,

$$X_j \text{ (sample no. j+1)} = X_{robot} + R_j * \cos \theta_j$$

$$= X_{robot} + R_j * \cos((j/2 - 90^o)*\pi/180^o + \theta_{robot}) \quad \text{--}Equ\ 4.5$$

$$Y_j \text{ (sample no. j+1)} = Y_{robot} + R_j * \sin \theta_j$$

$$= Y_{robot} + R_j * \sin((j/2 - 90^o)*\pi/180^o + \theta_{robot}) \quad \text{--}Equ\ 4.6$$

With *Equ 4.5* and *Equ 4.6*, the location for every sample of laser reading (X_j, Y_j) can be computed, and thus the closed curve bounding the sensoring area can be drew into environment model later.

4.3.1.2 Interpretation of laser reading into knowledge

Refer to *Figure 4.4*, the laser reading data acquired with a value of 4 meters indicates the cell is absolutely emptied. On the other hand, if the data range, R_j is shorter than 4m, then this implies the cell may be either occupied or emptied.

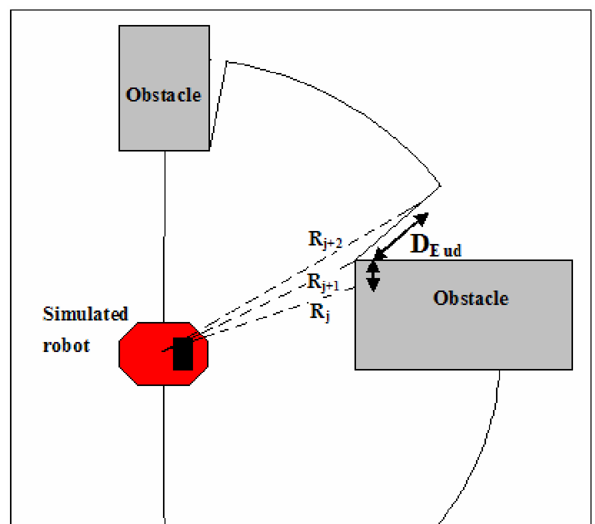


Figure 4.4 Information extraction from laser reading

As a result, to distinguish if the cell is occupied or emptied, apart from laser reading data (R_j), the Euclidean distance between two successive points need to be taken into account as well. If the Euclidean distance (D_{Eud}) between the current point (sample no. j) and next successive point (sample no. j+1) is smaller than 0.2m and the range of current reading (R_j) is less than 4m, then the cell evaluated from the current reading is occupied. Otherwise, the cell is defined as freespace.

```
If (R_j = 4)
        then (X_j,Y_j) is emptied

else if (Euclidean distance between (X_j,Y_j) and (X_j+1,Y_j+1) <0.2)
        then (X_j,Y_j) is occupied

else
        then (X_j,Y_j) is emptied
```

Pseudo Code 4.1 Interpretation of laser reading into knowledge

Pseudo Code 4.1 describes a methodology of interpreting laser reading into knowledge about property of a cell. The profile of the size, shape, distance and orientation of the obstacle in the scanned area thus can be estimated.

4.3.1.3 Determination of occupancy value

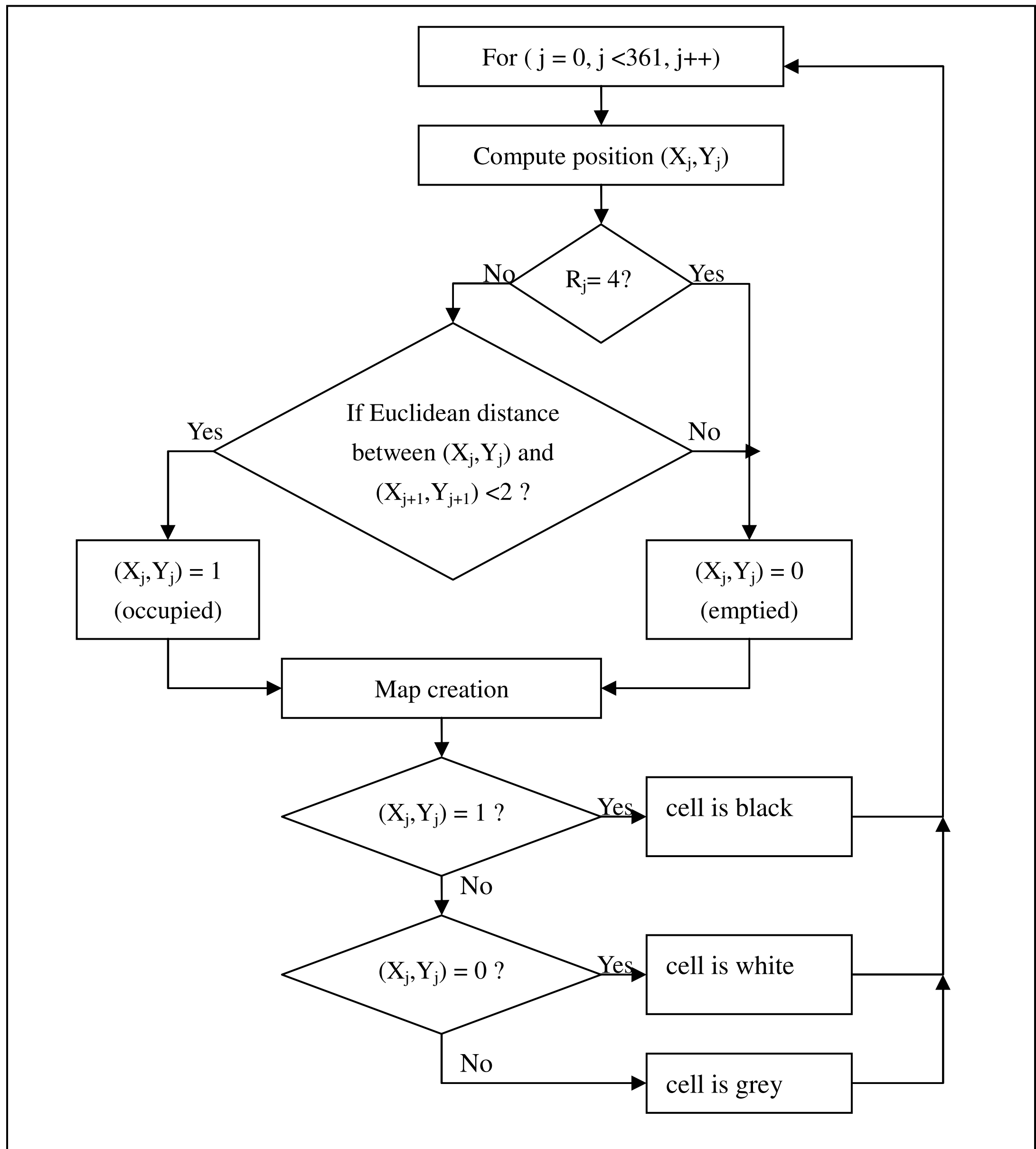
The objects are small areas tessellating the space and the basic property is the fact that a cell is occupied or not. For simplicity, cells are given certain occupancy values, where 0 represents an empty cell, 1 represents a cell that is occupied, and 0.5 is an unknown or unexplored cell.

In this map building algorithm, to produce an up-to-date map, all occupancy values will be updated, in such a way simply overwrote or replaced by new information.

4.3.1.4 Map building using PNGwriter Graphic Library

Lastly, the image of map model can be created using the PNGwriter Graphic Library. The image presents an occupancy grid map (Metric Map) with 160x160 pixels/cells. With occupancy value of 0.5, the cell will be grey in colour. Whereas, for occupancy value of 0 and 1, the cell is white and black respectively.

4.3.2 Map building algorithm



Flow Chart 4.1. Map building algorithm

The complete map building algorithm is illustrated in *Flow Chart 4.1*. For every sample of laser reading, the position of the laser reading is firstly computed, then interpreted into property of cell as either occupied or emptied using occupancy value [0,1], finally specific colours are given for map model rendering using PNGwriter Graphic Library.

4.4 Autonomous Navigation Algorithm for Map Exploration

4.4.1 Autonomous navigation architecture

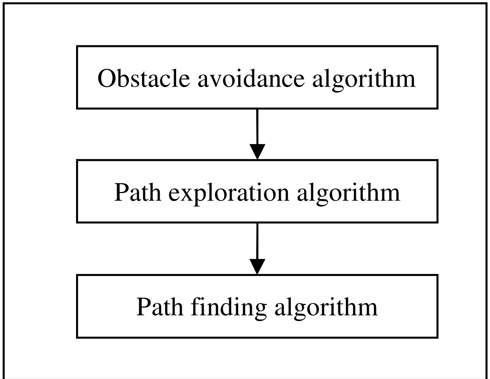
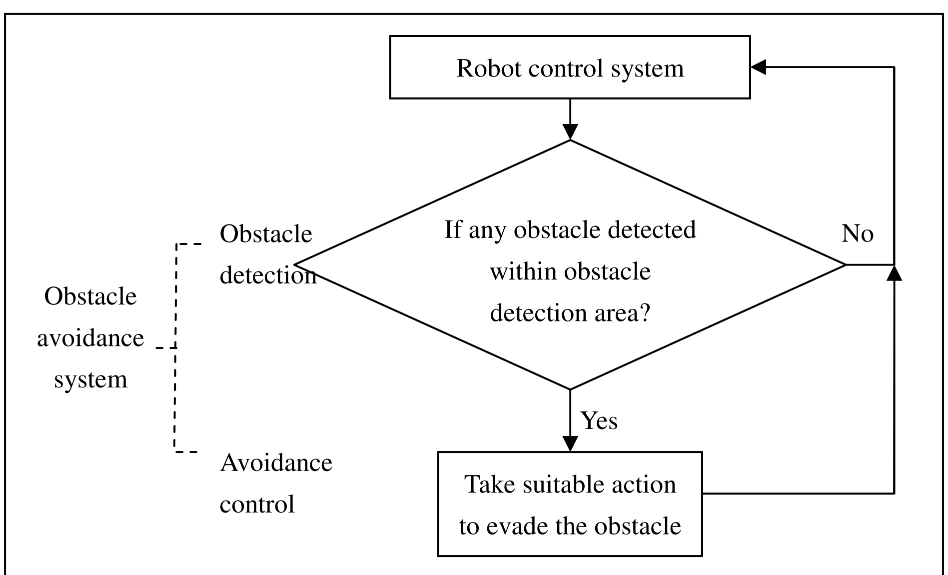


Figure 4.5 Autonomous navigation architecture

In the robot control system, an autonomous navigation system is used for map exploration in an unknown environment. Hence, obstacle avoidance and path finding are essential parts in avoiding uncertainty obstacle and navigating robots effectively. The navigation architecture is shown in *Figure 4.5*.

4.4.1.1 Obstacle avoidance algorithm



Flow Chart 4.2 Obstacle avoidance algorithm

In this obstacle avoidance system, the simulated robot simply avoid the obstacle in front. It can be divided into 2 parts, obstacle detection and avoidance control (*Flow Chart 4.2*).

For obstacle detection part, initially the maximum detection range needed to be considered, in such a way that the robot is able to move very close toward the front obstacle without hitting it. Next, the detection angle also needed to be determined, so that the robot only will avoid the front obstacle that located within certain angle.

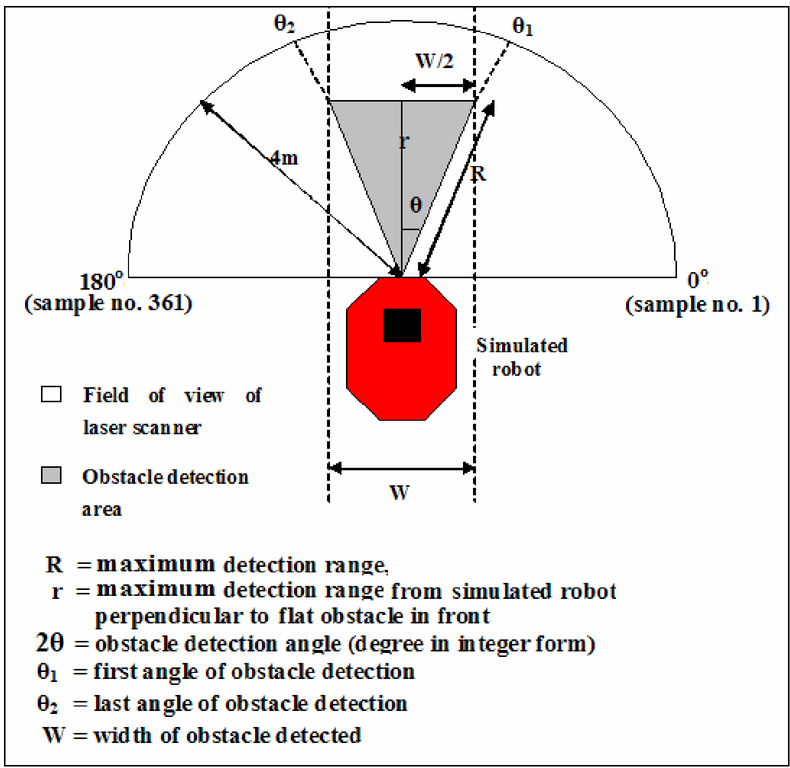


Figure 4.6 Obstacle detection area

Figure 4.6 shows the obstacle detection area of simulated robot. The simulated robot is a polygon with approximately 0.4m in width. Intuitively, the width of obstacle detected (W) must be larger than width of simulated robot to avoid obstacle collision. Hence, the value of W will be set to 0.5 meters. In considerating of time delays, the maximum detection range (R) will then set to 0.6 meters. so that enabling the simulated robot to move towards until 0.6m far from front obstacle.

To obtain obstacle detection area desired, some mathematical calculations must be applied as following:

Assume θ is small enough, that $r \approx R$,

Given that W=0.5 and R=0.6,

$$\sin \theta = W / (2*R)$$

$$\theta \approx 25^o$$

$$\theta_1 = 90^o - \theta = 65^o$$

$$\theta_2 = 90^o + \theta = 115^o$$

Since the resolution of laser sensor is 0.5^o,

thus, the sample no. corresponding to $\theta_i = 2*\theta_i + 1$

sample no. of $\theta_1 = 2*\theta_1 + 1 = 131$

sample no. of $\theta_2 = 2*\theta_2 + 1 = 231$

The obstacle detection area is thus a cone shape, with a radius of 0.6m and angle of 50°, start from 65° (sample no. 131) until 115° (sample no. 231). Once an obstacle is detected within this area (*Pseudo Code 4.2*) irrespective of the type of obstacle, reasonable avoidance action (*Pseudo Code 4.3*) will be taken by turning the robot at its current position until there is no obstacle detected, it then move forward for another 2 seconds before it can perform other tasks.

The direction of turning depends on the position of the obstacle. If the minimum laser reading of sample number 131-181 is smaller than that of sample number 181-231, indicating the right upper corner of robot closer to the obstacle, then the robot should turn to the left, and vice versa.

```
For (laser sample no. 143 until 219)
{
        if laser reading[sample no.] < 0.6
                then obstacle is detected
}
```

Pseudo Code 4.2 Obstacle detection

```
If obstacle detected
{
        for (laser sample no. 131 until 181)
                find minimum laser reading, MinRight

        for (laser sample no. 181 until 231)
                find minimum laser reading, MinLeft

        if (MinLeft <= MinRight)
                turn to right until no obstacle detected
        else
                turn to left until no obstacle detected

        move forward for 2 seconds
}
```

Pseudo Code 4.3 Avoidance control

4.4.1.2 Path exploration algorithm using modified DFS paradigm

The path exploration algorithm is vital for searching navigable paths/points and determining which path/point should robot move to for map acquisition without revisiting. To compute the navigable points, first, the interpretion of sensory data into knowledge must be done (as explained in *section 4.3.1.1* and *section 4.3.1.2*) to determine free space and obstacle. The middle point (X_{mid},Y_{mid}) between 2 obstacles (X_1,Y_1) and (X_2,Y_2) then is computed and evaluated as navigable point (*Equ 4.7*).

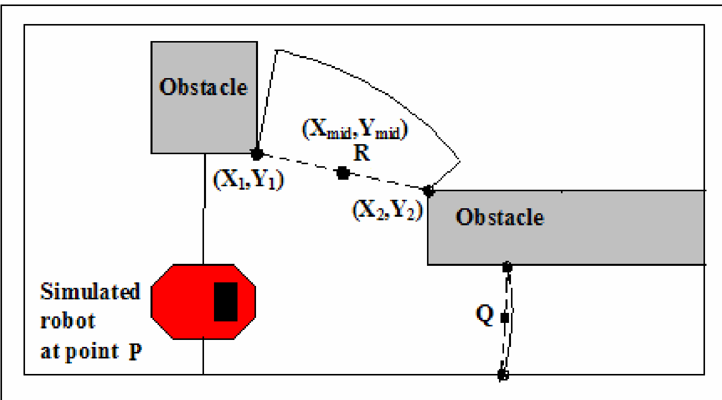


Figure 4.7 Determination of navigable points for path planning

From *Figure 4.7*,

$$(X_{mid}, Y_{mid}) = ((X_1+X_2)/2 , (Y_1+Y_2)/2) \text{ ---------------------------------- } \textit{Equ 4.7}$$

Next, by using the Depth-First Search (DFS) concept, one of navigable points is chosen and followed until an dead end is reached, the robot then backs up until the point ("parent") has unvisited "child" and continue the unvisited "child". Hence, the path generation by robot in *Figure 4.7* is P – Q – P – R.

To reduce map exploration time, the DFS algorithm is modified to eliminate redundancy. In *Figure 4.7*, both R and Q are childs of P, the robot reverse directly from Q to R without passing through their parent P. The path will then be P – Q – R.

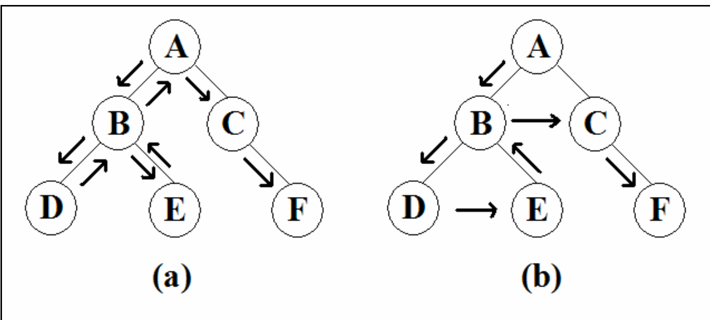


Figure 4.8 Path generated using
(a) DFS algorithm
(b) modified DFS algorithm

To explain this further, *Figure 4.8(a)* and *Figure 4.8(b)* show the comparison of the paths generated using DFS and the Modified DFS algorithm respectively. Obviously, the path generated using Modified DFS is shorter than that of DFS due to shortcut between D-E and B-C.

Furthermore, the Modified DFS algorithm is created in such that it can deal with complex and cross-link paths, as shown in *Figure 4.9(a)*. Once an unvisited "child" is being scanned twice, the "child" will be erased to avoid revisiting.

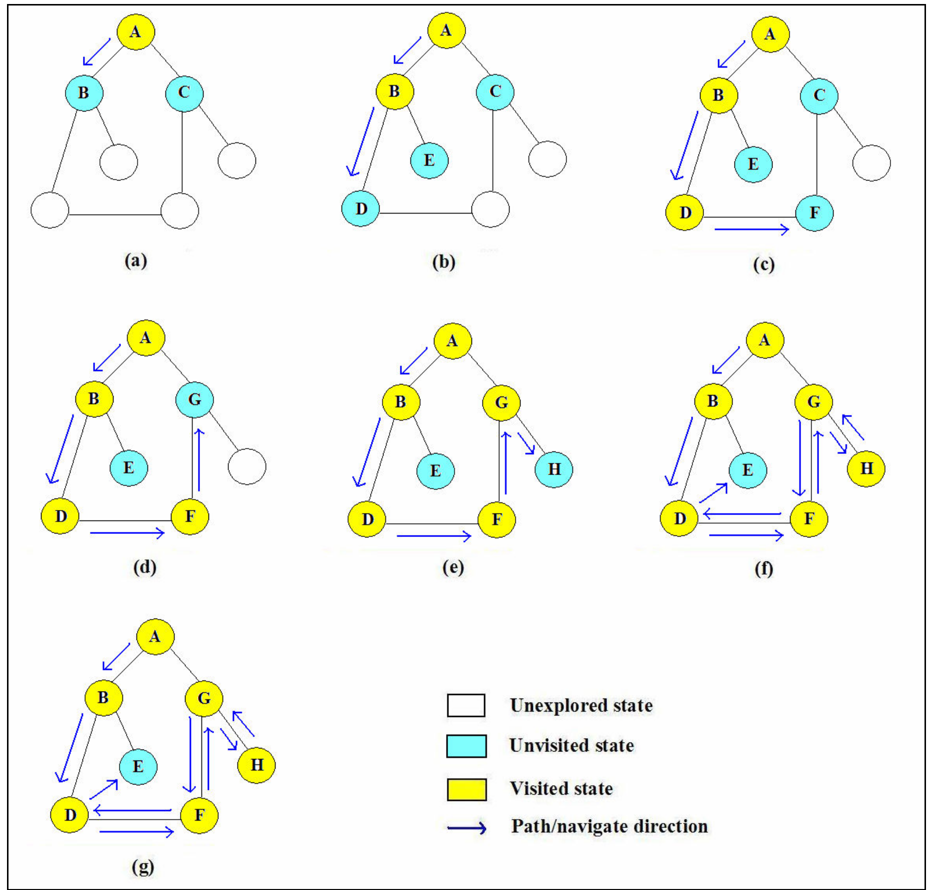


Figure 4.9 Modified DFS algorithm for complex and cross-link paths exploration

Below are the steps of path exploration for *Figure 4.9*:

(a) Initial node A is marked as "Visited", and the robot scans 360° and finds the unvisited child node B and C, B is chose to navigate to.

(b) Reaches B, marks B "Visited", scans and finds out unvisited child D and E, chose to forward to D.

(c) Reaches D, marks D "Visited", scans and finds out unvisited child F, navigate to child F.

(d) Reaches F and marks F "Visited", scans, seeks out child G and discovers C being re-scanned twice, C is erased and replaced by G, move to G.

(e) Reaches G and marks G "Visited", scans, discovers unvisited child H, move to H.

(f) Reaches H and marks H "Visited", scans, discovers it is a dead-end without further unvisited child, decides to reverse to unvisited E via G (H's parent), F (G's parent) and D (F's parent).

(g) Reaches E and marks E "Visited", scans and dead-end is found, since there is no other unvisited nodes, implies that all possible paths have been explored and visited, robot terminates.

In complex and cross-link paths, the robot may have a few of transition paths from one node to another. For an instance, at step (f) above, when the robot reverse from node H to node E, there are more than one navigation paths, i.e. H – G – F – D – E path, H – G – A – B – E path. To fasten the navigation time, a shortest path should be evaluated. A path planning algorithm herewith play an essential role, which will be discussed later.

To put into practice, the modified DFS path exploration algorithm can be implemented using a state machine as shown in *Figure 4.10*.

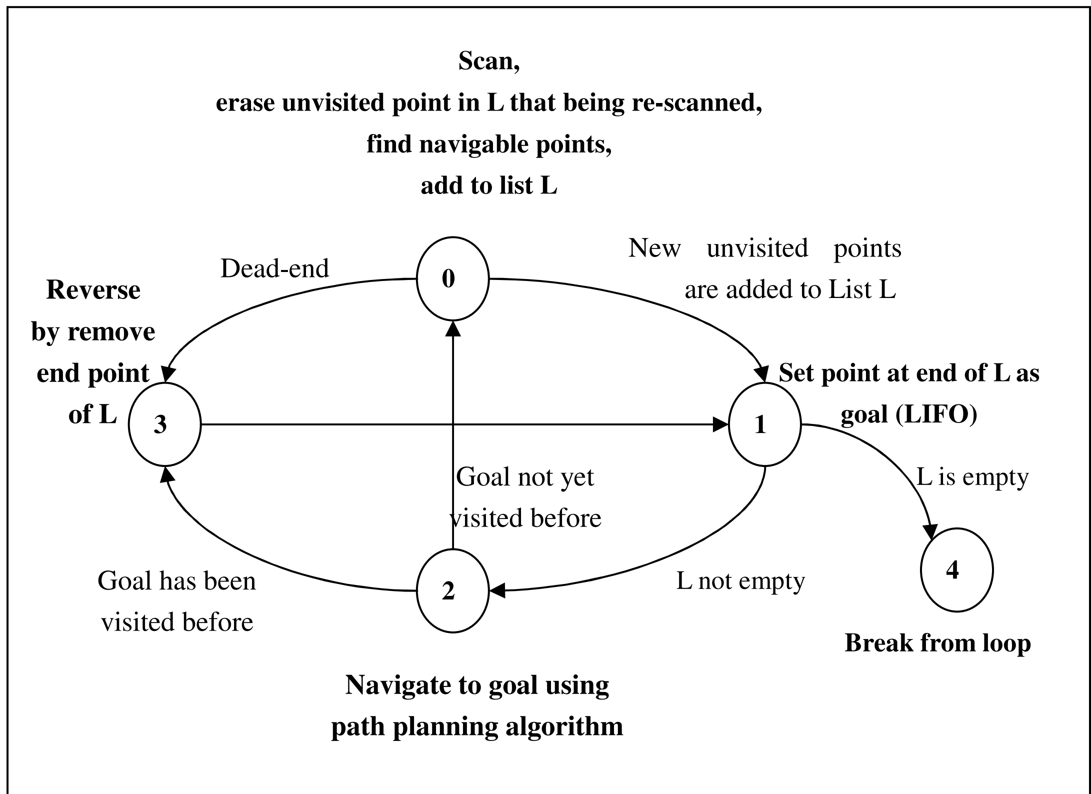


Figure 4.10 State machine of path planning algorithm using modified DFS

```
state = 0;
list L = empty

while ( state !=4 )
{
    if state = 0
        scan 360°
        if an unvisited point in L is re-scanned
            remove that point from L
        determine new navigable points and add into L
        if point at end of L is unvisited
            state = 1
        else
            state =3

    else if state = 1
        if L not empty
            set point at end of L as goal , state = 2
        else
            state = 4

    else if state = 2
        navigate robot towards goal
        if the goal has been visited, state = 3
        else mark goal as visited, state = 0

    else if state = 3
        remove point from end of L, state = 1
}
```

Pseudo Code 4.4 Path exploration algorithm using modified DFS paradigm

Below is the step by step implementation along with its pseudo code shown in *Pseudo Code 4.4*:

1. Initially, the state = 0.
2. The robot scans 360° and checks if any unvisited point in list L is located in current scanning field. If yes (means that point is linked to current point), remove that point from list L. Otherwise, do nothing.

3. Determines all navigable points using *Equ 4.7*, then push only new and unvisited points into list L. If the last point in list L is unvisited (indicates new unvisited points are added), then state = 1 (go to step 4). Otherwise, robot is assumed experiences a dead-end, state = 3 (go to step 7).
4. If the list L is not empty, using LIFO (Last -In First-Out), set the point in the end of list L as goal, state = 2 (go to step 5). Otherwise, map building is assumed complete, state = 4 (go to step 8).
5. Navigate robot towards goal with assist of a path planning algoritm
6. Repeat step 5 until the robot reaches the goal, if the goal originally mark unvisited, then mark it as visited and state = 0 (go to step 2). If the goal already marked as visited, then state = 3 (go to step 7)
7. Pop out the point in the end of list L, state = 1 (go to step 4)
8. Break from while loop

4.4.1.3 A path finding algorithm*

Once a goal is determined, the A* path finding algorithm is implemented to plan a shortest and safest route from the initial state to goal. This improves efficiency, provides time saving and removes the obstacle collision problem which the robot may face without A* algorithm.

Instead of searching every point, A* expands the node/state that appears to be closest to the goal and avoids expanding paths that are already expensive. A* uses evaluation function (*Equ 4.8*) to select which nodes/state to expand.

$$f(n)=g(n) + h(n) \qquad \text{--} \; \textit{Equ 4.8}$$

where g(n) - the cost (so far) to reach the node n

h(n) - estimated cost to get from the node n to the goal

f(n) - estimated total cost of path through n to goal

Two fundamental issues need to be known before getting start to A* algorithm:

1. Method for estimating heuristic, H *[62]*

Manhattan Method (*Equ 4.9*), Diagonal Shortcut Method (*Equ 4.10*) and Euclidean Distance Method (*Equ 4.11*) are widely used methods for calculating heuristic, H. The closer estimated H is to the actual remaining distance along the path to the goal, the faster A* will find the goal.

Manhattan Method:

$$H = 10 * (\Delta x + \Delta y) \qquad \text{--------} \quad \textit{Equ 4.9}$$

Diagonal Shortcut Method:

$$H = \begin{cases} 14*\Delta y + 10*(\Delta x - \Delta y)\text{ , if } \Delta x > \Delta y \\ 14*\Delta x + 10*(\Delta y - \Delta x)\text{ , otherwise} \end{cases} \qquad \text{--------} \quad \textit{Equ 4.10}$$

Euclidean Distance Method:

$$H = 10 * \sqrt{(\Delta x^2 + \Delta y^2)} \qquad \text{--------} \quad \textit{Equ 4.11}$$

where Δx = abs(currentX-targetX), Δy = abs(currentY-targetY)

2. G scoring and arrow interpretation

In *Figure 4.11(a)*, a cost of 1 should be assigned to G for each horizontal or vertical move, and a cost of 1.414 (√2) for a diagonal move. For simplicity sake, 10 and 14 are used as shown in *Figure 4.11(b)*.

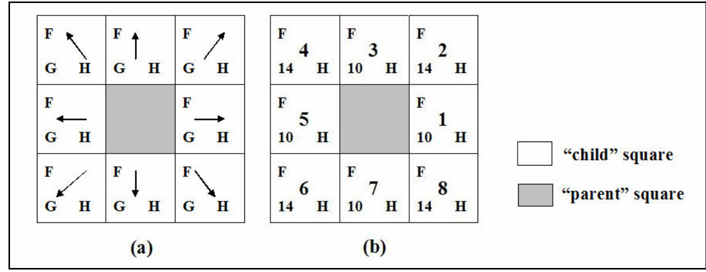


Figure 4.11 G scoring and arrow interpretation

The arrows shown in *Figure 4.11(a)* indicate all of the neighbouring squares are "child" squares of the middle grey square ("parent" square). This parent square pointing is important for tracing path. To interpret the arrow pointing into word for programming, chain coding with 8-connectivity representation *[37, 38]* is used to define the position of one square relative to its "parent" square, as shown in *Figure 4.11(b)*.

```
open list O = empty, closed list C = empty, bool success = true, int state = 1
add starting square into list O
while (current point !=goal)
{
        if (state =1)
                if O is empty, success = false, break
                else, selects lowest F square in O as "parent", moves it form O to C
                state = 2
        else
                for (each "child" square)
                        if ("child" square is walkabe and not in C)
                                if ("child" square not in O)
                                        calculate F, G, H, add to O with chain code
                                else
                                        if(new G < existing G)
                                                replace with new chain code, F, G, H
                                if "child" square is 8th, state = 1
}
if (success), find path form C by working backwards from goal to starting square
```

Pseudo Code 4.5 A path planning algorithm*

Below is the implementation of the A* algorithm (refer to *Pseudo Code 4.5*):

1. The initial square is added to open list.
2. If the open list is empty, go to step 5. Otherwise, select a square with lowest F cost in open list, set it as "parent" square and swap it to closed list.
3. For each of the 8 adjacent/"child" squares to this "parent" square,
   i. if it is the goal, adds it to closed list, with chain code as well, go to step 4.
   ii. if it is not walkable or it is in the closed list, skip step iii and iv.

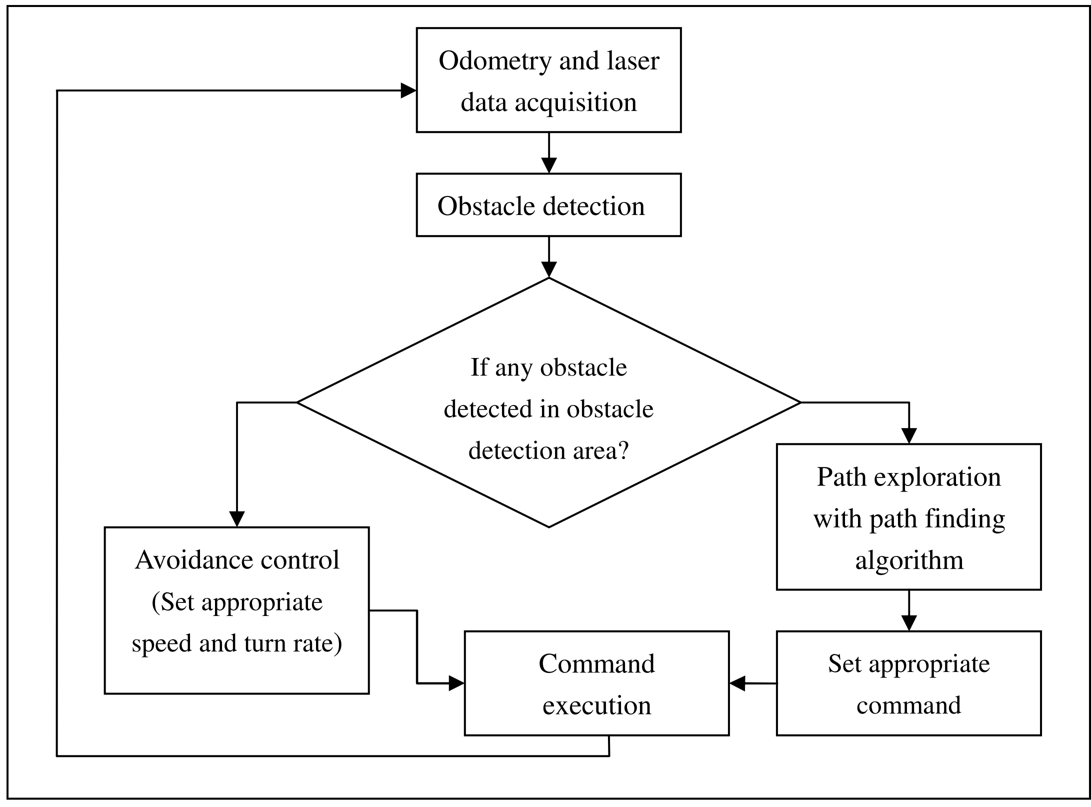
iii. if it is not in open list, adds it to the open list. Simultanously, the F, G and H scores of the square are computed, and appropriate chain code (points to current "parent" square) is recorded.

iv. if it is in the open list already, check if new calculated G lower than existing G. If so, means the new G gives better path, replace an appropriate chain code to the existing square (indicates current "square point" it should point to), and recalculate the G and F scores.

v. if it is the 8th "child" square, go to step 2. Otherwise, go to step 3.

4. From closed list, find out the goal square, go reverse to its parent (which chain code point to), then go from that square to its parent again, and so on, until starting square. This is the path. Path finding task terminates.
5. Fail to find goal, path finding task terminates with failure.

4.4.2 Autonomous navigation algorithm



Flow Chart 4.3 Autonomous navigation algorithm

Flow Chart 4.3 shows the autonomous navigation algorithm. First and foremost, the simulated robot checks if any obstacle is detected within the obstacle detection area. If there is an obstacle, then it executes avoidance algorithm. Otherwise, it will proceed to path exploration or path finding processes. The appropraite commands will then be executed.

4.5 Control Architecture – Sense-Plan-Act (SPA) Approach

By using the Sense-Plan-Act approach, the autonomous robotics first attempts to interpret its sensory data to build a model of the world, and next the robot uses the model to plan its actions, and finally it would act on those plans.

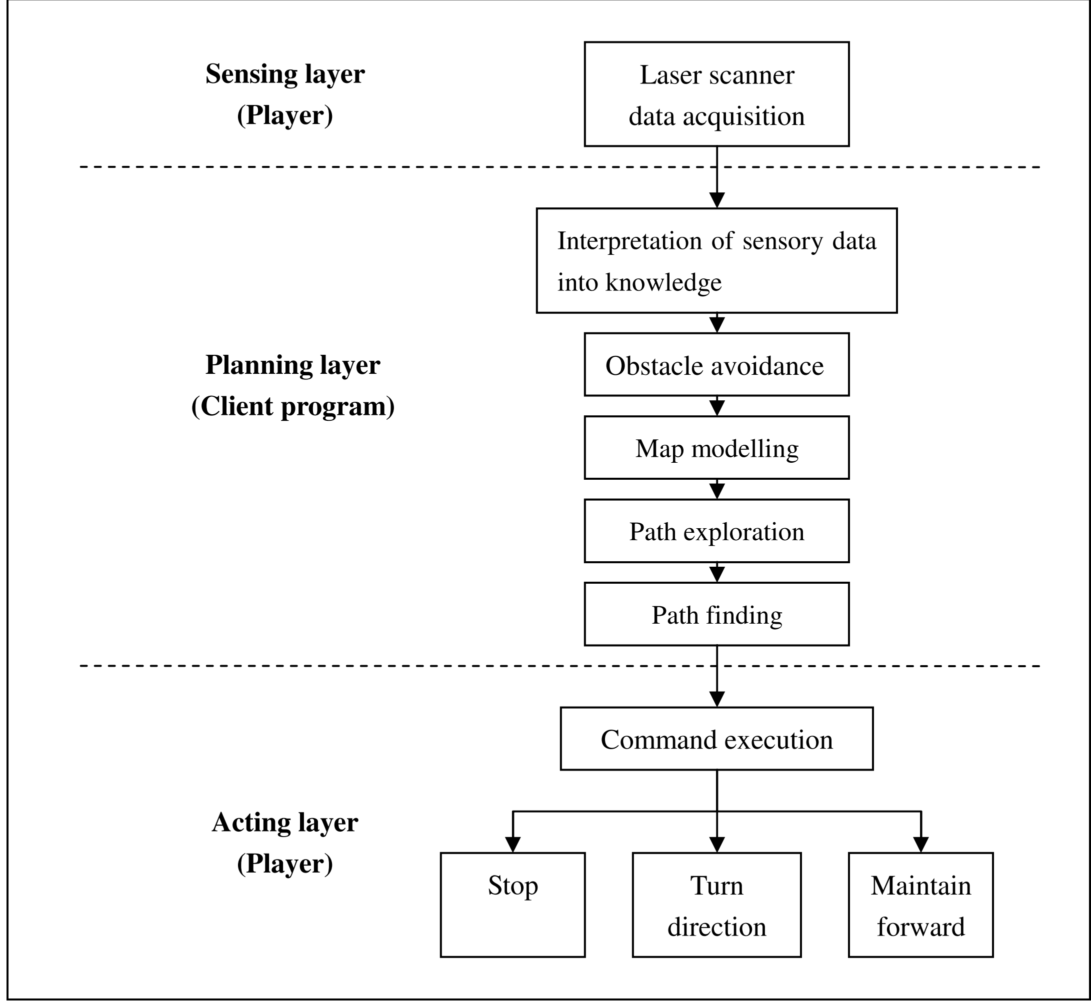


Figure 4.12 Structure of robot control system using SPA paradigm

Figure 4.12 illustrates the structure of robot control system using Sense-Plan-Act concept for autonomous navigation with map building. Obviously, Player provides the benefit of acquiring the laser scanner reading (Sensing Layer) and generating the motion command (Acting Layer).

For the Planning Layer, client program interprets the sensory data into knowledge, implements obstacle detection and executes avoidance action if required. Otherwise, it produces the model map of world. The model map then served as input along with goal (determined by path exploration algorithm) to the path finding process, develops a path for robot. Appropriate speed and direction of robot are evaluated and determined.

Finally, the decided motion command is issued to the Player for execution, which means down to the actuator level (robot motor).

Chapter 5
Map Benchmarking Suite

The accuracy of map model is crucial in estimating the effectiveness and reliability of the map building algorithm. A fitness function must be used to evaluate the quality of the map created in order to guide the map building algorithm. In order to gauge the accuracy and effectiveness of the map building system, a variety of benchmarking methods *[44, 71]* for comprehensive analysis of maps generated have been carried out. The comprehensive suite of map benchmarking techniques includes:

1. Cross Correlation *[28]*
2. Map Score
3. Map Score of Occupied Space

5.1 Cross Correlation

Baron's Cross Correlation Coefficient *[80, 81]* is a basis image comparison algorithm for evaluating map. It is based on template matching, which was initially used for face reorganization *[79]*. Baron's cross correlation coefficient, C_N (*Equ 5.1*), rescales the energy distributions of the template and the image, in order to match their averages and variances.

$$C_N(y) = \frac{\langle I_T T\rangle - \langle I_T\rangle\langle T\rangle}{\sigma(I_T)\sigma(T)}$$ --- *Equ 5.1*

where $C_N(y)$ = cross correlation coefficient over the area being matched

$<I_T> = \frac{\Sigma\, i_{x,y}}{n}$, average or mean of generated map, I_T

$<T> = \frac{\Sigma\, t_{x,y}}{n}$, average or mean of ideal map, T

$<I_T\, T> = \frac{\Sigma\,(i_{x,y} * t_{x,y})}{n}$, average of 2 combined maps

$\sigma(I_T) = \sqrt{\frac{\Sigma\,(i_{x,y} - <I_T>)^2}{n}}$, standard deviation of a map I_T

$\sigma(T) = \sqrt{\frac{\Sigma\,(t_{x,y} - <T>)^2}{n}}$, standard deviation of a map T

where $i_{x,y}$= value of cell at (x,y) in map I_T

$t_{x,y}$= value of cell at (x,y) in map T

n = number of cells in map I_T

This benchmark is quite robust to noise, and can be normalised to allow pattern matching independently of scale and offset in the images. A higher coefficient value indicates the map being tested has a high degree of similarity to ideal map.

However, Cross Correlation also has drawback of having possibility to get high correlation value even when the map generated is inaccurate. This is due to the fact that it factors the average and standard deviation of the map, instead of cell by cell comparison.

For an instance, *Figure 5.1(a)* shows an ideal map and *Figure 5.1(b)* shows a generated map with curved obstacle distorted by robot odometry error, even they are

quite difference, these two maps' average values are similar through coincidence as they have equal number of occupied and emptied cells, as a consequence, they might be given a high correlation value.

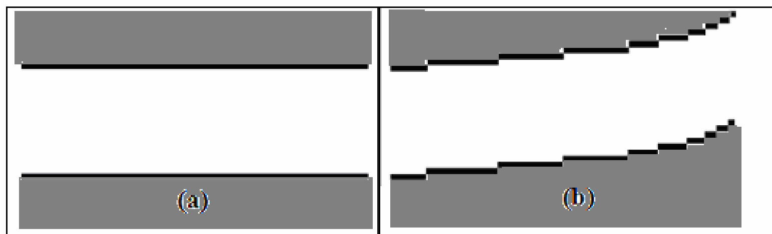


Figure 5.1 A typical corridor (a) ideal map, (b) generated map with curved obstacle (source from [44])

5.2 Map Score

Martin and Moravec *[78]* proposed Map Score, a map comparison measure specifically for probabilistic maps. Unlike correlation, map scoring calculates the difference of two maps based on a cell-by-cell comparison. The lower the difference, the greater the similarity is. Given two maps, M and N, the score between them is calculated using equation:

$$Match = \sum_{m_{X,Y} \in M, n_{X,Y} \in Y} \left(m_{X,Y} - n_{X,Y}\right)^2$$

--- *Equ 5.2*

where $m_{X,Y}$ = value of the cell at position (x,y) in map M

$n_{X,Y}$ = value of the cell at position (x,y) in map N

The weakness of this map matching technique is that it overestimates the empty regions of space. This is because in many environments, there are large amounts of unoccupied spaces, with a few small obstacles distributed within that space. Even if the sensor model misses an obstacle, it is only slightly increase score value, since it

computes the wrong value for small number of cells.

5.3 Map Score of Occupied Cells

To remedy the weakness of the Map Score explained above, the Map Score is modified in such a way only testing the correctness of the obstacles in the map, ignoring the free space areas. For any two maps M and N, if either the value $m_{X,Y}$ >0.5 or $n_{X,Y}$ >0.5, then the squared difference between those two cells is added to the final score. Otherwise, they are ignored.

This technique is able to indicate the strengths and weaknesses of the laser model used. If the sensor model misses an obstacle that it should detect, it will give a worse (higher) score. Besides, if the reading is too long which results in shadows behind the obstacle, then the worse score will be given as well.

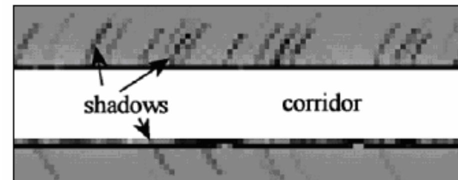


Figure 5.2 A typical corridor with 'shadow' (source from [44])

In *Figure 5.2*, when scoring using Map Score method, it would seem to be a very good map, again due to well-defined and large amount of the free space. However when comparing just the occupied cells, the score will be much more higher since there are many more occupied cells than there should be.

Unfortunately, both Map Score and Map Score of Occupied Cells methods rely on the two maps being in the exact same orientation and translation, with no odometry error or with very effective localisation algorithms.

Chapter 6
Empirical Evaluation

6.1 Obstacle Avoidance Algorithm

The obstacle avoidance system is a key part of the control system and it is sensitive to time delays. A lack of synchronisation will effectively end-up in a collision.

6.1.1 Ideal case

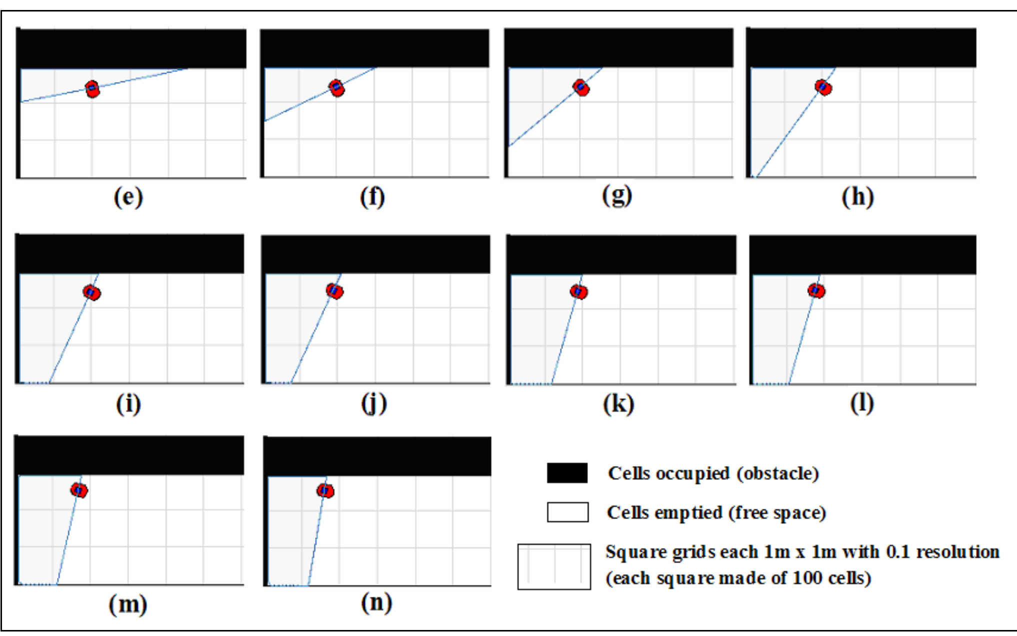


Figure 6.1 Ideal obstacle detection and avoidance

Figure 6.1 demonstrates the response of the mobile robot once the obstacle is detected within a preset range (0.6m in front of obstacle) in an ideal case without any delay. However, mobile robots nowadays are designed to be autonomous, intelligent and robust, so that it has capability of executing series of tasks. Thus, the processes comprise a series of computational functions, which may cause time delay and affect the performance of obstacle avoidance system.

6.1.2 Case of asynchronous lower layer functions

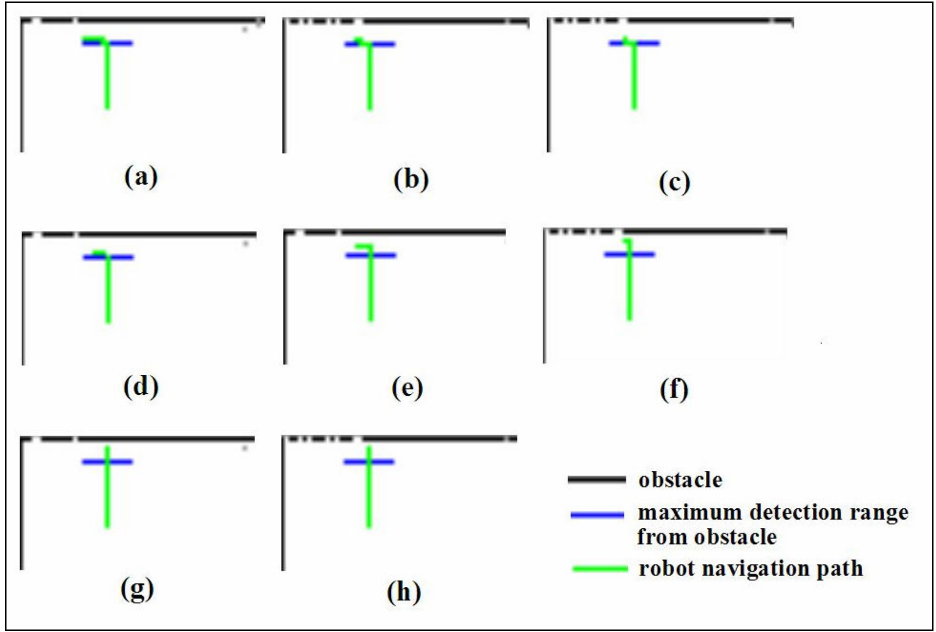
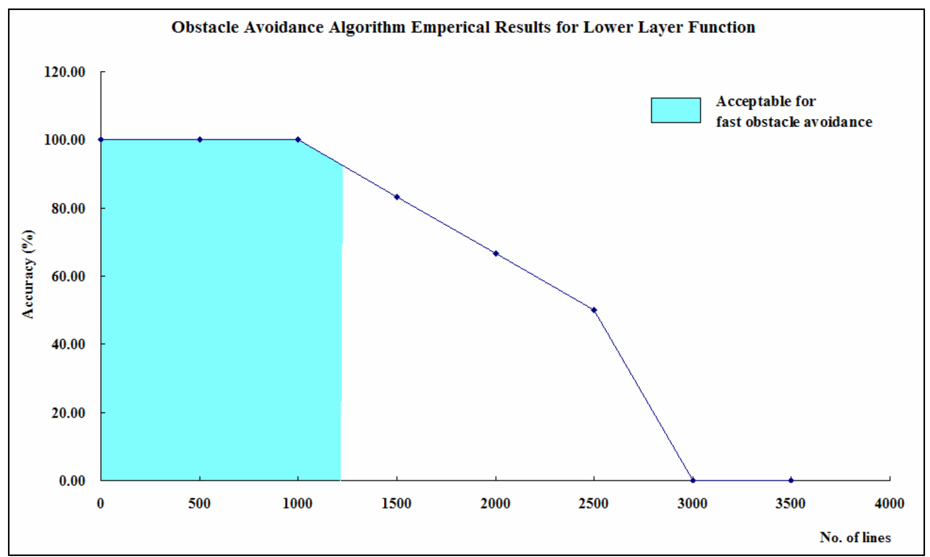


Figure 6.2 Series of obstacle avoidance algorithm empirical results for (a) ≈0, (b) ≈500, (c) ≈1000, (d) ≈1500, (e) ≈2000, (f) ≈2500, (g) ≈3000, (h) ≈3500, lines of lower layer functions

Figure 6.2 shows the case of asynchronous processes which execute simple computational functions for lower layer functions, such as wall following. Obviously, *Figure 6.2(a)*, *Figure 6.2(b)* and *Figure 6.2(c)*, which execute approximately 0, 500 and 1000 lines of computational instructions respectively, are able to perform real time obstacle detection and avoidance. This is due to the processes can be executed within one clock pulse.

Figure 6.2(d) describes the robot addresses to time delay problem due to execution of approximately 1500 lines of computational instructions, thus it only executes obstacle detection and avoidance at 0.5m away from obstacle. This become worse for *Figure 6.2(e)* and *Figure 6.2(f)*, obstacle avoidance only executed when the robot is 0.4m and 0.3m respectively far from obstacle.

When the robot executes more than 3000 computational instruction lines, due to long computational time, the robot is not able to detect obstacle within reasonable time. As a consequence, the robot crashes with the obstacle and become jammed.



Graph 6.1 Obstacle avoidance empirical results for lower layer functions

Graph 6.1 outlines the performance of obstacle avoidance system in term of number of simple computational instruction lines executed using formula below:

$$\text{Accuracy} = \begin{cases} \frac{D}{0.6} \times 100\% , & D>0.2 \\ 0\% , & D<=0.2 \end{cases}$$ --------------------------------*Equ 6.1*

where D = the distance between robot and obstacle where avoidance action took place

For the sake of executing fast obstacle avoidance, high accuracy of at least 90% must be obtained such that the robot able to detect the obstacle once it reaches 5.54m – 6m far from the obstacle, and appropriate action can be took instantly to avoid collision.

6.1.3 Case of asynchronous higher layer functions

In the case of executing higher layer functions, such as path planning and map building which embeds file reading and writing processes, the time consumed is much higher than that of lower layer functions.

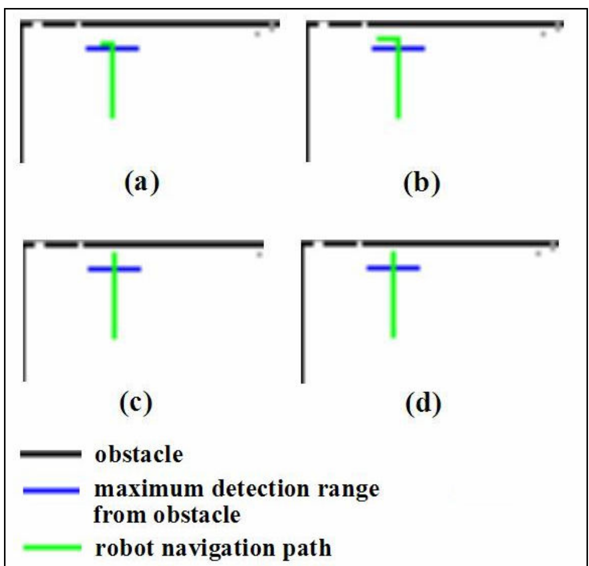
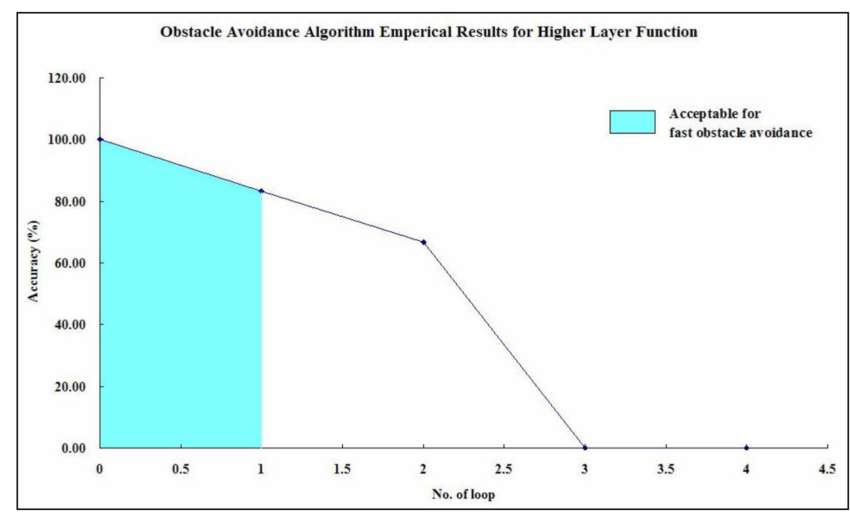


Figure 6.3 Series of obstacle avoidance algorithm empirical results for (a) 1, (b) 2, (c) 3, (d) 4, higher layer functions

Figure 6.3(a) shows robot detects the obstacle at 0.5m from it when implementing one higher layer function, gives accuracy of 83.33% using *Equ 6.1*. When implementing two higher layer functions (*Figure 6.3(b)*), obstacle detection occurs at 0.4m with accuracy of 66.67%. Besides, the robot collides with obstacle when implementing more than two higher layer functions as shown in *Figure 6.3(c)* and *Figure 6.3(d)*.



Graph 6.2 Obstacle avoidance empirical results for higher layer functions

Graph 6.2 illustrates that the execution of one higher layer function is acceptable for fast obstacle avoidance, which provides accuracy of 83.33%. In other words, execution of more than two asynchronous higher layer functions will significantly affect the performance of the obstacle avoidance system.

6.1.4 Discussion

A fast or real time obstacle avoidance system is able to detect and avoid the obstacle instantly. Conversely, a slow response obstacle avoidance system may not able to detect and avoid the obstacle which it should be. In the worst case, a high time delay due to asynchronous, complex and time-consuming process will cause the obstacle avoidance system lost its function, lead to collision and robot damage. To combat this problem, synchronous processes should be used.

6.2 A* Search Algorithm for Path Finding

There are four important criteria *[31, 45, 50]* for the search algorithm:

1. Completeness – ability of finding solution if one exists
2. Optimality – ability of finding the best solution from a set of possible solutions

3. Time complexity – defines how fast the algorithm perform a search
4. Space complexity – defines how much memory the algorithm requires to perform a search

6.2.1 Evaluation of A* search algorithm

The A* search algorithm is evaluated based on the four criteria for various terrain. Three heuristic estimation methods are evaluated as well for comparison, they are Manhattan Method, Diagonal Shortcut Method and Euclidean Distance Method.

For *Figure 6.4* until *Figure 6.11* (all maps are 160x160 cells in size), the following representation is used:

- source cell/node, destination cell/node
- unexplored free space cell, obstacle
- explored cell (saved in open list)
- explored cell (saved in closed list)
- path evaluated by A* search algorithm

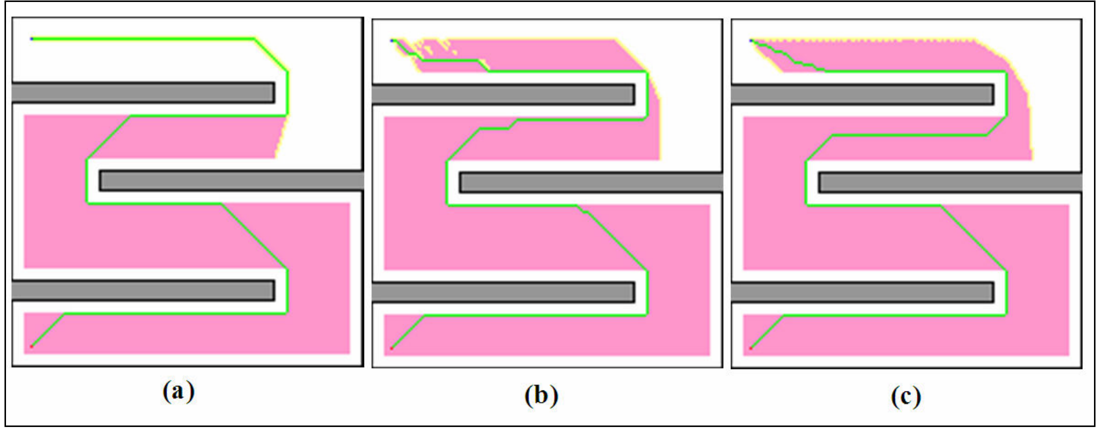


Figure 6.4 A search algorithm evaluation from node (10,10) to (10,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

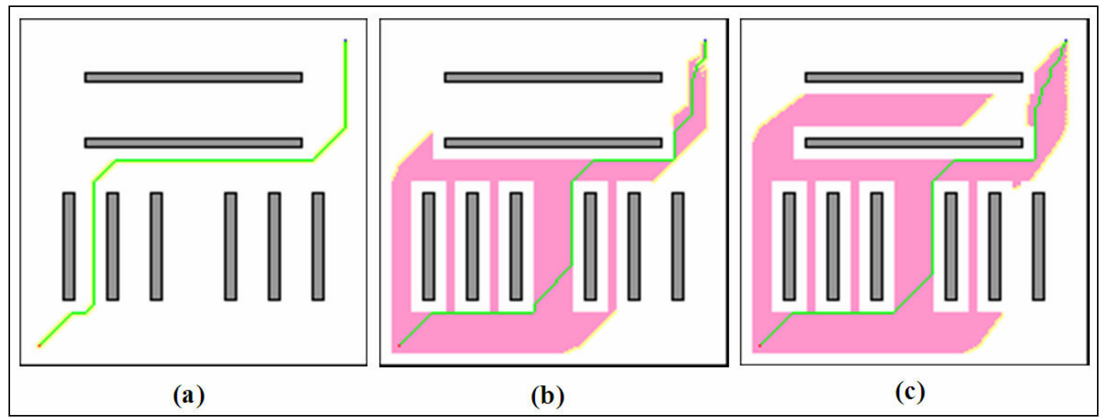


Figure 6.5 A search algorithm evaluation from node (10,10) to (150,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

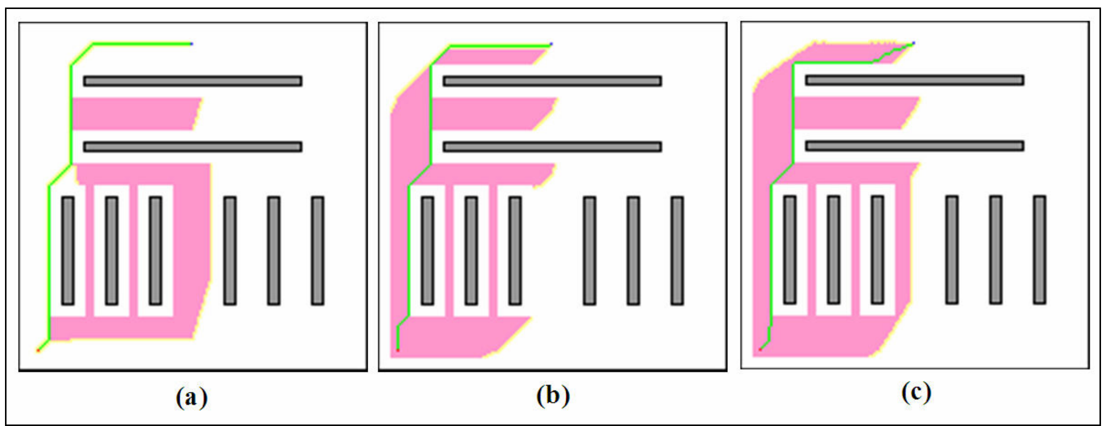


Figure 6.6 A search algorithm evaluation from node (10,10) to (80,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

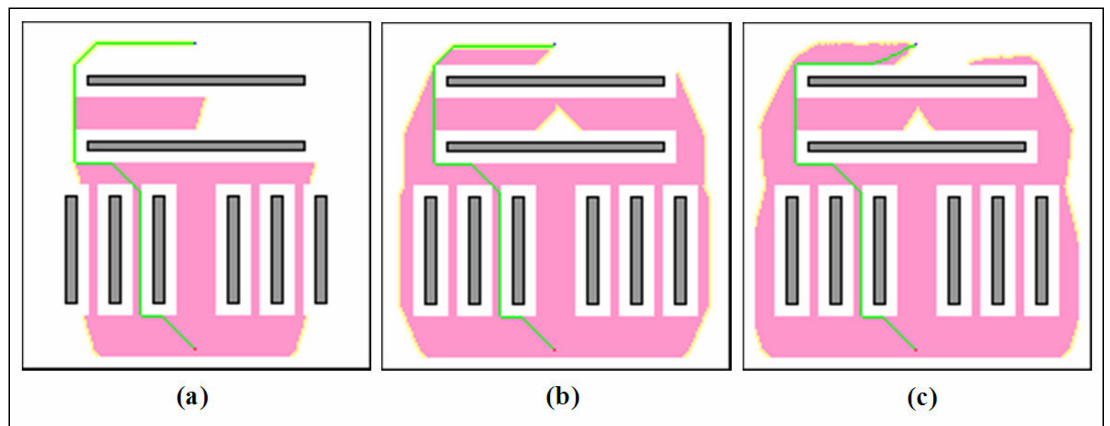


Figure 6.7 A search algorithm evaluation from node (80,10) to (80,150) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

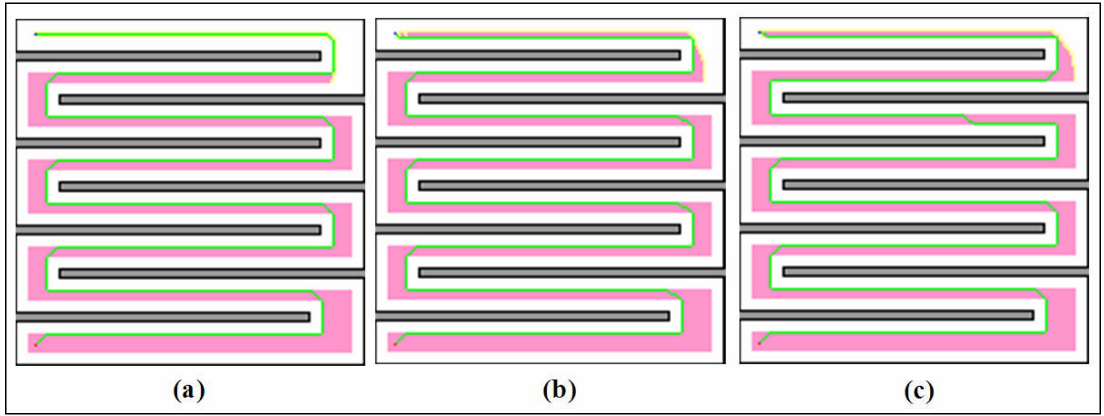


Figure 6.8 A search algorithm evaluation from node (10,10) to (10,153) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

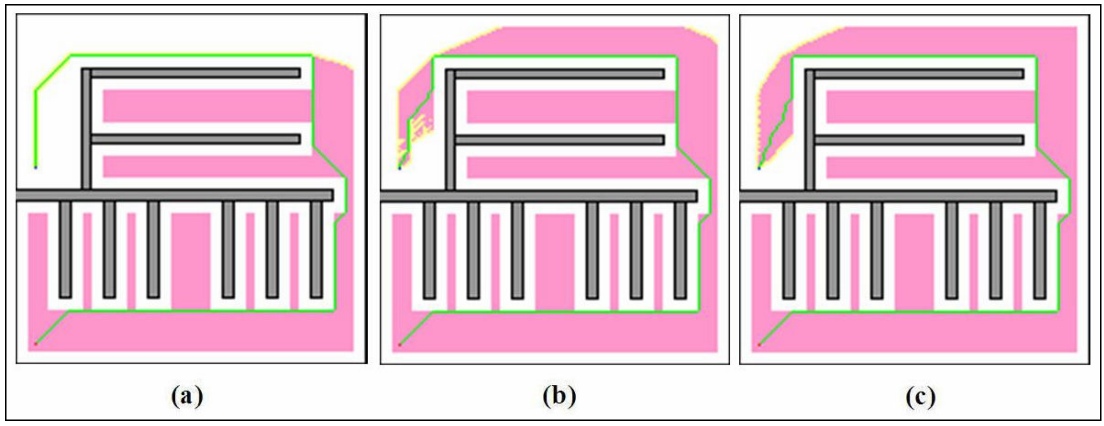


Figure 6.9 A search algorithm evaluation from node (10,10) to (10,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

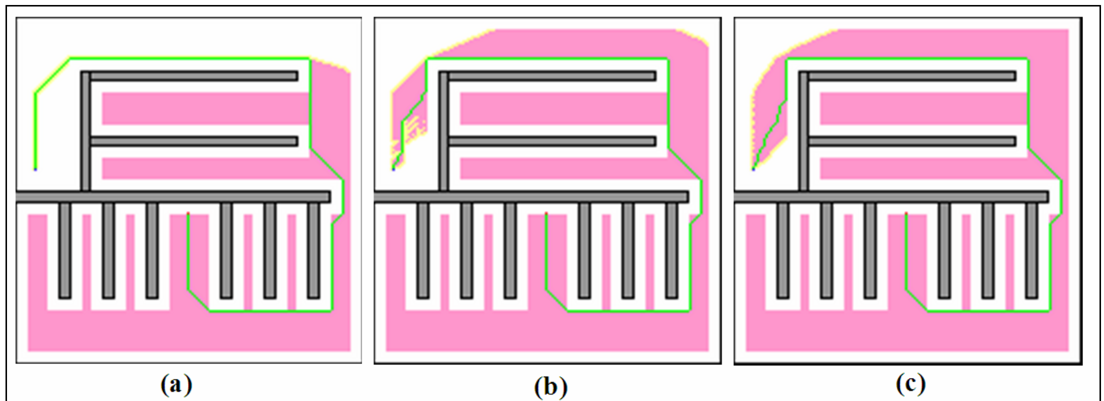


Figure 6.10 A search algorithm evaluation from node (80,70) to (10,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

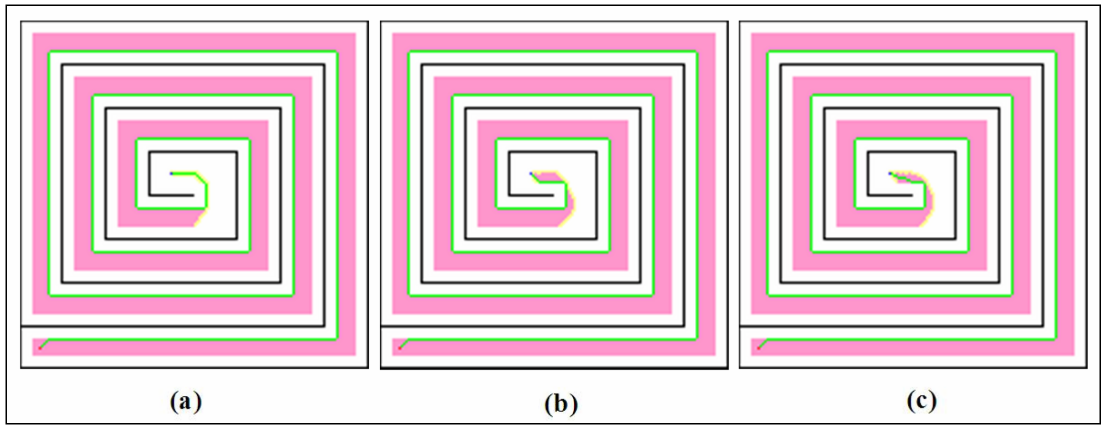


Figure 6.11 A search algorithm evaluation from node (10,10) to (70,90) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

All the figures above show that the A* algorithm is able to find a solution/path if one exists. All measurements are records in *Table 6.1* for comparison.

Figure		**Complete if one exists?**	**Optimal**			**Time taken**	**Space (memory)**		
			No. of cells	**Length**	**No. of turning points**		**Space in close list**	**Space in open list**	**Total space**
6.4	***(a)***	Yes	471	626.2	14	33	10876	307	11183
	(b)	Yes	471	626.2	24	38	12439	264	12703
	(c)	Yes	471	626.2	32	40	12709	261	12970
6.5	***(a)***	Yes	236	312.8	8	2	237	404	641
	(b)	Yes	222	287.6	20	11	5785	246	6031
	(c)	Yes	222	287.6	24	16	7948	254	8202
6.6	***(a)***	Yes	185	249	6	6	3771	488	4259
	(b)	Yes	185	249	8	8	4613	254	4867
	(c)	Yes	185	249	19	9	5339	260	5599
6.7	***(a)***	Yes	210	278	10	16	5874	269	6143
	(b)	Yes	210	278	11	26	9135	488	9623
	(c)	Yes	210	278	23	25	10224	419	10643
6.8	***(a)***	Yes	1146	1586.4	30	14	6761	299	7060
	(b)	Yes	1146	1586.4	37	15	7083	171	7254
	(c)	Yes	1146	1586.4	38	16	7134	172	7306
6.9	***(a)***	Yes	411	554.2	12	23	8696	269	8965
	(b)	Yes	411	554.2	22	28	10567	226	10793
	(c)	Yes	411	554.2	30	27	10863	147	11010

6.10	***(a)***	Yes	376	507.2	13	23	8697	269	8966
	(b)	Yes	376	507.2	23	28	10568	226	10794
	(c)	Yes	376	507.2	31	28	10864	147	11011
6.11	***(a)***	Yes	1134	1578.8	30	33	10698	67	10765
	(b)	Yes	1134	1578.8	31	33	10787	61	10848
	(c)	Yes	1134	1578.8	38	36	10794	63	10857

Table 6.1 A search algorithm evaluation for various terrain using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

From *Table 6.1*, all of the three heuristic methods generally assess same number of cells/nodes for path generated with same length, this mean the paths searched are the best and optimal.

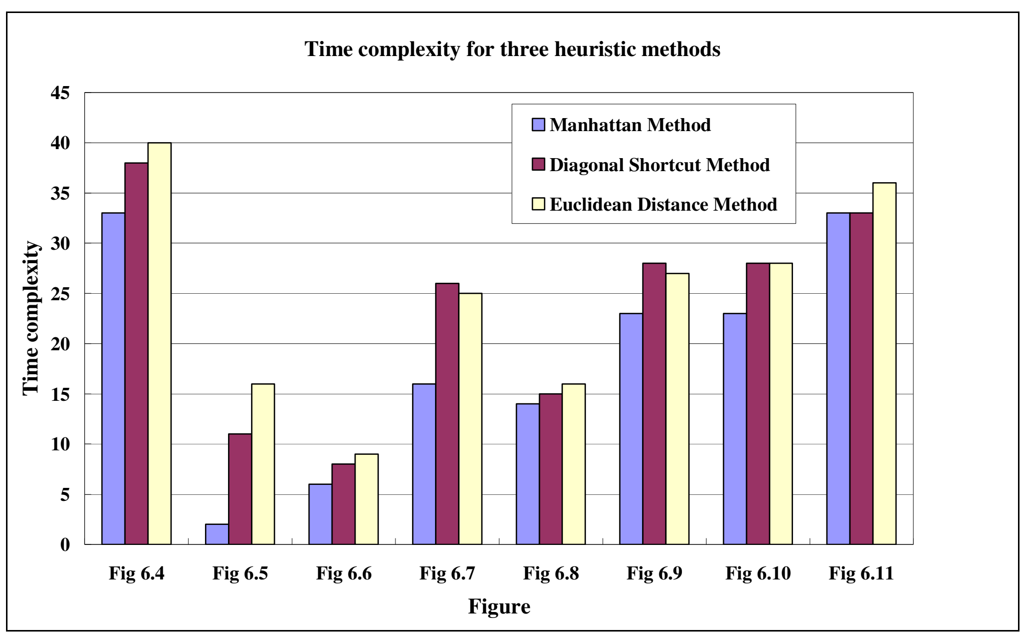
In considering of time complexity, searching using A* can be very time consuming if lots of obstacles exist between source and destination (such as *Figure 6.7* and *Figure 6.10*) or estimated H is inadmissible (such as *Figure 6.4*, *Figure 6.8*, *Figure 6.9*, *Figure 6.10*, *Figure 6.11*). Otherwise, A* algorithm can operate well with reasonable time consumed.

When considering space complexity, since all cells/nodes expanded must be saved in memory, a large amount of memory is required for existence of either many obstacles or inadmissible H estimated, these are proved in *Figure 6.4*, *Figure 6.9*, *Figure 6.10*, and *Figure 6.11*.

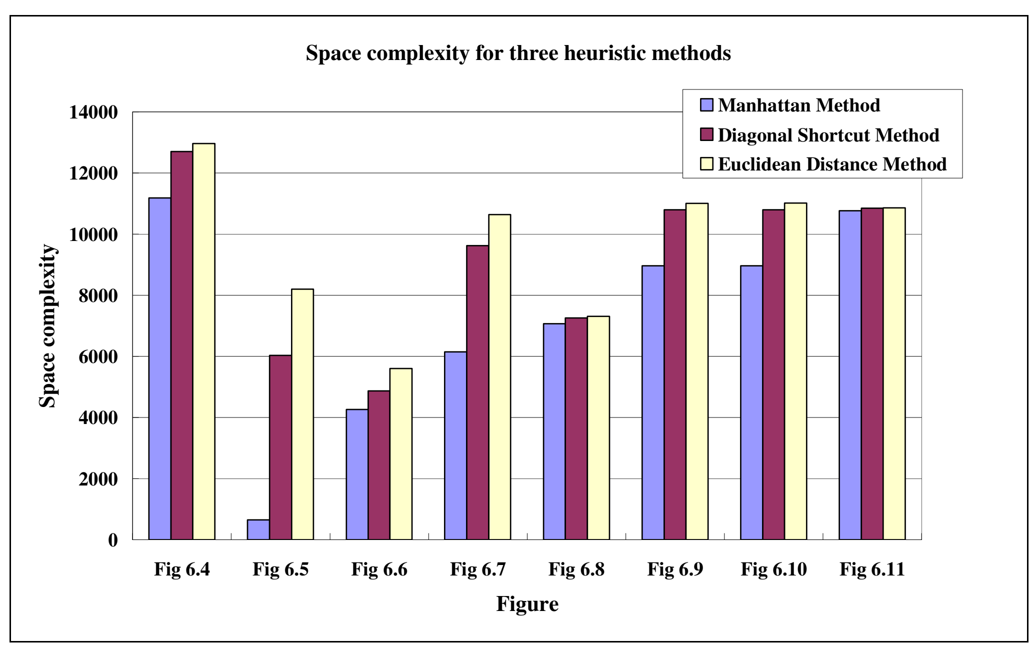
Again refer to *Table 6.1*, among the three heuristic methods, Manhattan Method has predominance which always provides the least number of turning points. Furthermore, compared to others, Manhattan Method always consumes the least time and space complexity (refer *Graph 6.3* and *Graph 6.4*), and provides faster path finding.

However, Manhattan Method may not give shortest or optimal path, for an

instance in *Figure 6.5*, A* using Manhattan Method evaluate path with 312.8 in length, which is longer than that of both Diagonal Shortcut and Euclidean Distance Methods (287.6 in length).



Graph 6.3 Time complexity for Manhattan, Diagonal Shortcut and Euclidean Distance Methods



Graph 6.4 Space complexity for Manhattan, Diagonal Shortcut and Euclidean Distance Methods

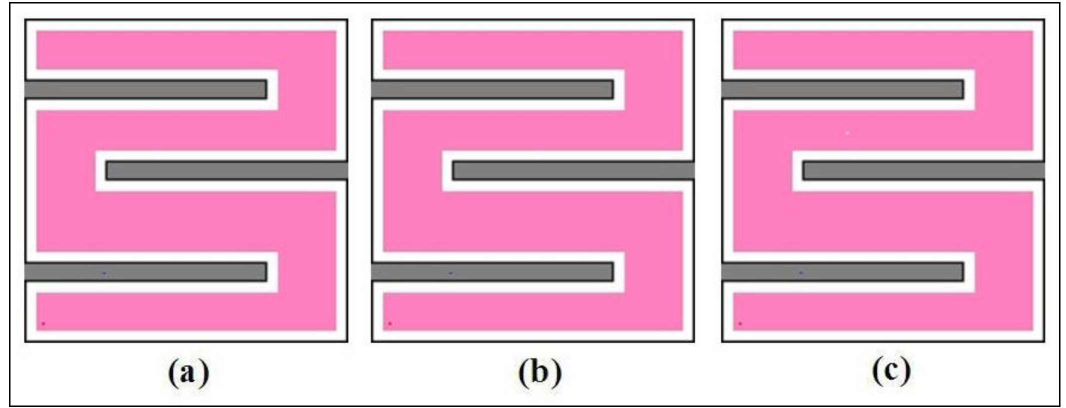


Figure 6.12 A algorithm evaluation from (10,10) to unreachable node (40,35) using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

Figure		Complete if one exists?	Optimal			Time taken	Space (memory)		
			No. of cells	Length	No. of turning points		Space in close list	Space in open list	Total space
6.12	***(a)***	-	-	-	-	75	14764	0	14764
	(b)	-	-	-	-	69	14764	0	14764
	(c)	-	-	-	-	83	14764	0	14764

Table 6.2 A search algorithm evaluation for unreachable goal using (a)Manhattan Method, (b)Diagonal Shortcut Method, (c)Euclidean Distance Method*

The A* algorithm faces problem of exponentially time and space complexity increasing. Seaching for an unreachable goal (*Figure 6.12*) exacerbates the problem, this is due to the searching only stop when the open list is empty (*Table 6.2*). In other words, all possible nodes are expanded. If a huge map is used, then time complexity will increase dramatically or even to infinity due to unlimited space complexity.

6.2.2 Discussion

In general, the A* search algorithm provides complete and optimal solution. It is well for existance of less obstacles between source to destination. The closer estimated H is to the actual remaining distance along the path to the goal, the faster

A* will find the goal. If there is lots of obstables between two points that make H inadmissible, then A* will face dificulty or even failure in finding the path to goal. Besides, the time and space complexity are vary within range [0,∞] depending on terrain and size of terrain.

Manhattan Method has predominance over both Diagonal Shortcut and Euclidean Distance Methods due to its advantages of lower time and space complexity, even the path found may not be optimal.

6.3 Modified DFS Algorithm for Path Exploration

6.3.1 Evaluation of Modified DFS algorithm

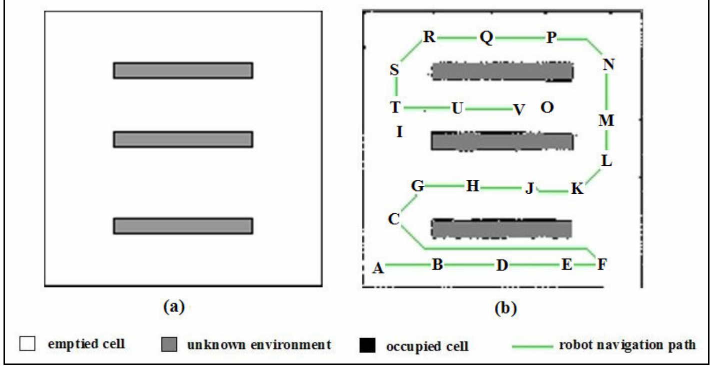


Figure 6.13 Modified DFS algorithm evaluation (a) original map (b) map model

Figure 6.13 illustrates how the Modified DFS algorithm operates for path exploration and navigation. *Figure 6.13(a)* shows an ideal map environment, and *Figure 6.13(b)* shows the exploration process and how the exploration is carried out by the robot from point A to point V.

Note that, by implementing the Modified DFS algorithm, the revisiting problem that is inherent in the original DFS has effectively been solved. In the example illustrated, during path exploration, points I and O are dynamically deleted and thus

will not be visited.

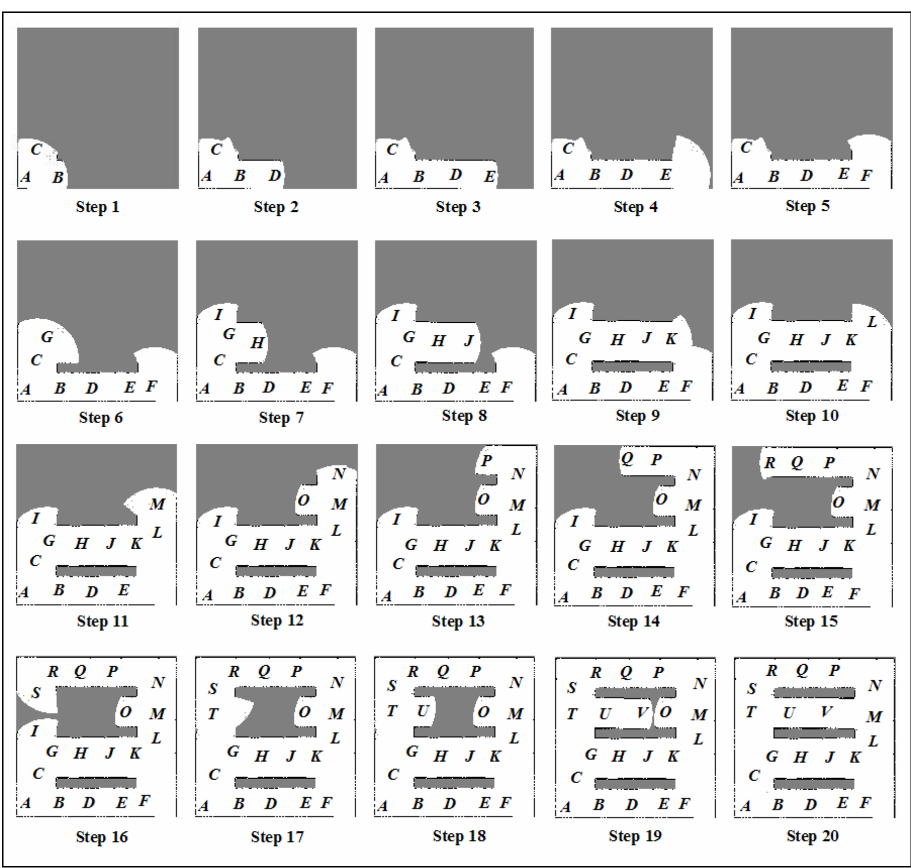


Figure 6.14 Steps of path exploration and navigation using modified DFS algorithm

Step	**Robot located at**	**Unvisited points detected**	**Existing unvisited points**		**Any unvisited points in list?**	**Action/ Navigation**
			Points detected	**Corresponding action**		
1	**A(10,10)**	B(46,14) C(21,43)	-	-	Yes	Moves to B
2	**B**	D(82,14)	-	-	Yes	Moves to D
3	**D**	E(118,14)	-	-	Yes	Moves to E
4	**E**	F(137,14)	-	-	Yes	Moves to F
5	**F**	-	-	-	Yes	Moves to C
6	**C**	G(31,58)	-	-	Yes	Moves to G
7	**G**	H(65,59) I(21,89)	-	-	Yes	Moves to H
8	**H**	J(99,58)	-	-	Yes	Moves to J
9	**J**	K(123,56)	-	-	Yes	Moves to K
10	**K**	L(139,77)	-	-	Yes	Moves to L
11	**L**	M(139,99)	-	-	Yes	Moves to M

12	M	N(139,132) O(102,105)	-	-	Yes	Moves to N
13	N	P(105,145)	-	-	Yes	Moves to P
14	P	Q(68,144)	-	-	Yes	Moves to Q
15	Q	R(36,144)	-	-	Yes	Moves to R
16	R	S(20,124)	-	-	Yes	Moves to S
17	S	T(20,104)	I	Erase I	Yes	Moves to T
18	T	U(57,104)	-	-	Yes	Moves to U
19	U	V(93,103)	-	-	Yes	Moves to V
20	V	-	O	Erase O	No	Stop (complete)

Table 6.3 Steps of path exploration and navigation using modified DFS algorithm

Figure 6.14 and Table 6.3 illustrate the step of path exploration and navigation in detail using the modified DFS algorithm. A list structure is used to store points. Starting at point A(10,10), the robot scans and finds out unvisited points B and C. Points C and B are then pushed into the list, and using LIFO (Last-In First-Out) the robot chooses to navigate to point B.

Once point B has been reached (at step 2), the algorithm marks B as “visited”, it then scans, identifies unvisited point D, pushes D into the list and chooses to navigate to D, and so on and so forth.

At step 5, since the does not scan or “see” any unvisited point, it assumes there is a dead-end. Hence, the robot will reverse to unvisited point C. Once point C has been reached (at step 6), the processes of scanning, identifying unvisited points, navigating to goal determined are repeated until step 20. At step 20, since the list possesses no unvisited point, the map model of environment is inferred completely built, the client program is terminated and the robot is stop.

Note that, at step 17, the existing unvisited point I in the list is detected within the current range of the laser scanner, thus point I is assumed visited and is erased from the list to avoid the revisiting problem. Point O is re-scanned and erased at step 20 as well, again to avoid revisiting.

6.3.2 Discussion

Overall, the Modified DFS algorithm works well. The simulated robot is able to explore an unknown environment without retracing its steps, and thus avoiding the problem of ending up in a permanent loop.

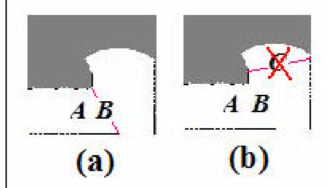


Figure 6.15 Problem of Modified DFS

However, for certain terrain, the robot may not be able to detect the navigable points. For an instance, refer to *Figure 6.15*. *Figure 6.15(a)* shows the simulated robot, which is located at A. It scans unvisited point B. In *Figure 6.15(b)*, point B is reached, the robot scans and finds out a navigable point C. But point C has been visited at a previous step, the robot thus infers C should be ignored. As a result, no new unvisited points will be added to the list, indicating a dead-end is met. The robot will not proceed forward, but instead reverses.

6.4 Quality of Map Model Using Laser Scanner

6.4.1 Evaluation of map model quality

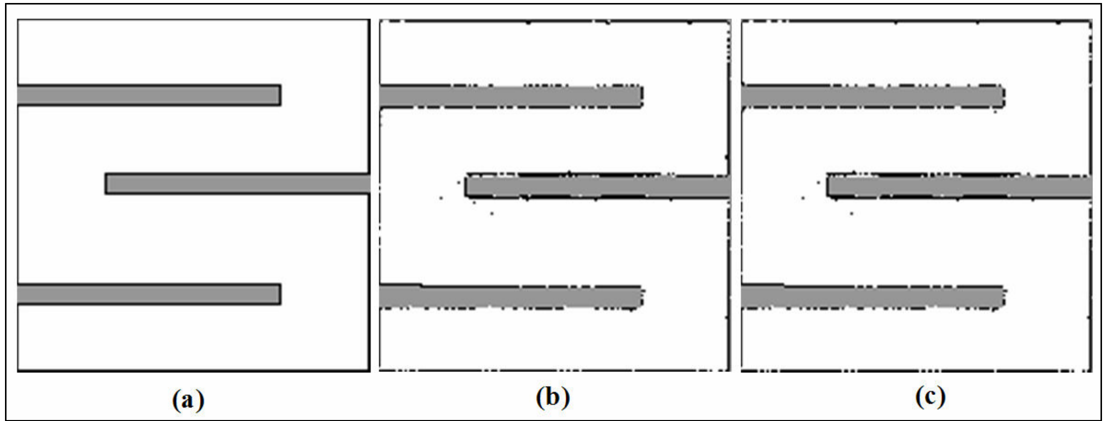


Figure 6.16 Grid-based map building algorithm

(a) original map, (b) map model built (1st test), (c) map model built(2nd test)

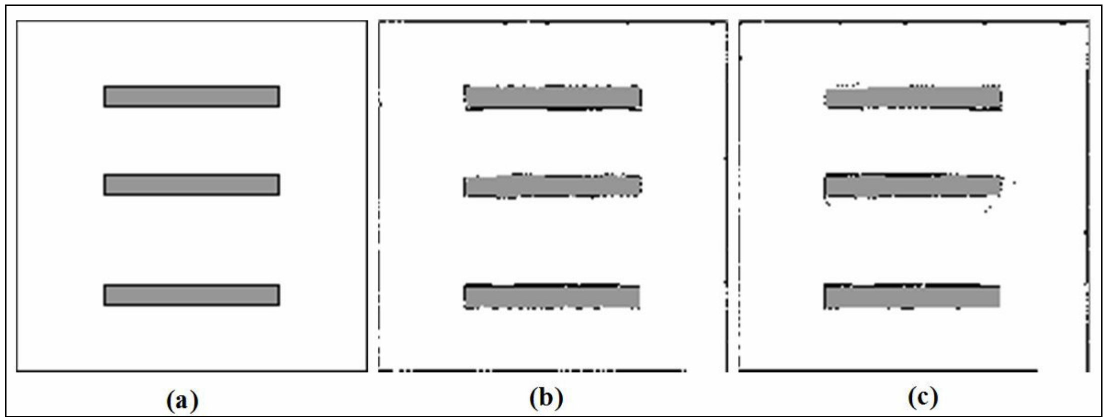


Figure 6.17 Grid-based map building algorithm

(a) original map, (b) map model built (1st test), (c) map model built(2nd test)

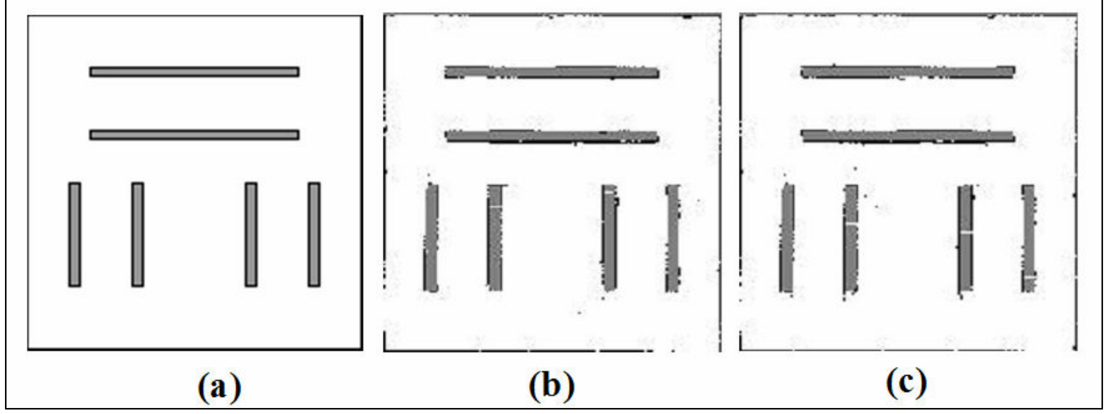


Figure 6.18 Grid-based map building algorithm

(a) original map, (b) map model built (1st test), (c) map model built(2nd test)

Map Model		Cross Correlation	Map Score	Map Score with Occupied Cells
Figure 6.16	(b) 1st testing	68.58%	4.35%	26.47%
	(c) 2nd testing	68.48%	4.36%	26.53%
Figure 6.17	(b) 1st testing	66.75%	3.74%	31.57%
	(c) 2nd testing	69.57%	3.42%	28.81%
Figure 6.18	(b) 1st testing	63.09%	4.50%	40.64%
	(c) 2nd testing	65.09%	4.29%	38.71%

Table 6.4 Quality of map model evaluated using

Cross Correlation, Map Score and Map Score with Occupied Cells Methods

From *Figure 6.16* to *Figure 6.18*, and as shown in *Table 6.4*, the Cross Correlation benchmark (template matching) shows a reasonable high degree of similarity of ideal map. Besides, the Map Score (cell by cell comparison) demonstrates a considerably low error indicating that the maps have been constructed fairly accurately.

However, Map Score benchmark overestimates the empty regions of space, missing an obstacle only increases the score value slightly. An alternative criterion – Map Score with Occupied Cells Method is used to compare just the occupied cells. Thus, missing an obstacle will increase the score value significantly. *Table 6.4* shows all maps model produce over 25% of score value, indicating relatively high error percentages.

6.4.2 Discussion

Since laser scanner is assumed ideal during simulation, all of the score values are dependent on how good is a map building algorithm. The map building algorithm used is simply an occupancy grid mapping, results relatively high error percentages of Map Score with Occupied Cells.

In reality, severe noise and inherent imperfections in the sensor may cause further deterioration in the simulated map quality, and will lead to a lower percentage of Cross Correlation score and higher percentages in the Map Score benchmark and Map Score with Occupied Cells.

In order to alleviate this problem, the occupancy grid mapping algorithm should be coupled with a probabilistic approach, such as Bayesian Theorem.

Chapter 7
Conclusion and Further Work

7.1 Conclusion

The SPA control approach that has been implemented is a reactive model. It eases the construction of online and incremental algorithms for map building. All tests made have investigated the related algorithms implemented, regardless of time delay, noise measurement, dynamic environment.

In an ideal case, starting from a known location, laser-guided exploration enable robot to explore unknown environment without revisiting same place using Modified DFS algorithm.

Priot to robot naviagtion, the optimal and effective path is always found using the A* algorithm with Manhattan Method (Diagonal Shortcut Method and Euclidean Distance Method are discussed and compared as well).

Using laser data acquired, the map environment of various warehouse terrains can be built successfully using occupancy grid mapping technique. The map models are created through PNGwriter Library.

For the sake of safety, obstacle avoidance algorithm is put into operation. Again ignoring time delay, obstacle can be detected instantly once the obstacle is located within specific range of laser scanner, and appropriate action is executed to avoid collision.

However, in reality, time delay and noise measurement must be taken into account during designing and programming. The planning layer of SPA architecture is where map building and path planning take place. These processes are usually computationally complex and time consuming. Apart from that, these processes operate asynchronously exacerbate the problem of time delay.

As a consequence, the plan built from map model may turn out to be inadequate to the environment actually encountered. Besides, the existence of sensor reading noise and pose error influences the inferring of map structure. Thus, alternative algorithms or improvements must be implemented.

7.2 Further Work

7.2.1 Sensor deployment

The accuracy of the occupancy grid mapping and related algorithm is dependent on the noisy or incomplete sensor data acquired. Even if the robot poses are known, it is difficult to infer that the environment is exactly occupied or emptied, due to ambiguities in the sensor data reading.

Hence, a better simulated map quality requires combination of various range sensors *[48, 90, 98]*, such as sonar sensors, laser range finders, and camera. This comes up with integration of sonar, laser and stereo range data by combining their strengths and nullifying their drawbacks, eventually increases the reliability of map

acquisition.

In addition, sensor fusion is required for data integration. Various methods have been subjected, such as Kalman filtering *[115]*, Bayesian reasoning *[112]*, artificial networks *[109]* and fuzzy logic *[113]*.

7.2.2 Self-localisation technique

The robot's accumulated pose error might be unboundedly large, ultimately render a large error-prone map. This is compensated with self-localization techniques, such as dead reckoning *[19, 43]*, Monte-Carlo Localization *[46]*, or landmark based matching algorithm.

7.2.3 Map building algorithm

In this paper, occupancy grid map paradigm applied only consists of three values 0, 0.5, and 1 indicating cell emptied, unknown, and occupied respectively for ease of construction. However, again environment inferring may not be reliable due to ambiguous data acquired. To alleviate this problem, occupancy grid mapping coupled with a probabilistic approach (Bayesian Theorem *[21, 47, 101]*) is suggested.

7.2.4 Simultaneous Localization and Mapping (SLAM)

During mapping, vehicle and map estimates are highly correlated. By using the SLAM algorithm *[20, 53, 54, 84, 104, 108]*, the vehicle can start in an unknown location in an unknown environment and proceed to incrementally build a navigation map of the environment while simultaneously use this map to update its location. Extended Kalman Filter (EKF) SLAM and FastSLAM *[50]* are two examples of SLAM algorithms.

7.2.5 Path finding algorithm

The A* search algorithm for path finding must reconstruct a new path when the state of the environment changes, gives rise to time consuming and inefficiency. D* search algorithm *[63, 64, 65]* is an alternative technique to cope with dynamic environment.

7.2.6 Distributed map building using multirobot system

Future exploration missions will use cooperative robots *[88, 89, 90, 91, 101, 120]* to explore and sample terrain, especially for the mission subjected to time critical, such as search and rescue job.

To proceed to distributed map building using multi-robot system, some problems needed to be considered, such as robot's capability of distinguishing obstacle and different robot, the way the robots exchange data, requirement of contingency and reactive model.

In the Player/Stage simulator, the use of fiducial interface provides access to devices that detect coded fiducials (markers) placed in the environment, enables robot to differentiate obstacle, unique robot and even natural landmarks. Besides, the use of opaque interface allows relay driver to repeat all commands it receives as data packets to all subscribed clients, thus enabling data exchange between robots.

When a robot detects another robot (via fiducial interface) during execution, it will then react to the new information by developing a new plan for data exchange (via relay driver) instead of navigation. Map model will be updated according to the data received, eventually generates fast map building. The new information is also used for avoiding revisiting problem, in other words, the corresponding robot will not re-visit the place another robot has visited.

References

[1] “Firefighter's guardian angel is a palm size robot”,
http://www.digitalyorkshire.org.uk/newsdetails.aspx?id=416,
last accessed 20th September 2007

[2] “Fire and rescue robots "could save lives"”,
http://blogs.guardian.co.uk/technology/archives/2007/01/18/fire_and_rescue_robots_could_save_lives.html,
last accessed 20th September 2007

[3] “Research: Active Project – Viewfinder”,
http://www.shu.ac.uk/mmvl/research/viewfinder/,
last accessed 20th September 2007

[4] “Research: Active Project – Guardians”,
http://www.shu.ac.uk/mmvl/research/guardians/,
last accessed 20th September 2007

[5] “Brooks Subsumption Architecture”,
http://www.cs.ucf.edu/~lboloni/Teaching/EEL6938_2005/slides/Presentation_RyanFitzGibbon_SubsumptionArchitecture.ppt#1,
last accessed 3rd November 2007

[6] “Player Project”,
http://playerstage.sourceforge.net/index.php?src=index,
last accessed 20th January 2008

[7] "Player",

http://playerstage.sourceforge.net/index.php?src=player,

last accessed 20th January 2008

[8] "Stage",

http://playerstage.sourceforge.net/index.php?src=stage

last accessed 20th January 2008

[9] "The Player Robot Device Interface",

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/

last accessed 20th January 2008

[10] "Quick start",

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/start.html,

last accessed 20th January 2008

[11] "sicklms200",

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__driver__sicklms200.html,

last accessed 20th January 2008

[12] "p2os",

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__driver__p2os.html,

last accessed 20th January 2008

[13] "laser (interface specification)",

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interface__laser.html,

last accessed 20th January 2008

[14] "position2d (interface specification)"

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__interface__position2d.html,

last accessed 20th January 2008

[15] "PlayerCc::LaserProxy Class Reference"

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/classPlayerCc_1_1LaserProxy.html,

last accessed 20th January 2008

[16] "PlayerCc::Position2dProxy Class Reference"

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/classPlayerCc_1_1Position2dProxy.html,

last accessed 20th January 2008

[17] "libplayerc++ example"

http://playerstage.sourceforge.net/doc/Player-2.0.0/player/group__cplusplus__example.html,

last accessed 20th January 2008

[18] "Beginner: Learn Linux",

http://linuxreviews.org/beginner/,

last accessed 22nd August 2007

[19] "Dead Reckoning: A Skill All Navigators Should Master"

http://64.233.183.104/search?q=cache:cuodQU509MoJ:www.pilothouseonline.com/IS2V1_00/Lessons/main0001.htm+Dead+Reckoning:+A+Skill+All+Navigators+Should+Master&hl=en&ct=clnk&cd=1,

last accessed 9th July 2007

[20] "Simultaneous Localization and Mapping",

http://www.cs.unc.edu/~lin/COMP790/LEC/13.ppt,

last accessed 29th July 2007

[21] "Robotic Mapping",

http://en.wikipedia.org/wiki/Robotic_mapping,

last accesed 29th July 2007

[22] "GD Graphics Library",

http://www.boutell.com/gd/,

last accessed 5th July 2007

[23] "High Performance JavaScript Vector Graphics Library",

http://www.walterzorn.com/jsgraphics/jsgraphics_e.htm,

last accessed 5th July 2007

[24] "PNGwriter is a C++ Library for creating PNG images",

http://pngwriter.sourceforge.net/,

last accessed 18th November 2007

[25] "PGPLOT Graphics Subroutine Library",

http://www.astro.caltech.edu/~tjp/pgplot/,

last accessed 5th July 2007

[26] "Combining Metric and Topological Navigation of Simulated Robots",

http://www.freeweb.hu/jataka/rics/cscs2004/cscs2004slides.pdf,

last accessed 29th July 2007

[27] "Informed Search Algorithm",

www.cs.brandeis.edu/~cs101a/lectures/Lecture3.ppt,

last accessed 15th December 2007

[28] "Design and Analysis of Algorithms",

http://www.ics.uci.edu/~eppstein/161/960215.html,

last accessed 15th December 2007

[29] "Blind Search",

http://www.cs.ualberta.ca/~lindek/366/slides/BlindSearch.ppt#1,

last accessed 15th December 2007

[30] "Solving Problems by Searching",

http://www.cs.pitt.edu/~litman/courses/cs2710/lectures/ch03RN.ppt#1,

last accessed 15th December 2007

[31] "Artificial Intelligence – From Search to Knowledge",

http://www.ace.tuiasi.ro/~fleon/Curs_AI_Kn/C2_Search.ppt,

last accessed 15th December 2007

[32] "SPA Architectures (Planning, deliberative),

http://web.cecs.pdx.edu/~mperkows/CLASS_479/May6/030.Deliberative-SPA.ppt,

last accessed 28th June 2007

[33] "Beginners Guide to Pathfinding Algorithms",

http://ai-depot.com/Tutorial/PathFinding.html,

last accessed 15th December 2007

[34] "Dijkstra's Algorithm",
http://www.cs.usask.ca/resources/tutorials/csconcepts/1999_8/tutorial/advanced/dijkstra/dijkstra.html,
last accessed 15th December 2007

[35] "Introduction to Robotics – Subsumption Architecture",
http://math.haifa.ac.il/robotics/Presentations/pdf/Ch11_Subsumption.PDF,
last accessed 28th June 2007

[36] "SSS: A Hybrid Architecture Applied to Robot Navigation",
http://www.lecs.cs.ucla.edu/~girod/official/talks/584-sss.ppt#1,
last accessed 28th June 2007

[37] "Shape Representation and Description: Contour-based Shape Representation and Description",
http://www.icaen.uiowa.edu/~dip/LECTURE/Shape2.html#chaincodes,
last accessed 17th October 2007

[38] "Representation and Description",
http://www.cvmt.dk/~hn/2005/BA1/Slides/8.mm/engelsk.ppt#4,
last accessed 17th October 2007

[39] "Cross Correlation",
http://local.wasp.uwa.edu.au/~pbourke/other/correlate/,
last accessed 29th December 2007

[40] "Near Optimal Hierarchical Pathfinding (HPA*)",
http://www.cs.ualberta.ca/~bulitko/F06/presentations/2006-09-29-ML.pdf,
last accessed 17th December 2007

[41] "Introduction to Planning as Search",
http://iew3.technion.ac.il/~dcarmel/planning05/lectures/basic_search.pdf
last accessed 17th December 2007

[42] Artificial Intelligence: Uninformed search methods,
http://www.cs.mcgill.ca/~jpineau/comp424/Lectures/02Search1.pdf
last accessed 15th December 2007

[43] "Dead Reckoning",
http://www.irbs.com/bowditch/pdf/chapt07.pdf,
last accessed 19th July 2007

[44] Shane O'Sullivan (2003) "An Empirical Evaluation Of Map Building Methodologies in Mobile Robotics Using The Feature Prediction Sonar Noise Filter And Metric Grid Map Benchmarking Suite", Master of Science, University of Limerick

[45] Mir Immad ud din (2005-06) "Incremental Perception in Swarm Robotics", Master of Science, Sheffied Hallam University

[46] F. Dellaert, D. Fox, W. Burgard, and S. Thrun (1999) "Monte Carlo Localization for Mobile Robots", *IEEE International Conference on Robotics and Automation (ICRA99)*

[47] Moravec and Elfes (1985) "High Resolution Maps From Wide Angle Sonar"

[48] Matthies and Elfes (1988) "Integration of Sonar and Stereo Range Data Using a Grid Based Representation"

[49] Konolige (1997) "Improved Occupancy Grids for Map Building"

[50] D. Hanel, W. Burgard, D. Fox, and S. Thrun (2003) “An efficient FastSLAM algorithm for generating maps of large scale cyclic environments from raw laser range measurements”

[51] H. Choset and K. Nagatani (2001) “Topological simultaneous localization and mapping (SLAM): toward exact localization without explicit localization”, *IEEE Transactions on Robotics and Automation*, 17(2)

[52] Erann Gat (1998) “On Three-Layer Architectures”, Jet Propulsion Laboratory, California Institute of Technology

[53] Andrew Howard, “Multi-robot Simultaneous Localization and Mapping using Particle Filters”, Jet Propulsion Laboratory, California Institute of Technology

[54] Chieh-Chih Wang, and Charles Thorpe, “A Hierarchical Object Based Representation for Simultaneous Localization and Mapping”

[55] Schiele, B. and Crowley, J. (1994) “A comparison of position estimation techniques using occupancy grids”, *Robotics and Autonomous Systems*

[56] J. Borenstein, Y. Koren (1989) “Real-time Obstacle Avoidance for Fast Mobile Robots”, *IEEE Transactions on Systems, Man, and Cybernetics*

[57] R. Gartshore, A. Aguado, C. Galambos (2002) “Incremental Map Building Using an Occupancy Grid for an Autonomous Monocular Robot”

[58] Seydou SOUMARE, Akihisa OHYA and Shin’ichi YUTA, “Real-Time Obstacle Avoidance by an Autonomous Mobile Robot using an Active Vision Sensor and a Vertically Emitted Laser Slit”, Intelligent Robot Laboratory, University of Tsukuba

[59] S. Albers and M. Henzinger, “Exploring unknown environments”

[60] Carlos Hernández1 and Pedro Meseguer, “Improving Real-Time Heuristic Search on Initially Unknown Maps”

[61] Patrick Lester (2005) “A* Pathfinding for Beginners”

[62] Patrick Lester (2004) “Heuristics and A* Pathfinding”

[63] Anthony Stentz, “Optimal and Efficient Path Planning for Partially-Known Environments”

[64] Dave Ferguson and Anthony Stentz, “The Delayed D* Algorithm for efficient Path Replanning”, Carnegie Mellon University

[65] Anthony Stentz, “The Focussed D* Algorithm for Real-Time Replanning”, Robotics Institute, Carnegie Mellon University

[66] Anand Veeraswamy (2004) “Implementation of Path Finding Techniques in Homeland Security Robots”

[67] Kelly Manley, “Pathfinding: From A* to LPA”, University of Minnesota

[68] Csaba Szepesv´ari, “Shortest Path Discovery Problems: A Framework, Algorithms and Experimental Results”, Computer and Automation Research Institute of the Hungarian Academy of Sciences

[69] Dieter Fox, Jonathan Ko, Kurt Konolige, and Benjamin Stewart, “A Hierarchical Bayesian Approach to the Revisiting Problem in Mobile Robot Map Building”

[70] Sven Koenig, "Fast Replanning for Navigation in Unknown Terrain"

[71] J.J. Collins, Malachy Eaton, Mark Mansfield, and David Haskett., "Developing a Benchmarking Framework for Map Building Paradigms", University of Limerick

[72] Thrun, Sebastian (2002) "Robotic Mapping: A Survey", Carnegie Mellon University

[73] Shane O'Sullivan, J.J. Collins, Mark Mansfield, David Haskett, and Malachy Eaton, "Linear Feature Prediction for Confidence Estimation of Sonar Readings in Map Building"

[74] Rodney A. Brooks (1986) "A Robust Layered Control System for a Mobile Robot.", *IEEE Transactions on Robotics and Automation*, 2(1), pages 14-23

[75] Jonathan Simpson, Christian L. Jacobsen and Matthew C. Jadud (2006) "Mobile Robot Control The Subsumption Architecture and occam-pi"

[76] Martin C. Martin, "Visual Obstacle Avoidance Using Genetic Programming: First Results", Carnegie Mellon University

[77] Jonathan H. Connell, "Creature Design with the Subsumption Architecture", MIT Artificial Intelligence Lab, Cambridge

[78] Martin C. Martin, Hans P. Moravec (1996) "Robot Evidence Grids", CMU-RI-TR-96-06, The Robotics Institute, Carnegie Mellon University

[79] Brunelli, R. and Poggio, T. (1993) "Face recognition: Features versus templates", *IEEE Transactions on Pattern Analysis and Machine Intelligence*,

15(10):1042–1052

[80]Baron, R. (1981) "Mechanisms of human facial recognition", *International Journal of Man-Machine Studies*, 15:137–178

[81]M.J.T. Reinders, "Eye Tracking by Template Matching using an Automatic Codebook Generation Scheme", Dept. of Electrical Engineering, Delft University of Technology

[82]Hemant M. Joshi, Joshua A. McAdams, "Search Algorithms in Intelligent Agents"

[83]M Kamali, "Artificial Intellegence and Search Algorithms"

[84]Eduardo Nebot (2006) "Simultaneous Localization and Mapping", Australian Centre for Field Robotics, University of Sydney NSW, Australia

[85]Denis Wolf and Gaurav S. Sukhatme (2004) "Online Simultaneous Localization and Mapping in Dynamic Environments", Robotic Embedded Systems Laboratory, University of Southern California

[86]Michael Milford, Gordon Wyeth, David Prasser, "Simultaneous Localisation and Mapping from Natural Landmarks using RatSLAM", University of Queensland

[87]C. M. Smith and J. J. Leonard (1997) "A multiple-hypothesis approach to concurrent mapping and localization for autonomous underwater vehicles", *Proc. Int. Conf. Field and Service Robotics*

[88]Raj Madhavan, Kingsley Fregene, Lynne E. Parker (2004) "Distributed Cooperative Outdoor Multirobot Localization and Mapping"

[89] Vivek A. Sujan, Steven Dubowsky, Terry Huntsberger, Hrand Aghazarian, Yang Cheng, Paul Schenker “A Multi Agent Distributed Sensing Architecture with Application to Planetary Cliff Exploration”

[90] D.Rus, A. Kabir, K. Kotay, M. Soutter (1996) “Guiding Distributed Manipulation with Mobile Sensors”, Department of Computer Science, Dartmouth College

[91] Scott A. Deloach, Eric T. Matson, Yong Hua Li, “Exploiting Agent Oriented Software Engineering in Cooperative Robotics Search and Rescue”, Department of Computing and Information Sciences, Kansas State University

[92] Christian Icking, Thomas Kamphans, Rolf Klein, Elmar Langetepe, “Exploring an Unknown Cellular Environment”

[93] X. Deng, T. Kameda, C. Papadimitriou (1991) “How to Learn an Unknown Environment (Extended Abstract)”

[94] F. Hoffmann, C. Icking, R. Klein, K. Kriegel, “The Polygon Exploration Problem: A New Strategy and a New Analysis Technique”

[95] A. Elfes (1987) “Sonar-based real-world mapping and navigation”, *IEEE Journal of Robotics and Automation*, RA-3(3):249–265

[96] J. Borenstein and Y. Koren. (1991) “The vector field histogram – fast obstacle avoidance for mobile robots”, *IEEE Journal of Robotics and Automation*, 7(3):278–288

[97] J. Buhmann, W. Burgard, A.B. Cremers, D. Fox, T. Hofmann, F. Schneider, J.

Strikos, and S. Thrun (1995) “The mobile robot Rhino”, *AI Magazine*, 16(1)

[98] Mark A. Lanthier, Doron Nussbaum and An Sheng, “Improving Vision-Based Maps By Using Sonar and Infrared Data”, Carleton University

[99] D. Guzzoni, A. Cheyer, L. Julia, and K. Konolige (1997) “Many robots make short work”, *AI Magazine*, 18(1):55–64

[100] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. H¨ahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz (2000) “Probabilistic algorithms and the interactive museum tour-guide robot Minerva”, *International Journal of Robotics Research*, 19(11):972–999

[101] Mark M. Chang and Gordon F. Wyeth, “Achieving Cooperation in a Distributed Multi-Robot Team”, University of Queensland,

[102] B. Kuipers and Y.-T. Byun. (1991) “A robot exploration and mapping strategy based on a semantic hierarchy of spatial representations”, *Journal of Robotics and Autonomous Systems*, 8:47–63

[103] H. Choset and J.W. Burdick. (1996) “Sensor Based Planning: The Hierarhical Generalized Voronoi Graph”, *Proc. Workshop on Algorithmic Foundations of Robotics*, Toulouse

[104] J. Guivant and E. Nebot, (2001) “Optimization of the simultaneous localization and map building algorithm for real time implementation”, *IEEE Transaction of Robotic and Automation*

[105] F. Lu and E. Milios (1997) “Globally consistent range scan alignment for environment mapping”, *Autonomous Robots,* 4:333–349

[106] H Shatkay and L. Kaelbling (1997) "Learning topological maps with weak local odometric information", *Proceedings of IJCAI-97. IJCAI, Inc*

[107] M. C. Torrance. (1994) "Natural communication with robots", Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge

[108] P. Newman. (2000) "On the Structure and Solution of the Simultaneous Localisation and Map Building Problem", PhD thesis, Australian Centre for Field Robotics, University of Sydney

[109] Humberto Mart´ınez Barber´a, Antonio G´omez, Skarmeta, Miguel Zamora Izquierdo and Juan Bot´ıa Blaya (2000) "Neural Networks for Sonar and Infrared Sensors Fusion", *3rd Intl. Fusion Conference*, Paris

[110] L. Iocchi, K. Konolige, and M. Bajracharya (2000) "Visually realistic mapping of a planar environment with stereo", *Proceedings of the 2000 International Symposium on Experimental Robotics*

[111] Y. Liu, R. Emery, D. Chakrabarti, W. Burgard, and S. Thrun. (2001) "Using EM to learn 3D models with mobile robots", *Proceedings of the International Conference on Machine Learning (ICML)*

[112] P. Sykacek and I. Rezek (2000) "Markov Chain Monte Carlo Methods for Bayesian Sensor Fusion", *Kluwer Acadamic Publishers*

[113] Richard A. Wasniowski (2005) "Multisensor Agent Based Intrusion Detection", *Proceedings of World Academy of Science*

[114] J.J. Leonard and H.F. Durrant-Whyte (1991) "Simultaneous map building and localization for an autonomous mobile robot", *IEEE/RSJ Int. Workshop on Intelligent Robots and Systems*, vol.3, pp. 1442-47, 1991

[115] G. Welch and G. Bishop (2003) "An Introduction to Kalman Filter", University of North Carolina, TR, 95-041

[116] H. Zhao and R. Shibasaki. (2001) "Reconstructing Urban 3D Model using Vehicle-borne Laser Range Scanners", *Proc. of the Third Int. Conf. on 3-D Digital Imaging and Modeling*, pp. 349-56

[117] D. Toal, C. Flanagan, C. Jones, B. Strunz, "Subsumption Architecture for the Control of Robots"

[118] Frank Dellaert (2002) "The Expectation Maximization Algorithm"

[119] Jonathan Reynolds (2005) "An Exploration of Mapping Algorithms for Autonomous Robotic Mapping", Colorado State University

[120] K. Konolige, D. Fox, B. Limketkai, J. Ko, and B. Stewart (2003) "Map merging for distributed robot navigation", *Proc. IEEE Intl. Conf. on Intelligent Robots and Systems*, Las Vegas

Appendices

Appendix A: *.world* configuration file

```
# Description: 1 pioneer robot with laser

include "pioneer.inc"
include "map.inc"
include "sick.inc"
size [16 16]

# set the resolution of the underlying ray trace model in meters
resolution 0.02

# update the screen every 10ms (we need fast update for the stest demo)
gui_interval 20

# configure the GUI window
window
(
        size [ 640.000 640.000 ]
        center [0.000 0.000]
        scale 0.028
)

# load an environment bitmap
map
(
   bitmap "bitmaps/example1.png"
   size [16 16]
   name "example1"
)

# create a robot
pioneer2dx
(
   name "robot1"
   color "red"
   pose [-7 -7 45]
   sick_laser( samples 361 laser_sample_skip 4 )
)
```

Appendix B: *.cfg* configuration file

```
# Description: Player sample configuration file for controlling Stage devices

# load the Stage plugin simulation driver
driver
(
    name "stage"
    provides ["simulation:0" ]
    plugin "libstageplugin"

    # load the named file into the simulator
    worldfile "example1.world"
)

driver
(
    name "stage"
    provides ["map:0"]
    model "example1"
)

# Create a Stage driver and attach position2d and laser interfaces to the model "robot1"
driver
(
    name "stage"
    provides ["position2d:0" "laser:0" ]
    model "robot1"
)
```

Appendix C: *.cc* executable file

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// Description: A C++ client program for a laser-guided robot,
///                 the mobile robot possesses capability of building an unknown warehouse map with autonomous
///                 navigation system, the starting location of the robot is known
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////

#include <iostream>
#include <math.h>
#include <pngwriter.h>
#include <fstream>
//#include <conio.h>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>
#include <string>
using namespace std;
#include <libplayerc++/playerc++.h>
using namespace PlayerCc;
#include "args.h"

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// functions declaration
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void scan(LaserProxy* LP , Position2dProxy& P2P , PlayerClient & ROBOT);
void mapbuilding(LaserProxy* LP , Position2dProxy& P2P , PlayerClient & ROBOT, pngwriter* PW);
void eraseOverlap (LaserProxy* LP , Position2dProxy& P2P);
bool finishMapping ( );
bool pathPlanning ( int DepX , int DepY, int DestX , int DestY );
int findG ( int g , int arrow);
int findH ( int depX , int depY, int destX , int destY );
int findF ( int g , int h);
int lowestFloc( vector < vector <int> > *vec);
bool extendSize( vector< vector <int> > *vec, int size );
int checkLocAvailable(vector< vector <int> > *vec , int size);
void findDepXY ( int arrow , int parentX , int parentY , int &DepX , int &DepY );
bool notInCL ( vector< vector <int> > *vec , int x , int y);
```

```
int inOLloc ( vector< vector <int> > *vec , int x , int y);
bool isObstacle ( int x , int y , int arrow );
void insertRow( vector< vector <int> > *vec , int loc , int x, int y, int arrow , int g , int h , int f);
void clearRow(    vector< vector <int> > *vec , int loc );
void pathDecision ( vector< vector <int> > *vec , bool found);
void moveRobot(LaserProxy* LP , Position2dProxy& P2P , PlayerClient & ROBOT);

template<typename T>
inline T approximate(T a);

template<typename T>
inline T sq ( T a);

template < typename T >
void printList( const list< T > &listRef );

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// global variables declaration
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
#define PI 3.1415927
struct mapData
{
        int gridNum;
        int gridX;
        int gridY;
        double value;
};

vector< vector<double> > matrixVal(161, vector<double>(161,0.5));
vector< vector<int> > navigablePath ( 1, vector<int>(2,0));
list <int> XX;
list <int> YY;
list <int> sampleNo;
list <int> planX;
list <int> planY;
list <int> visited;    // 0 = not visited, >0 = visited
list <int>::iterator listIteratorX;
list <int>::iterator listIteratorY;
list <int>::iterator listIteratorV;
double opacity = 1.0;    //0.0 to 1.0
double minfrontdistance = 0.5;
double speed = 0.200;
```

```
double offsetX =10, offsetY=10 ;
double offsetAngle = 45.0/180.0*PI;
double maxLaser = 4.0;
double turn180;
int mapDestX, mapDestY, mapDepX, mapDepY, oriDepX, oriDepY;
bool forward = true;
int state=0;
int gocount = 0;
double newturnrate=0.0f, newspeed=0.0f;

///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// main program
///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int main(int argc, char** argv)
{
	parse_args(argc,argv);
	LaserProxy *lp = NULL;
	FiducialProxy *fp = NULL;
	pngwriter *image = NULL;

	try
	{
		PlayerClient robot(gHostname, gPort);
		Position2dProxy pp(&robot, gIndex);
		lp = new LaserProxy (&robot, gIndex);
		fp = new FiducialProxy (&robot, gIndex);
		image = new pngwriter(160,160,0.5,"out.png");    ///grey as background colour
		image->close();

		int input;
		int randint;
		int randcount = 0;
		int avoidcount = 0;
		bool obs = false;
		double minleft, minright;
		int unused = 0;
		bool startPathPlanning= true;
		bool incompletePath = false;
		int incompCount = 0;

		planX.push_back( (int) offsetX *10);
		planY.push_back( (int) offsetY *10);
```

```
	visited.push_back(1);


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
	/// SPA control algorithm

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
	do
	{
		pp.SetMotorEnable (true);
		robot.Read();


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
		/// obstacle detection

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
		obs = false;
		for (uint i = 143; i < 219; i++)
		{
			if((*lp)[i] < minfrontdistance)
		obs = true;
		}


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
		/// an obstacle detected, obstacle avoidance is implemented
		////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
		//////
		if(obs || avoidcount || pp.GetStall ())
		{
			if(obs )
			{
				randcount = 0;
				gocount = 0;
				minright = (*lp)[131];
				minleft = (*lp)[231];

				for ( uint j = 132 ; j < 181 ; j++)
				{
					if ( minright > (*lp)[j] )
						minright = (*lp)[j];
				}
```

```
                        for (uint k = 230 ; k >179 ; k--)
                        {
                                if (minleft > (*lp)[k])
                                        minleft = (*lp)[k];
                        }

                        if (minleft <= minright )
                                newturnrate = -0.5;
                        else if ( minleft > minright)
                                newturnrate = 0.5;
                        newspeed = 0.0;
                        avoidcount = 50;
                }

                else if (avoidcount)
                {
                        newturnrate = 0.0;
                        newspeed = 0.2;
                        avoidcount--;
                }
                pp.SetSpeed(newspeed, newturnrate);
        }


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        /// no obstacle detected, path exploration or navigation is implemented
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        //////
        else
        {
                avoidcount = 0;


////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                /// scans unvisited navigable points

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
                if (state == 0)
                {
                        eraseOverlap (lp,pp);
                        scan (lp,pp,robot);
                        eraseOverlap (lp,pp);
                        mapbuilding(lp,pp,robot,image);
```

```
                if (visited.back() == 0)
                {
                        state =1;
                        forward = true;
                }

                else
                {
                        state =3;
                        forward = false;
                }

                oriDepX = mapDepX = mapDestX;
                oriDepY = mapDepY = mapDestY;
                pp.SetSpeed(0.0,0.0);
        }

        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///////
        /// plan to go which point
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        ///////
        else if (state == 1)
        {
                if ( !planX.empty() && !planY.empty() && !finishMapping() )
                {
                        mapDestX = planX.back();
                        mapDestY = planY.back();

                        if ( mapDestX < 6 || mapDestX >154 || mapDestY < 6 || mapDestY >154 )
                        {
                                if (    mapDestX < 6 )
                                        mapDestX = 6;
                                else if (    mapDestX > 154 )
                                        mapDestX = 154;

                                if (    mapDestY < 6 )
                                        mapDestX = 6;
                                else if (    mapDestX > 154 )
                                        mapDestX = 154;
                        }

                        if ( (matrixVal [mapDestX][mapDestY] <0.5) )
                                state = 2;
```

```
                else
                        state = 3;

                pp.SetSpeed(0.0,0.0);
        }
        else
                state = 4;    /// map building completed
}
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///////
/// path finding, robot navigates to the goal
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///////
else if (state ==2)
{
        /// path finding
        if ( startPathPlanning == true)
        {
                if ( (mapDepX != mapDestX) || (mapDepY != mapDestY) )
                {
                        /// shortest and safest path is found
                        if ( pathPlanning ( mapDepX , mapDepY , mapDestX , mapDestY) )
                                startPathPlanning = false;

                        /// fail to find path
                        else
                                state = 3;
                }
                navigablePath.resize(navigablePath.size()-1);
                mapDestX = navigablePath[navigablePath.size()-1][0];
                mapDestY = navigablePath[navigablePath.size()-1][1];
        }

        /// navigate to the goal
        else
        {
                moveRobot(lp, pp,robot);

                if ( (navigablePath.size()==0) )
                {
                        startPathPlanning = true;
                        incompCount = 0;
                        if (visited.back() == 0 )
                        {
```

```
                        visited.pop_back();
                        visited.push_back(1);
                        state = 0;
                    }

                    else
                    {
                        state = 3;
                        mapDepX = mapDestX;
                        mapDepY = mapDestY;
                    }
                }
            }
        }

        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
    ///////
        /// reverse, remove the point located at the end of the list

        ////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
        ///////
        else if (state == 3)
        {
            planX.pop_back();
            planY.pop_back();
            visited.pop_back();
            state = 1;
            pp.SetSpeed(0.0,0.0);

            if (visited.back() == 0)
                forward = true;
            else
                forward = false;
        }
    }
  } while (state !=4);
}

catch (PlayerCc::PlayerError e)
{
std::cerr << e << std::endl;
return -1;
}
}
```

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// scans 360o and finds out unvisited navigable points
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void scan(LaserProxy* LP , Position2dProxy& P2P, PlayerClient & ROBOT)
{
	int X1,X2,Y1,Y2,S1,S2,midX,midY;
	int obsX , obsY ;
	int preX, preY;
	int proX, proY;
	double x = P2P.GetXPos()*cos(offsetAngle)+ P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
	double y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);
	double angle = P2P.GetYaw() + offsetAngle;
	double angle2;
	bool occupied = false;
	bool findMid = false;
	bool unoccupiedAngle0 = false;
	bool unoccupiedAngle360 = false;

	XX.clear();
	YY.clear();
	sampleNo.clear();

	for (int b = 0; b < 2 ; b++)
	{
		for (uint a = 0; a < 361; a++) //0.5 degree resolution
		{
			angle2 = angle + ((a/2.0-90.0)*PI/180.0);
			if (a < 360)
			{
				proX = limit ( (int) approximate (((*LP)[a+1]*cos(angle2) + x + offsetX)*10.0) , 0 , 160);
				proY = limit ( (int) approximate (((*LP)[a+1]*sin(angle2) + y + offsetY)*10.0) , 0 , 160);

			}

			if (a==0)
			{
				obsX = limit ( (int) approximate (((*LP)[a]*cos(angle2) + x + offsetX)*10.0) , 0 , 160);
				obsY = limit ( (int) approximate (((*LP)[a]*sin(angle2) + y + offsetY)*10.0) , 0 , 160);

				if ((*LP)[a] == maxLaser && b == 0 )
				{
```

```
            occupied = false;
            unoccupiedAngle0 = true;
        }

        else if ((*LP)[a] == maxLaser && b == 1 && unoccupiedAngle360)
            occupied = false;

        else if ((*LP)[a] == maxLaser && b == 1 && !unoccupiedAngle360)
        {
            occupied = false;
            XX.push_front(obsX);
            YY.push_front(obsY);
            sampleNo.push_front(a);
        }

        else
            occupied = true;
    }

    else if (a== 360)
    {
        if ((*LP)[a] == maxLaser && b == 0 && !occupied )
            unoccupiedAngle360 = true;

        else if ((*LP)[a] == maxLaser && b == 0 && occupied )
        {
            unoccupiedAngle360 = true;
            occupied = false;
            XX.push_front(obsX);
            YY.push_front(obsY);
            sampleNo.push_front(a);
        }

        else if ((*LP)[a] == maxLaser && b == 1 && unoccupiedAngle0
        && !occupied)
        {
            XX.push_back(XX.front());
            YY.push_back(YY.front());
            sampleNo.push_back(sampleNo.front());
            XX.pop_front();
            YY.pop_front();
            sampleNo.pop_front();
        }
```

```
        else if ((*LP)[a] == maxLaser && b == 1 && !unoccupiedAngle0
        && !occupied)
        {
            XX.push_front(obsX);
            YY.push_front(obsY);
            sampleNo.push_front(a);
        }

        else if ((*LP)[a] == maxLaser && b == 1 && unoccupiedAngle0 &&
        occupied)
        {
            XX.push_back(obsX);
            YY.push_back(obsY);
            sampleNo.push_back(a);
        }

        else
            occupied = true;
    }

    else if (!occupied)
    {
        if ( sqrt ( sq(obsX-proX) + sq(obsY-proY) ) < 2.0 && (*LP)[a]<maxLaser )
        {
            occupied = true;
            XX.push_front(obsX);
            YY.push_front(obsY);
            sampleNo.push_front(a);
        }
    }

    else
    {
        if (    (*LP)[a] == maxLaser      )
        {
            occupied = false;
            XX.push_front(preX);
            YY.push_front(preY);
            sampleNo.push_front(a);
        }
    }

    preX = obsX;
    preY = obsY;
```

```
            obsX = proX;
            obsY = proY;
        }

        if (b == 0)
        {
            /// turn 180 degree
            if (P2P.GetYaw() ==0)
            {
                turn180 = PI;

                while ( (abs (P2P.GetYaw()- turn180)>0.005) && (abs (P2P.GetYaw() + turn180)
>0.005) )
                {
                    P2P.GoTo (x, y, turn180);
                    P2P.SetMotorEnable (true);
                    ROBOT.Read();
                }
            }

            else
            {
                if ( P2P.GetYaw() > 0 )
                    turn180 = P2P.GetYaw() - PI;

                else
                    turn180 = P2P.GetYaw() + PI;

                while ( abs(P2P.GetYaw() - turn180) >0.005)
                {
                    P2P.GoTo ( (x*cos(offsetAngle)) + (y*sin(offsetAngle)) ,    (y*cos(offsetAngle)) –
                    (x*sin(offsetAngle)), turn180);

                    P2P.SetMotorEnable (true);
                    ROBOT.Read();
                }
            }

            P2P.SetSpeed(0.0, 0.0);
            P2P.SetMotorEnable (true);
            ROBOT.Read();

            x = P2P.GetXPos()*cos(offsetAngle) + P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
            y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);
```

```
            angle = P2P.GetYaw() + offsetAngle;
        }
    }

    if (    !XX.empty() && !YY.empty() )
    {
        if ( XX.size() % 2 == 1)
        {
            XX.pop_front();
            YY.pop_front();
            sampleNo.pop_front();
            cout << "size recorrect" <<endl;
        }

        while ( !XX.empty() && !YY.empty() )
        {
            if ( !findMid )
            {
                X1 = XX.front();
                XX.pop_front();
                Y1 = YY.front();
                YY.pop_front();
                S1 = sampleNo.front();
                sampleNo.pop_front();
                findMid = true;
            }

            else
            {
                X2 = XX.front();
                XX.pop_front();
                Y2 = YY.front();
                YY.pop_front();
                S2 = sampleNo.front();
                sampleNo.pop_front();
                findMid = false;

                if ( sqrt (sq(X1-X2)+sq(Y1-Y2)) > 10.0 && abs (S2-S1) > 2)
                {
                    midX = (int)(X1+X2)/2;
                    midY = (int)(Y1+Y2)/2;

                    if ( matrixVal [midX][midY] == 0.5)
                    {
```

```
                        planX.push_back(midX);
                        planY.push_back(midY);
                        visited.push_back(0);
                    }
                }
            }
        }
    }
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// map building, map model created in .png file usinf PNGwriter
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void mapbuilding (LaserProxy* LP , Position2dProxy& P2P , PlayerClient & ROBOT, pngwriter* PW)
{
    int obsX , obsY ;
    int preX,preY;
    int proX, proY;
    int obsXX , obsYY;
    double val;
    int count = LP->GetCount(); //count = 361
    double x = P2P.GetXPos()*cos(offsetAngle) + P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
    double y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);
    double angle = P2P.GetYaw() + offsetAngle;
    double angle2;
    bool occupied = false;

    obsX = (int) approximate((x+offsetX)*10.0);
    obsY = (int) approximate((y+offsetY)*10.0);
    val = 0.0;
    PW->plot_blend ( obsX, obsY,opacity, (1.0-val)*1.0,(1.0-val)*1.0,(1.0-val)*1.0);
    matrixVal [obsX][obsY] = val;

    for ( int b = 0 ; b<2 ; b++)
    {
        for (uint a = 0; a < count; a+=1)
        {
            for (double i = 0.1; i<(*LP)[a];i+=0.1)
            {
                obsXX = limit ( (int) approximate ((i*cos(angle2) + x + offsetX)*10.0) , 1 , 160);
                obsYY = limit ( (int) approximate ((i*sin(angle2) + y + offsetY)*10.0) , 1 , 160);
                val = 0.0;
```

```
        PW->plot_blend (obsXX ,obsYY ,opacity , (1.0-val)*1.0 , (1.0-val)*1.0 , (1.0-val)*1.0 );
        matrixVal [obsXX][obsYY] = val;
}
angle2 = angle + ((a/2.0-90.0)*PI/180.0);
if (a <( count-1 ))
{
        proX = limit ( (int) approximate (((*LP)[a+1]*cos(angle2) + x + offsetX)*10.0) , 1 , 160);
        proY = limit ( (int) approximate (((*LP)[a+1]*sin(angle2) + y + offsetY)*10.0) , 1 , 160);

}

if (a==0)
{
        obsX = limit ( (int) approximate (((*LP)[a]*cos(angle2) + x + offsetX)*10.0) , 1 , 160);
        obsY = limit ( (int) approximate (((*LP)[a]*sin(angle2) + y + offsetY)*10.0) , 1 , 160);

        if ((*LP)[a] == maxLaser)
        {
                val = 0.0;
                occupied = false;
        }
        else
        {
                val = 1.0;
                occupied = true;
        }
}

if (a==(count-1))
{
        if ((*LP)[a] == maxLaser)
                val = 0.0;
        else
                val = 1.0;
}

else if (!occupied)
{
        if ( sqrt ( sq(obsX-proX) + sq(obsY-proY) ) < 2.0 && (*LP)[a]<maxLaser )
        {
                val = 1.0;
                occupied = true;
        }
        else
```

```
                    val = 0.0;
            }
            else
            {
                if (    (*LP)[a] == maxLaser )
                {
                    val = 0.0;
                    occupied = false;
                }

                else
                    val = 1.0;
            }

            PW->plot_blend( obsX , obsY , opacity , (1.0-val)*1.0 , (1.0-val)*1.0 , (1.0-val)*1.0 );
            matrixVal [obsX][obsY] = val;

            preX = obsX;
            preY = obsY;
            obsX = proX;
            obsY = proY;
        }

        if (b==0)
        {
            /// turn 180 degree
            if (P2P.GetYaw() ==0)
            {
                turn180 = PI;
                while ( (abs (P2P.GetYaw()- turn180)>0.005) && (abs (P2P.GetYaw() + turn180)
>0.005) )
                {
                    P2P.GoTo (x, y, turn180);
                    P2P.SetMotorEnable (true);
                    ROBOT.Read();
                }
            }

            else
            {
                if ( P2P.GetYaw() > 0 )
                    turn180 = P2P.GetYaw() - PI;
                else
                    turn180 = P2P.GetYaw() + PI;
```

```
                    while ( abs(P2P.GetYaw() - turn180) >0.005) /// 0.0005 = 0.286 degree
                    {
                        P2P.GoTo ( (x*cos(offsetAngle)) + (y*sin(offsetAngle)) , (y*cos(offsetAngle)) -
                            (x*sin(offsetAngle)) , turn180);
                        P2P.SetMotorEnable (true);
                        ROBOT.Read();
                    }
                }

                P2P.SetSpeed(0.0, 0.0);
                P2P.SetMotorEnable (true);
                ROBOT.Read();
                x = P2P.GetXPos()*cos(offsetAngle) + P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
                y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);
                angle = P2P.GetYaw() + offsetAngle;

                obsX = (int) approximate((x+offsetX)*10.0);
                obsY = (int) approximate((y+offsetY)*10.0);
                val = 0.0;
                PW->plot_blend ( obsX, obsY,opacity, (1.0-val)*1.0 , (1.0-val)*1.0 , (1.0-val)*1.0 );
                matrixVal [obsX][obsY] = val;
            }
        }
        PW->close();
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
/// detects if any existing unvisited point in list is re-scanned on not, if so, erase it
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
void eraseOverlap (LaserProxy* LP , Position2dProxy& P2P)
{
        int obsX , obsY ;
        int obsXX , obsYY;
        double x = P2P.GetXPos()*cos(offsetAngle) + P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
        double y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);
        double angle = P2P.GetYaw() + offsetAngle;
        double angle2;
        bool overlap = true;
        bool Erase = true;

        for (int a = 0; a < 361; a++)
        {
            angle2 = angle + ((a/2.0-90.0)*PI/180.0);
```

```
obsX = limit ( (int) approximate (((*LP)[a]*cos(angle2) + x + offsetX)*10.0) , 0 , 160);
obsY = limit ( (int) approximate (((*LP)[a]*sin(angle2) + y + offsetY)*10.0) , 0 , 160);

if( matrixVal [obsX][obsY] == 0.5 )
        overlap = false;

for (double i = 0.1; i<(*LP)[a] ;i+=0.1)
{
        obsXX = limit ( (int) approximate ((i*cos(angle2) + x + offsetX)*10.0) , 0 , 160);
        obsYY = limit ( (int) approximate ((i*sin(angle2) + y + offsetY)*10.0) , 0 , 160);

        if( matrixVal [obsXX][obsYY] == 0.5 )
                overlap = false;

        if ( !planX.empty() && forward)
        {
                listIteratorY = planY.begin();
                listIteratorV = visited.begin();
                Erase = true;

                for( listIteratorX = planX.begin(); listIteratorX != planX.end(); listIteratorX++ )
                {
                        if ( *listIteratorX == oriDepX && *listIteratorY == oriDepY && Erase)
                                Erase = false;

                        else if    ( *listIteratorX == mapDestX && *listIteratorY == mapDestX && Erase)
                                Erase = false;

                        else if ( *listIteratorX == obsXX && *listIteratorY == obsYY && Erase)
                        {
                                if ( *listIteratorV == 0)
                                {
                                        cout << "erase" << *listIteratorX <<endl;
                                        cout << "erase" << *listIteratorY <<endl;
                                        planX.erase (listIteratorX);
                                        planY.erase (listIteratorY);
                                        visited.erase (listIteratorV);
                                        break;
                                }
                                else
                                        break;
                        }
                        listIteratorY ++;
                        listIteratorV ++;
```

```
                }
            }
        }
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// checks if the map environment is completely explored and built
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
bool finishMapping ()
{
    bool finish = true;

    listIteratorV = visited.begin();
    for( listIteratorV = visited.begin(); listIteratorV != visited.end(); listIteratorV++ )
    {
        if ( *listIteratorV == 0)
        {
            finish = false;
            break;
        }
    }
    return finish;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// path finding using A* algorithm
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
bool pathPlanning ( int DepX , int DepY, int DestX , int DestY )
{
    int chainCode = 0;
    int parentX =0, parentY=0, parentG=0;
    int OLcount =0, CLcount =0;
    int OLloc, CLloc;
    int G = 0, H = 0;
    vector < vector <int> > openL ( 2, vector <int> (6, 0));
    vector < vector <int> > closeL ( 2, vector <int> (6, 0));

    openL.clear();
    closeL.clear();
```

```
if (extendSize( &openL, OLcount) )
{
        insertRow( &openL, openL.size()-1 , DepX, DepY, 0 , 0 , 0 , 0);
        OLcount ++;
}

else
{
        OLloc = checkLocAvailable( &openL , OLcount);
        insertRow( &openL, OLloc , DepX, DepY, 0 , 0 , 0 , 0);
        OLcount ++;
}

while ( (DepX != DestX) || (DepY != DestY))
{
        if (chainCode ==0)
        {
                OLloc = lowestFloc ( &openL );

                if (OLloc != -1)
                {
                        DepX = parentX = openL[OLloc][0];
                        DepY = parentY = openL[OLloc][1];
                        parentG = openL[OLloc][3];

                        if (extendSize( &closeL, CLcount) )
                        {
                                insertRow( &closeL , closeL.size()-1 , openL[OLloc][0] , openL[OLloc][1] ,
                                        openL[OLloc][2] , openL[OLloc][3] , openL[OLloc][4] , openL[OLloc][5] );
                                CLcount ++;
                        }

                        else
                        {
                                CLloc = checkLocAvailable( &closeL , CLcount);
                                insertRow( &closeL , CLloc , openL[OLloc][0] , openL[OLloc][1] ,
openL[OLloc][2] ,
                                        openL[OLloc][3] , openL[OLloc][4] , openL[OLloc][5] );
                                CLcount ++;
                        }

                        if ( abs(DestX-DepX) < 4 && abs(DestY - DepY) <4)
                        {
```

```
                    if (extendSize( &closeL, CLcount) )
                    {
                        insertRow(&closeL, closeL.size()-1 , DestX , DestY ,9,0,0,0);
                        CLcount ++;
                    }
                    pathDecision (&closeL, true);
                    return true;
                }
                clearRow ( &openL , OLloc );
                OLcount--;
                chainCode ++ ;
            }

            else
            {
                pathDecision (&closeL, false);
                return false;
            }
        }

        else
        {
            findDepXY ( chainCode , parentX , parentY , DepX , DepY );

            if ( (DepX != -1) && (DepY != -1) )
            {
                if ( notInCL ( &closeL , DepX , DepY) )
                {
                    G = findG( parentG , chainCode);
                    H = findH( DepX , DepY, DestX , DestY );

                    OLloc = inOLloc ( &openL , DepX , DepY );
                    if ( OLloc != -1 )
                    {
                        if ( G < openL[OLloc][3] )
                        {
                            openL[OLloc][2] = chainCode;
                            openL[OLloc][3] = G;
                            openL[OLloc][5] = G + openL[OLloc][4];
                            }
                    }
                    else
                    {
                        if ( !isObstacle(DepX,DepY, chainCode) )
```

```
                                    {
                                          if (extendSize( &openL, OLcount) )
                                          {
                                                insertRow( &openL, openL.size()-1 , DepX, DepY, chainCode , G ,
H ,
                                                      G+H );
                                                OLcount ++;
                                          }
                                          else
                                          {
                                                OLloc = checkLocAvailable( &openL , OLcount);
                                                insertRow( &openL, OLloc , DepX, DepY, chainCode , G , H ,
G+H );
                                                OLcount ++;
                                          }
                                    }
                              }
                        }
                  }
                  if ( chainCode < 8 )
                        chainCode ++;
                  else
                        chainCode =0;
            }
      }

      if (chainCode > 0)
            chainCode --;
      else
            chainCode = 8 ;

      if (extendSize( &closeL, CLcount) )
      {
            insertRow( &closeL , closeL.size()-1 , DepX, DepY, chainCode , G , H , G+H );
            CLcount ++;
      }

      pathDecision (&closeL, true);
      return true;
}
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – find G
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

```
////////
int findG ( int g , int arrow)
{
        if ( arrow == 0 )
                return g;
        else if ( arrow % 2 == 1 )
                return g+10;
        else
                return g+14;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – find H
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int findH ( int depX , int depY, int destX , int destY )
{
        /// using Manhattan Method
        int xDist = abs(depX -destX);
        int yDist = abs(depY -destY);
        return 10*(yDist + xDist);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – find F
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int findF ( int g , int h)
{
        return g+h;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – find location of lowest F in openL
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int lowestFloc( vector < vector <int> > *vec)
{
        int f, loc=0;

        if ( !(*vec).size() )
```

```
		return -1;

	else
	{
		for ( int a = 0 ; a < (*vec).size() ;a++)
		{
			if ( (*vec)[a][0] != 0)
			{
				f = (*vec)[a][5];
				loc = a;

				for ( int b = a+1 ; b < (*vec).size() ; b++)
				{
					if ( ((*vec)[b][0] >0) && ((*vec)[b][5] < f) )
					{
						f = (*vec)[b][5];
						loc = b;
					}
				}

				return loc;
			}
		}
	}

	return -1;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – extend size of openL or closeL, return 1 if succeed
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
bool extendSize( vector< vector <int> > *vec, int size )
{
	if(size == (*vec).size())
	{
		(*vec).resize( size +1);
		(*vec)[size].resize(6);
	}

	else
		return false;
}
```

```
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – return the location at which the pointer is available/idle
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int checkLocAvailable(vector< vector <int> > *vec , int size)
{
        for(int location=0;location<vec->size();location++)
        {
                if( (*vec)[location][0]==0)
                        return location;
        }
        return -1;
}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – determine the “child” point of the current “parent” point
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void findDepXY ( int arrow , int parentX , int parentY , int &DepX , int &DepY )
{
        if ( (arrow == 1) || (arrow == 2) || (arrow == 8) )
                DepX = parentX +1;
        if ( (arrow == 2) || (arrow == 3) || (arrow == 4) )
                DepY = parentY + 1;
        if ( (arrow == 4) || (arrow == 5) || (arrow == 6) )
                DepX = parentX - 1;
        if ( (arrow == 6) || (arrow == 7) || (arrow == 8) )
                DepY = parentY - 1;
        if ( (DepX < 6) || (DepX >154) )
                DepX = -1;
        if ( (DepY < 6) || (DepY >154) )
                DepY = -1;
}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – check if the current point is already exist in closed list or not
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
bool notInCL ( vector< vector <int> > *vec , int x , int y)
```

```
{
        for( int a = 0; a< (*vec).size() ; a++)
        {
                if (    ((*vec)[a][0] == x) && ((*vec)[a][1] == y) )
                        return false;
        }
        return true;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – check if current point is already exist in open list or not, if so,
///                         determine its location
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
int inOLloc ( vector< vector <int> > *vec , int x , int y)
{
        for( int a = 0; a< (*vec).size() ; a++)
        {
                if (    ((*vec)[a][0] == x) && ((*vec)[a][1] == y) )
                        return a;
        }
        return -1;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – check if the current point is obstacle or not
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
bool isObstacle ( int x , int y , int arrow )
{
        if ( (x<6) || (x>154) || (y<6) || (y>154) )
                return true;

        if ( arrow == 1 || arrow == 2 || arrow == 8 )
        {
                for ( int a = (y-5) ; a < (y+6) ; a++)
                {
                        if ( (matrixVal[x][a] >= 0.5) || (matrixVal[x+1][a] >= 0.5) || (matrixVal[x+2][a] >= 0.5) ||
                        (matrixVal[x+3][a] >= 0.5) || (matrixVal[x+4][a] >= 0.5) || (matrixVal[x+5][a] >= 0.5)    )
                                return true;
                }
        }
```

```
        if ( arrow == 2 || arrow == 3 || arrow == 4 )
        {
                for ( int b = (x-5) ; b < (x+6) ; b++)
                {
                        if ( (matrixVal[b][y] >= 0.5) || (matrixVal[b][y+1] >= 0.5) || (matrixVal[b][y+2] >= 0.5) ||
                        (matrixVal[b][y+3] >= 0.5) || (matrixVal[b][y+4] >= 0.5) || (matrixVal[b][y+5] >= 0.5)    )
                                return true;
                }
        }

        if ( arrow == 4 || arrow == 5 || arrow == 6 )
        {
                for ( int c = (y-5) ; c < (y+6) ; c++)
                {
                        if ( (matrixVal[x][c] >= 0.5) || (matrixVal[x-1][c] >= 0.5) || (matrixVal[x-2][c] >= 0.5) ||
                        (matrixVal[x-3][c] >= 0.5) || (matrixVal[x-4][c] >= 0.5) || (matrixVal[x-5][c] >= 0.5)    )
                                return true;
                }
        }

        if ( arrow == 6 || arrow == 7 || arrow == 8 )
        {
                for ( int d = (x-5) ; d < (x+6) ; d++)
                {
                        if ( (matrixVal[d][y] >= 0.5) || (matrixVal[d][y-1] >= 0.5) || (matrixVal[d][y-2] >= 0.5) ||
                        (matrixVal[d][y-3] >= 0.5) || (matrixVal[d][y-4] >= 0.5) || (matrixVal[d][y-5] >= 0.5)    )
                                return true;
                }
        }
        return false;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – record the related information into open list or closed list
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void insertRow( vector< vector <int> > *vec , int loc , int x, int y, int arrow , int g , int h , int f)
{
        (*vec)[loc][0] = x;
        (*vec)[loc][1] = y;
        (*vec)[loc][2] = arrow;
        (*vec)[loc][3] = g;
```

```
        (*vec)[loc][4] = h;
        (*vec)[loc][5] = f;
}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – clear specified information
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void clearRow(    vector< vector <int> > *vec , int loc )
{
        (*vec)[loc][0] = 0;
        (*vec)[loc][1] = 0;
        (*vec)[loc][2] = 0;
        (*vec)[loc][3] = 0;
        (*vec)[loc][4] = 0;
        (*vec)[loc][5] = 0;
}

/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// A* algorithm – path tracing once goal is found
/////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void pathDecision ( vector< vector <int> > *vec , bool found)
{
        int px=0, py=0, code=0, ploc=0;
        navigablePath.clear();
        navigablePath.resize(1);
        navigablePath[0].resize(2);

        if (found == true)
        {
                if ( (*vec)[(*vec).size()-1][2] != 9 )
                {
                        px = navigablePath[0][0] = (*vec)[(*vec).size()-1][0];
                        py = navigablePath[0][1] = (*vec)[(*vec).size()-1][1];
                        code = (*vec)[(*vec).size()-1][2];
                }

                else
                {
                        navigablePath[0][0] = (*vec)[(*vec).size()-1][0];
                        navigablePath[0][1] = (*vec)[(*vec).size()-1][1];
```

```
            navigablePath.resize(2);
            navigablePath[1].resize(2);

            px = navigablePath[1][0] = (*vec)[(*vec).size()-2][0];
            py = navigablePath[1][1] = (*vec)[(*vec).size()-2][1];
            code = (*vec)[(*vec).size()-2][2];
        }
}

else
{
        for (int b = 0; b < (*vec).size()-1 ;b++)
        {
            if ( (abs((*vec)[b+1][0] - (*vec)[b][0]) > 1) || (abs((*vec)[b+1][1] – (*vec)[b][1]) > 1) )
            {
                px = navigablePath[0][0] = (*vec)[b][0];
                py = navigablePath[0][1] = (*vec)[b][1];
                code = (*vec)[b][2];
                break;
            }
        }

        if ( navigablePath[0][0] == 0)
        {
            px = navigablePath[0][0] = (*vec)[(*vec).size()-1][0];
            py = navigablePath[0][1] = (*vec)[(*vec).size()-1][1];
            code = (*vec)[(*vec).size()-1][2];
        }
}

while (code)
{
        if ( (code==1) || (code==2) || (code==8) )
            px = px-1;
        if ( (code==2) || (code==3) || (code==4) )
            py = py-1;
        if ( (code==4) || (code==5) || (code==6) )
            px = px+1;
        if ( (code==6) || (code==7) || (code==8) )
            py = py+1;

        for (int a = 0; a<(*vec).size() ;a++)
        {
```

```
                        if ( ((*vec)[a][0] == px) && ((*vec)[a][1] == py)    )
                        {
                                px = (*vec)[a][0];
                                py = (*vec)[a][1];

                                if ( (*vec)[a][2] != code )
                                {
                                        navigablePath.resize( navigablePath.size() +1);
                                        navigablePath[navigablePath.size()-1].resize(2);

                                        navigablePath[navigablePath.size()-1][0] = (*vec)[a][0];
                                        navigablePath[navigablePath.size()-1][1] = (*vec)[a][1];
                                        code = (*vec)[a][2];
                                }
                                break;
                        }
                }
        }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// robot navigation
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void moveRobot(LaserProxy* LP , Position2dProxy& P2P , PlayerClient & ROBOT)
{
        double x ;
        double y ;
        double goX,goY;
        int mapPoseX, mapPoseY;

        x = P2P.GetXPos()*cos(offsetAngle) + P2P.GetYPos()*cos(PI/2.0 + offsetAngle);
        y = P2P.GetXPos()*sin(offsetAngle) + P2P.GetYPos()*sin(PI/2.0 + offsetAngle);

        mapPoseX = limit ( (int) approximate ((x + offsetX)*10.0) , 0 , 160);
        mapPoseY = limit ( (int) approximate ((y + offsetY)*10.0) , 0 , 160);

        if ( (mapPoseX !=    mapDestX) || (mapPoseY != mapDestY ) )
        {
                x = (mapDestX/10.0)-offsetX;
                y = (mapDestY/10.0)-offsetY;
                goX = (x*cos(offsetAngle)) + (y*sin(offsetAngle));
                goY = (y*cos(offsetAngle)) - (x*sin(offsetAngle)) ;
```

```
        P2P.GoTo ( goX , goY , 0.0);
    }

    else if ( (mapPoseX == mapDestX) && (mapPoseY == mapDestY ) )
    {
        if ( navigablePath.size() > 1)
        {
            navigablePath.resize(navigablePath.size()-1);
            mapDestX = navigablePath[navigablePath.size()-1][0];
            mapDestY = navigablePath[navigablePath.size()-1][1];
        }

        else
            navigablePath.clear();

        P2P.SetSpeed (0.0,0.0);
    }
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// convert a float or double value approximately to integer value
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
template<typename T>
inline T approximate(T a)
{
    T b = a - floor (a);

    if ( (b>=0.0) && (b <0.5) )
    return floor(a);
    else
    return floor(a) + 1.0;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// calculate square of a number
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
template<typename T>
inline T sq ( T a)
{
    return a*a;
```

```
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
/// uses ostream_iterator and copy algorithm to output list elements
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////
void printList( const list< T > &listRef )
{
        if ( listRef.empty() )
                cout << "List is empty";

        else
        {
                ostream_iterator< T > output( cout, " " );
                copy( listRef.begin(), listRef.end(), output );
        }
}
```