ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

AS A PARTIAL REQUIREMENT
TO OBTAIN A
MASTERS IN ELECTRICAL ENGINEERING
M.Eng.

BY
ELIE MAALOUF

NONLINEAR CONTROL OF WHEELED MOBILE ROBOTS

MONTREAL, 2 SEPTEMBER 2005

THIS THESIS HAS BEEN EVALUATED

BY A JURY COMPOSED OF:

Mr. Maarouf Saad, projet director
Department of electrical engineering at École de technologie supérieure

Mr. Hamadou Saliah-Hassne, codirector
Télé Université

Mr. Vahé Nerguizian, president of jury
Department of electrical engineering at École de technologie supérieure

Mr. Bruno De Kelper, member of jury
Department of electrical engineering at École de technologie supérieure

IT HAS BEEN PRESENTED BEFORE A JURY AND PUBLIC

ON THE 22 OF JUNE 2005

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

NONLINEAR CONTROL OF A WHEELED MOBILE ROBOT

Elie Maalouf

SUMMARY

The purpose of this project is to implement an autonomous navigation system using nonlinear control techniques to control a wheeled mobile robot (WMR) to follow a preplanned trajectory and track a path. Two other aspects of navigation are studied: path planning and obstacle avoidance. Those three aspects are integrated into a navigation strategy that manages navigation and prevents deadlocks.

Two nonlinear control techniques for path tracking and trajectory following have been developed and implemented. In the first approach, a fuzzy logic controller is used to drive the robot through a set of waypoints leading to the destination. In another approach, a controller derived from a Lyapunov function is used to track a reference trajectory that is time dependent. For path planning, a novel optimization technique based on dynamic programming has been developed. The curvature velocity method has been used for obstacle avoidance.

The testing was conducted on a P3-AT all-terrain mobile robot equipped with encoders, a gyroscope, and sonar sensors for localization and environment perception. The test results validate the effectiveness of the different approaches that have been developed.

NONLINEAR CONTROL OF A WHEELED MOBILE ROBOT

Elie Maalouf

ABSTRACT

The purpose of this project is to implement an autonomous navigation system using nonlinear control techniques to control a wheeled mobile robot (WMR) to follow a preplanned trajectory and track a path. Two other aspects of navigation are studied: path planning and obstacle avoidance. Those three aspects are integrated into a navigation strategy that manages navigation and prevents deadlocks.

Two nonlinear control techniques for path tracking and trajectory following have been developed and implemented. In the first approach, a fuzzy logic controller is used to drive the robot through a set of waypoints leading to the destination. In another approach, a controller derived from a Lyapunov function is used to track a reference trajectory that is time dependent. For path planning, a novel optimization technique based on dynamic programming has been developed. The curvature velocity method has been used for obstacle avoidance.

The testing was conducted on a P3-AT all-terrain mobile robot equipped with encoders, a gyroscope, and sonar sensors for localization and environment perception. The test results validate the effectiveness of the different approaches that have been developed.

COMMANDE NONLINÉAIRE D'UN ROBOT MOBILE À ROUES

Elie Maalouf

SOMMAIRE

Le but de ce projet de recherche est de développer un système de navigation autonome, en utilisant des méthodes de commande non-linéaires pour contrôler un robot mobile à roues pour le suivi d'un chemin ou la poursuite d'une trajectoire. Deux autres aspects de navigation sont examinés, la planification de trajectoire et l'évitement des obstacles. Les trois aspects sont intégrés dans une stratégie de navigation afin d'éviter le blocage du robot et de bien gérer la navigation.

Deux techniques de commande non-linéaires ont été développées et implémentées pour deux types différents d'exécution de trajectoire. Dans la première approche, un contrôleur logique floue est utilisé pour contrôler le robot pour suivre des points intermédiaires menant à la destination. Dans la deuxième approche où la trajectoire de référence est en fonction du temps, un contrôleur dérivé d'une fonction de Lyapunov a été implanté. Pour la planification de trajectoire, une nouvelle technique d'optimisation basée sur la programmation dynamique a été développée. Pour l'évitement d'obstacles, la méthode des courbures-vitesses a été implémentée.

Les tests ont été conduits sur le robot tout terrain P3AT équipé d'encodeurs, d'un gyroscope, ainsi que des capteurs sonar pour la localisation et la perception de l'environnement. Les résultats des tests ont validé la performance des travaux réalisés.

COMMANDE NONLINÉAIRE DES ROBOTS MOBILES

Elie Maalouf

RÉSUMÉ

Lors des deux dernières décennies, de nombreuses recherches sur la navigation des robots mobiles à roues motrices ont été réalisées. L'avancement en technologie VLSI et la disponibilité de processeurs performants à des prix compétitifs a largement contribué au développement de systèmes des robots mobiles autonomes et semi autonomes. Les robots mobiles sont de plus en plus disponibles sur le marché pour des applications dans l'agriculture, l'industrie des mines, l'exploration spatiale, les domaines militaires, ainsi que dans de nombreuses autres applications où l'environnement peut être hostile à des êtres humains. La navigation en est un aspect commun dans tous les systèmes de robots mobiles. En gros, la navigation consiste en plusieurs tâches tel que la perception de l'environnement, la localisation, la planification de trajectoire, l'évitement d'obstacles, ainsi que l'exécution d'une trajectoire. Le but de ce projet est de réaliser un système de navigation dont la localisation et la perception de l'environnement sont disponibles. Une nouvelle technique de planification de trajectoire a été développée en partant de la programmation dynamique classique. Deux approches d'exécution d'une trajectoire ont été réalisées, une fait appel à un contrôleur logique floue et l'autre en utilisant un contrôleur dérivé à partir d'une fonction de Lyapunov. Pour ce qui est de l'évitement d'obstacles, la méthode de courbure-vitesse a été utilisée. Le tout sera intégré dans une stratégie de navigation sous la forme d'une machine à états finis. Dû à des contraintes de temps, cette dernière étape n'a pas été implantée en temps réel, mais a été expliquée en détail dans le Chapitre 7. Les trois éléments de navigation ont été implantés en temps réel sur le robot P3-AT disponible dans le laboratoire du GREPCI à l'ÉTS. Ce robot robuste et performant est équipé d'encodeurs, d'un gyroscope bidimensionnel pour la localisation et de capteurs sonars pour la perception de l'environnement et la détection

d'obstacles. La commande sur la dynamique n'est pas accessible, ce qui nous emmène à faire une commande sur le modèle cinématique du robot. Le robot est contrôlé par un microcontrôleur embarqué qui communique avec un PC ou un ordinateur embarqué par un port série. Une interface de haut niveau sous la forme d'une librairie C++ orientée objet dénommée ARIA s'occupe d'établir la communication avec le microcontrôleur alors que l'utilisateur peut mettre son effort sur la commande du robot à haut niveau. Le calcul à haut niveau est alors fait sur l'ordinateur, tandis que l'acquisition des données des capteurs et la commande à bas niveau des moteurs sont faites par le microcontrôleur. Le robot P3AT est un robot mobile à quatre roues qui est conduit par une traction différentielle, qu'on distingue des robots mobiles de type voiture et les autres classes mentionnées dans la première partie du Chapitre 3. Le modèle cinématique en coordonnées Cartésiennes ainsi que le modèle dynamique d'un robot à traction différentielle sont développés dans la deuxième partie du troisième chapitre.

La planification de trajectoire est faite lorsqu'un modèle de l'environnement du robot est disponible d'avance, soit sous la forme d'une image ou d'autre. Le modèle de l'environnement est discrétisé et est mit sous la forme d'une matrice de coût. En supposant que le robot se déplace dans un environnement bidimensionnel, l'environnement peut être classifié par des régions accessibles et des régions inaccessibles (les obstacles). Les éléments de la matrice qui correspondent aux régions inaccessibles sont négligés, et le reste est modélisé par des nœuds liés par des liens avec un coût qui correspond à la distance entre les nœuds. Plusieurs techniques ont été développées dans la littérature pour trouver un chemin optimal entre le nœud qui correspond à la position initiale du robot et le nœud qui correspond à une nouvelle position désirée dans l'environnement. La plus célèbre de ces techniques est l'algorithme de recherche A* de graphes et ses différentes variantes. Cette technique fait appel à une fonction heuristique pour estimer le coût jusqu'au nœud final à partir du nœud initial et les nœuds intermédiaires. La fonction heuristique est choisie par le programmeur, et la performance en dépend par conséquent. Dans ce projet, une nouvelle

technique pour déterminer une solution optimale a été développée en se basant sur la programmation dynamique classique qui ne peut pas être utilisée tel quel pour résoudre un graphe. Après un nombre d'itérations déterministe sur les différentes couches, une solution garantie optimale peut être trouvée. Cette nouvelle technique est capable d'exploiter le parallélisme dans un processeur multi-unités de traitement, tel qu'un DSP. Le Chapitre 4 contient les détails de cette approche, ainsi qu'une preuve formelle que la solution converge vers la solution optimale. Le seul défaut de cette approche par rapport à l'algorithme de recherche A* est le temps de calcul. Cependant, ce calcul est fait hors ligne lorsque le robot n'est pas entrain de se déplacer, et la performance, du temps réel du robot, ne sera pas altérée aucunement. Cette approche a été implémentée sur MATLAB dont l'interface facilite le développement rapide de programmes. La solution optimale est donnée sous la forme d'une liste de nœuds intermédiaires entre le premier nœud et la position finale désirée. Une trajectoire temporelle peut être définie à partir de cette liste de nœuds correspondants à des positions par rapport à un repère de référence globale. La programmation dynamique itérative trouve actuellement toutes les trajectoires optimales par rapport à tous les nœuds (le nœud initial bien-compris) vers le nœud final. Cette information est stockée en termes de pointeurs entre les nœuds qui mènent à la position finale en passant sur la trajectoire optimale. Chaque nœud contient un seul pointeur vers un autre nœud. En suivant les pointeurs à partir d'un certain nœud, on arrive obligatoirement vers le nœud final sur une trajectoire optimale. Cette méthode a été testée sur de nombreux cas, et on obtient une trajectoire optimale dans tous les cas.

En utilisant la solution obtenue par la trajectoire optimale, un contrôleur est nécessaire pour conduire le robot sur cette trajectoire. Une approche pour le cas où la solution optimale n'est pas limitée par des contraintes temporelles a été développée. Un contrôleur par logique floue inspiré de la conduite d'une voiture par un être humain a été réalisé. La trajectoire désirée consiste de la liste des nœuds positions qui forme la solution optimale et sont utilisées en tant que points de référence 'waypoints', et le robot traverse vers sa destination à proximité de ces points. Selon la vitesse actuelle et la courbure du trajet, le conducteur d'une voiture modifie la vitesse et oriente la voiture selon sa perception visuelle. Si la courbure est faible, il garde la même vitesse en passant. Selon la courbure il agit sur la vitesse et la direction de façon à obtenir une trajectoire lisse et continue. Le 'waypoint' actuel est défini comme étant le nœud le plus proche du robot sur la trajectoire optimale. Les deux angles entre le nœud actuel et les deux prochains nœuds sont calculés pour déterminer un facteur qui correspond au degré de la courbure. Ce facteur est utilisé comme une entrée 'feed-forward' pour le contrôleur à logique floue. Les autres entrées du contrôleur sont les erreurs de position et d'orientation ainsi que la vitesse actuelle. L'état de chaque entrée est déterminé par des fonctions membres d'une manière continue selon le niveau d'appartenance à un certain état. Une certaine valeur d'une entrée peut appartenir à un ou plusieurs états. Les sortis du contrôleur de type Takagi-Sugeno utilisé dans ce projet ont des états discrets. Une base de règles d'inférence fait le lien entre les états des entrées et les états des sorties. Les sorties des contrôleurs sont les vitesses linéaire et angulaire, et sont calculées selon le poids de chaque règle par rapport aux entrées. Le contrôleur à logique floue est robuste et fiable, malgré la difficulté de prouver sa stabilité théoriquement pour quatre entrées. Pour implanter le contrôleur à logique floue, la librairie Free Fuzzy Logic Library (FFLL) qui respecte un standard industriel sous forme de fichier FCL a été utilisée. Les résultats de tests ont prouvé la fiabilité de cette approche.

Une autre approche dont la trajectoire est en fonction du temps a été implémentée. Le contrôleur est dérivé par une fonction de Lyapunov, ce qui assure la stabilité du système. Les erreurs en position et en orientation sont développées pour obtenir les équations des erreurs sous forme d'un modèle dans l'espace d'états. La fonction d'énergie de Lyapunov est choisie de telle manière à stabiliser les variables d'erreur autour de zéro. La trajectoire de référence est obtenue en interpolant les positions de la trajectoire optimale et en les dérivant pour obtenir l'orientation de références en fonction du temps. Cette approche est moins robuste que l'approche précédente et a été utilisée pour faire une étude théorique, malgré sa performance adéquate.

Une approche en temps réel pour l'évitement des obstacles en cas d'obstacles dynamiques ou de changements dans l'environnement lorsque le robot est en train d'exécuter une trajectoire est nécessaire pour éviter des collisions éventuelles avec d'autres objets. La méthode de courbure-vitesse pour l'évitement des obstacles qui est bien connue dans la littérature a été le choix dans ce projet. Cette approche est notamment connue pour sa fiabilité et son efficacité en temps de calcul. Cette méthode peut être utilisée également pour explorer un environnement inconnu. Les obstacles perçus par les capteurs sonar sont modélisés sous forme de cercles à rayon constant, ce qui est convenable pour cette méthode qui fait appel à l'espace de courbure. Les positions des centres des obstacles sont déterminées selon un calcul géométrique simple. Les obstacles sont élargis par le rayon du robot pour être capable de modéliser le robot comme étant un point dans l'espace cartésien, ce qui sert à simplifier les calculs. Les obstacles à plus de trois mètres de robot seront négligés. Cette méthode consiste essentiellement à optimiser une fonction linéaire dans l'espace des vitesses par rapport à trois critères : la vitesse de navigation, la sécurité de navigation, et la recherche de la position cible désirée. Une ligne droite tracée passant par l'origine du robot correspond à une courbure dans l'espace cartésien du repaire de référence global (voir Chapitre 6 pour plus de détails). L'espace des vitesses est divisé sur des intervalles d'obstacles par des lignes droites passant par l'origine, et limité par les contraintes dynamiques du robot tel que les accélérations et les vitesses maximales. Il s'agit de trouver un point dans ces intervalles qui maximisent la fonction objective. Ce point correspond à un couple de vitesses linéaire et angulaire. Des approximations ont été faites pour respecter l'efficacité du calcul en temps réel. Chaque intervalle entre deux courbures correspond à une distance entre le robot et l'obstacle. Les distances des intervalles qui ne contiennent pas d'obstacles sont fixées à trois mètres. Les intervalles créent des zones triangulaires dans l'espace des vitesses émanant de l'origine et sont limités par les contraintes sur les vitesses et accélérations. La fonction objective dépend linéairement de la vitesse linéaire, de la distance associée à l'intervalle et de l'erreur d'orientation par rapport à la position de destination. La fonction objective est alors maximale sur les extrémités extérieures, ce qui rend le temps de calcul assez raisonnable. Le calcul sera fait pour deux points additionnels, l'origine et le point qui oriente le robot vers la position désirée. Ceci étant dit, une stratégie de navigation pour faire la coordination entre les trois aspects de navigation et pour éviter que le robot ne soit bloqué est nécessaire. La stratégie de navigation est implantée sous forme d'une machine à états finis similaire à une architecture hybride de commande. Sous cette architecture, les résultats de perception de l'environnement sont utilisés par le module de planification de la trajectoire qui est considéré à haut niveau ainsi que par les modules à plus bas niveau dans le sens hiérarchique tel que l'exécution d'une trajectoire et l'évitement d'obstacles qui sont des actions de type comportemental. La machine à états est toujours dans un de quatre modes : mode d'arrêt quand le robot est stationnaire, mode de planification de trajectoire, mode d'exécution de trajectoire, ou en mode d'évitement d'obstacle. Au début le robot est en mode d'arrêt, et si un modèle de l'environnement est disponible, le robot passe en mode de planification de trajectoire. Lorsque la trajectoire est planifiée, le robot sera mit en mode d'exécution de trajectoire. Si un obstacle est détecté sur le chemin du robot, le robot passera en mode d'évitement d'obstacles, avec un nœud cible qui correspond à un 'waypoint' sur la trajectoire après l'obstacle. Quand le nœud cible intermédiaire est atteint, si un obstacle est encore sur la trajectoire, le robot reste en mode d'évitement d'obstacles et une autre cible intermédiaire est fixée. Lorsqu'il n'y a plus d'obstacle sur la trajectoire, le robot passe en mode d'exécution de trajectoire. Dans le programme de la machine à états, on vérifie continuellement si le robot passe dans une certaine région plus qu'une fois. Si c'est le cas, le robot passe en mode d'arrêt et puis en mode d'exécution de trajectoire. Le modèle de l'environnement est toujours modifié selon les valeurs retournées par les capteurs sonar.

Dans le cas où l'environnement n'est pas connu d'avance, le robot passe en mode d'évitement d'obstacles et construit l'environnement au fur et à mesure qu'il avance. Si le robot est bloqué, le robot passe en mode d'arrêt et puis en mode de planification de trajectoire en utilisation toute information disponible sur l'environnement jusqu'à présent, et les régions inconnues seront considérées comme étant accessibles. Il y a toujours des limitations pratiques sur cette approche étant donné que les encodeurs de vitesse des moteurs ont une erreur de positionnement qui augmente tant que le robot se déplace. La même chose arrive avec le gyroscope, ce qui mène à une erreur d'orientation qui augmente avec le temps.

En conclusion, ce travail a été une opportunité pour la réalisation d'un prototype pour un système de navigation. D'autres étudiants qui vont faire la recherche sur le même sujet pourraient ajouter d'autres fonctionnalités tel qu'il est mentionné dans la section des recommandations à la fin de ce document.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to Maarouf Saad, my project director for his help and encouragement, and for his precious advice and remarks all throughout the project.

I express my thanks also to Hamadou Saliah, my project codirector, for his help and encouragement. I equally thank Vahé Nerguizian, president of the jury.

My sincere thanks go to Bruno De Kelper, member of the jury, for his valuable remarks and suggestions that helped enrich this document in terms of content and quality.

Finally and very specially, I dedicate this project to my parents for their help and support all throughout my studies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS AND SYMBOLS

AMR	Autonomous Mobile Robot
ARIA	Activmedia Robot Interface for Applications
AROS	Activmedia Robotics Operating System
CVM	Curvature-Velocity Method
FCL	Fuzzy Control Language
FFLL	Free Fuzzy Logic Library
FLC	Fuzzy Logic Control
IEEE	Institute of Electrical and Electronic Engineers
LAC	Look-Ahead Curvature
P3-AT	Pioneer 3 – All Terrain
RV	Rotational Velocity, rds/sec
SIP	Server Information Packets
WMR	Wheeled Mobile Robot
$\alpha 1$, $\alpha 2$	Curvature angles
c	Curvature
C	Curvature steepness
C^+	Current optimal cost
C^*	Optimal cost between nodes
DP	Dynamic programming operator
e_x, e_y	Errors in positions in x and y respectively
f	Objective function to be optimized

F	Traction force
G	Cost to move from starting node to current node
H	Estimated cost to reach destination node
λ	Vector of eigenvalues of a matrix
L	Robot length in meters
L_{max}	Maximum sonar distance in meters
m	Mass of robot in Kg
R	Wheel radius
$R(\theta)$	Rotation matrix with respect to. global frame of reference
θ	Orientation with respect to the global frame of reference in radians
θ_g	Goal heading
τ_i	Torque on wheel *i*
t	Time in seconds
T_c	Time constant in seconds
TV	Translational Velocity, m/sec
V	Translational velocity
V_o	Lyapunov energy function
V_{max}	Maximum translational velocity
ω	Angular velocity in radians per second
w	Robot width in meters
x, y	Robot coordinates

$\dot{x}, \dot{y}$	Speeds in x and y directions respectively
$\ddot{x}, \ddot{y}$	Accelerations in x and y directions respectively

INTRODUCTION

Wheeled mobile robots (WMR) are widely used today in a variety of fields such as agriculture, industry, land mining, military applications, space exploration, and in many other applications where the environment is inaccessible or hazardous to humans such as in nuclear plants. Mobile robot navigation is a common aspect to all applications involving WMR, and is a common research topic in almost all engineering faculties.

For stationary robots such as manipulators with a fixed base, a rather complex dynamic controller is used to precisely control the motion of the robot, while trajectory planning and tracking is more easily achieved since the environment of the robot can be easily modeled and perceived. In the case of wheeled mobile robots, the problems of path planning, trajectory following, and obstacle avoidance are the more challenging and interesting topics.

The objective of this project is to develop an autonomous navigation system for the P3-AT robot available at the GREPCI research laboratory at ETS. Localization and sonar perception routines are already available with the robot through the ARIA application interface. The aim is to develop an optimal path planning technique to make good usage of any available information on the environment, and to develop a control technique to drive the robot along the planned path.

If no information is available about the environment or if a dynamic obstacle suddenly emerges, an obstacle avoidance technique that can autonomously drive the robot safely to the destination position without colliding with objects on its way is desirable. Those three aspects along with the already available localization and perception of the environment are then integrated into a navigation strategy that can detect failures and local minima and coordinate actions.

The project has been carried out having in mind that the navigation system might be adapted later on for navigation on a rugged three dimensional terrain.

The main objectives of the project are as follows:

- Evaluation of several path tracking techniques and the development and implementation of two techniques, one for path following and the other for trajectory tracking;
- Development of a path planning technique that can be used to plan a trajectory in a three-dimensional environment;
- Study and implementation of a real time obstacle avoidance technique available in the literature;
- Integration of all developed aspects in a reliable and robust navigation strategy that prevents the robot from getting blocked.

At the beginning of the project, a literature review of the available methodologies and techniques for robot navigation and control related to path planning, path tracking, and obstacle avoidance was performed. Some of the most remarkable researches done in these fields were discussed and evaluated. In the second phase of the project, a path planning algorithm was developed, analyzed, and tested. When the robot shipment arrived, the main work was to get familiarized with the ARIA robot interface and the operation of the robot, as well as with the user interface. Then two control techniques for path following and trajectory tracking were developed and implemented in real time. The first technique is based on fuzzy logic control while the second one is derived from a Lyapunov function and is based on an article available in the literature. Afterwards, an obstacle avoidance technique widely referred to in the literature was developed and implemented. Finally, a navigation strategy in the form of a state machine to integrate the three aspects of navigation developed throughout this project. This navigation strategy fits in the class of hybrid control architectures.

This step by step and modular structured approach to the project helped simplify the development, testing, and debugging.

CHAPTER 1

LITERATURE REVIEW

1.1 Introduction

During the past two decades, extensive research effort has been invested in all aspects of mobile robot navigation. The revolutionary advances that the VLSI industry has witnessed and the availability of high performance embedded systems at reasonable prices had a direct impact on the developments in mobile robotics. A literature review of what has been achieved in path planning, path tracking, and real time obstacle avoidance is presented. In the last section, mobile robot navigation architectures are briefly discussed and their use in the context of this project is explained.

1.2 Path Planning and Trajectory Generation

In many applications, a model of the environment in which the robot operates is often available. It would be quite advantageous to use this information to plan an optimal path even if some changes in the environment might occur in real time due to the appearance of some dynamic obstacles. In Chapter 3 of Pruski (1996), several of the techniques used in modeling static environments are described. In a commonly used approach, the environment is sampled at regular intervals and projected on a two dimensional space. The discrete samples are referred to as nodes, with each node linked to all adjacent nodes through links as in a graph. Each link is assigned a weight that would be calculated based on some optimization criteria, such as the safety of the robot, the time needed to traverse between two nodes, as well as other criteria that are task dependent.

A path planning algorithm is used to determine an optimal path from the current (start) position of the robot to the desired destination position, also modeled as nodes. There are numerous optimization algorithms that can be used for path planning. The most widely used is the A* search algorithm (Pruski, 1996) and some of its variants (hierarchical, differential, D*, parallel A*). Other algorithms that can be used include Dijkstra and numerous tree searching techniques (Pruski, 1996). In this article, a new algorithm based on dynamic programming that was developed will be described. The creation and development of this algorithm was inspired by the work published in Gifford and Murphy (1996). The advantage of this approach is that an optimal path is guaranteed, and the optimal paths from all nodes (positions) in the environment towards the destination nodes are determined in the process, which is useful in the case of a multi-robot system. Another advantage is the ability to use parallelism when implemented on a parallel processor without any compromise in the optimality of the solution.

1.3 Path Following and Trajectory Tracking

In mobile robot navigation, the path tracking controller is usually implemented at a low level in the control hierarchy. Its function is to execute a path planned by the higher level path planner with the least possible error in position and with minimal control effort. The high level planner's function is planning a path either offline or online and depending on environment changes. The path can also be generated in such a way as to simply follow another robot. The task of the lower level path tracker controller is to guarantee that the robot will track the path in a precise, reliable, and efficient manner. The path following problem is highly nonlinear, and several approaches have been developed to solve the problem of path tracking through direct control of the robot's dynamics. In some of these approaches (Xu and Yang, 2001; DeSantis, Hurteau et al., 2002; Zhang, Xu et al., 2002), nonlinear controllers are derived based on the Lyapunov approach. Other types of controllers were designed (Koh and Cho, 1995; Caracciolo, de Luca et al., 1999; Zhang, Xu et al., 2001) using sliding mode control or other nonlinear techniques and were applied to the nonlinear dynamic and kinematical model of the robot. A behavioral approach (Yang, Li et al., 2003) for path tracking was implemented using fuzzy logic control for wheel steering.

Irrespective of the performance of these approaches, they cannot be implemented if the robot dynamics are inaccessible. If no direct control on motor torques and traction forces can be done, such techniques cannot be used. A controller at a higher level can be used to solve this problem. Motion is controlled using the kinematic model of the robot as the system. The control law has to respect the kinematic constraints. The variables calculated by the control law are the translational and rotational velocities, based on the position, orientation, and the current values of the translational and rotational velocities. Here is a brief overview of some of the control techniques for control at the higher level on the kinematic model.

Many control techniques have been developed and proposed to control the robot at the kinematic level. In Xu and Yang (2001), a controller is implemented using a biologically inspired shunting model integrated into a bang-bang controller. In another approach (Weiguo, Huitang et al., 1999), a controller is designed using a back-stepping technique. In Ollero and Heredia (1995) the stability of a pure pursuit path tracking algorithm is analyzed for a kinematical model using a linearized kinematical model. The analysis is done for the case of a straight line and a circle, with the reasoning that most trajectories can be decomposed into pieces of constant curvature.

A generalization of the quadrature curve approach (Yoshizawa, Hashimoto et al., 1996) has been implemented. The idea is to make the robot follow a quadratic curve to a reference point on a desired path. The reference point is moved in time until the goal destination reached.

A path tracking algorithm that uses a scalar controller (Davidson and Bahl, 2001) based on static path geometry with position feedback has been implemented on three types of wheeled mobile robots, one of which is differentially steered.

Despite the interesting features of all these controllers, they are difficult to tune, in contrast to the flexibility that fuzzy logic control provides. Fuzzy logic control (FLC) is an interesting tool to be applied to the problem of path tracking since the output varies smoothly as the input changes. In this project, we will discuss a fuzzy path tracking controller designed based on expert experience and knowledge that was implemented on a four-wheel differentially steered mobile robot. The rules are based on reasoning similar to that of a human driving a car on a road that is free of obstacles and other cars. If the road is straight, the driver can displace at higher speeds. When faced with a curvature, he lowers the speed and makes a smooth turn. The behavior of the human driver is apparent in the fuzzy inference rules. The membership functions were derived based on the kinematical constraints of the robot. The built-in PID controllers were used to control the vehicle dynamics. Each wheel is controlled separately, and a user can only change the gains of the PID, which is virtually futile for the purpose of path tracking.

If the built-in dynamic level controller doesn't perform properly in accordance with the controller at the kinematic level due to changes in the physical properties of the surface, an adjustment technique can be used to sidestep the problem. An intelligent predictive control approach that adapts the reference inputs (velocities) based on real-time learning (Yang, He et al., 1998) is one good technique to be explored and added in cascade after the path tracking controller in the control loop in case the robot dynamics are inaccessible.

Some of the path tracking control techniques mentioned above will be discussed in more detail in what follows.

Direct Wheel Servo Control: Path tracking at the lowest level of the control hierarchy is done through direct servo control of the wheel motors. In (Koh and Cho, 1995), an adaptive feed-forward wheel controller was added to the wheel servo to obtain a higher level of accuracy. The wheel controller and the path tracking controller have been integrated to obtain smooth motion. Model parameters were estimated using the least squares method.

Sliding Mode Control (Zhang, Xu et al., 2001): The nonlinear dynamic model of the robot with coupled inputs is obtained in state space representation using a nonlinear transformation to decouple the inputs, and a state space trajectory is used as a reference. The controller uses the state trajectory starting from any point of state space and converges asymptotically on sliding surfaces towards an equilibrium position.

Backstepping Control (Weiguo, Huitang et al., 1999): A path tracking controller is derived by applying the backstepping technique to the kinematic model in polar coordinates. The translational velocity is assumed constant, and the model reference is avoided using polar angle as parameter. Backstepping design has the merit of simplifying the design of nonlinear control and rendering it simple.

In Laumond (2001), three types of control problems are discussed: path following, trajectory stabilization, and stabilization of fixed configurations. In path following, the robot traces a curve at a constant velocity. A control technique using state feedback is proposed. In trajectory stabilization, the robot follows the reference curvature with time constraints and the translation velocity is not a constant. Control stabilizes the error in position and orientation to zero. In fixed configurations, the robot position and orientation are controlled in time.

1.4 Real Time Obstacle Avoidance

Mobile robots often need to navigate and execute their tasks in environments cluttered with obstacles. The robot must successfully reach a goal position without colliding with the obstacles encountered on its path, so as to prevent the robot and the objects encountered on its way from damage. Since no camera and laser range finder are installed on the robot, it will be assumed that the environment is flat or very smoothly inclined. In the case of a well known environment in which all obstacles are static, one of the algorithms mentioned in section 1.1 can be used to find an optimal trajectory that avoids all static obstacles.

These techniques are too costly to be used in real time applications in a dynamic environment where objects can be displaced and new obstacles can come into the scene. Consequently, a real time approach is usually needed so that the robot can avoid obstacles in real time, and reach the destination. These techniques often drive the robot in non optimal paths; however it is necessary to compromise optimality to improve safety.

The first research done on obstacle avoidance in real time was done by Khatib (1985). The concept of artificial potential fields is used to control the behavior of the robot, hence the term behavioral approach. This approach is efficient in calculation time and is independent of the geometric and kinematical transformations. The destination point in the space or plane is considered as an attractive pole and is dotted with a positive weight, while the obstacles are considered as repulsive poles and are dotted with negative weights. In this approach, the robot is prone to get stuck in local minima. This approach is also unstable when the destination point is close to an obstacle. The author's main interest at the time was in applying the technique on fixed base manipulators, but the concept can be applied to mobile robots as well.

An approach that makes use of artificial potential fields with simulated annealing (Lee, 2001) is used to plan local and global trajectories. This approach has been designed to avoid local minima and seems to produce interesting results. However the drawback of this approach is the calculation time required, which makes it too costly to be used in real time.

In another technique (Ju, 2002), obstacles are modeled as ellipses so as to reduce the complexity of calculation and detect collisions with moving obstacles. It has the advantage of generating an optimal trajectory while avoiding the obstacles as modeled. Still, the calculation time required renders it costly to be used in real time. The usage of a single ellipse to represent elongated obstacles may result in a suboptimal trajectory with respect to real obstacles.

Some other techniques using the same concept with some improvements have been developed. The Vector Field Histogram (VFH) technique (Borenstein, 1990) has the capacity to maintain a static representation of an obstacle at the level of the world model and the intermediate data level. This approach has the advantage that the robot can navigate in narrow corridors without oscillating and at high speeds. Still, this approach can lead the robot into an obstacle in certain cases, and it does not take into account that the robot moves in arcs and not in straight lines. The same authors (Ulrich, 1998) improved this approach and enhanced its reliability. The VFH+ takes into account the radius of the robot and leads the robot in smoother trajectories. This approach can lead the robot to local minima and thus to a dead end. To eliminate this effect, the VFH+ technique was used along with the A* algorithm and upgraded it to become the VFH* (Ulrich, 2000). This technique projects the trajectory of the robot on a few steps ahead and evaluates the consequences. It is also capable of finding solutions for the case when the robot should slow down and even stop. Although this method has been proved to be effective, it is more costly in terms of the calculations involved.

In the curvature velocity method (Simmons, 1996), an objective function in the curvature-velocity space is optimized in terms of speed of navigation, safety, and goal seeking. This technique can be used if the environment is unknown and no reference trajectory is available, and can be used for exploration purposes as well. This method will be used for this project.

Approaches based on the artificial potential fields act mainly as reactive systems and can be integrated with a global path planner for planning long trajectories in case the environment is partially known.

Supervised learning systems using neural networks, fuzzy logic, as well as neuro-fuzzy techniques (Fagg, Lotspeich et al., 1994; Beom and Cho, 1995; Mucientes, Iglesias et al., 2001; Macek, Petrovic et al., 2002; Xin, Vadakkepat et al., 2002) have been developed so that the robot can navigate autonomously while avoiding obstacles. Even if not all of the data corresponding to all possible single cases are supplied to the learning system, an intelligent system can adapt to individual cases of distribution of obstacles and perform adequately. Supervised learning can be either done online or offline.

1.5 Navigation Control Architectures

According to Driankov and Saffiotti (2001), navigation architectures that are used to control robot navigation can mainly be classified into two architectures: hierarchical and hybrid.

Exteroceptive sensors onboard the robot are used for the perception of the environment. The most common exteroceptive sensors used on wheeled mobile robots (WMR) are cameras, sonar arrays, and laser range finders. In contrast, proprioceptive sensors are used for the perception of the internal state of the robot, such as acceleration, velocity, and heading.

In hierarchical architectures (Figure 1) the data acquisitioned by exteroceptive sensors are used at the higher planning level, such as path planning. Consequently, planning is done in real time, and a more expensive processor would be required to perform the required computations since the whole model of the environment might be used.

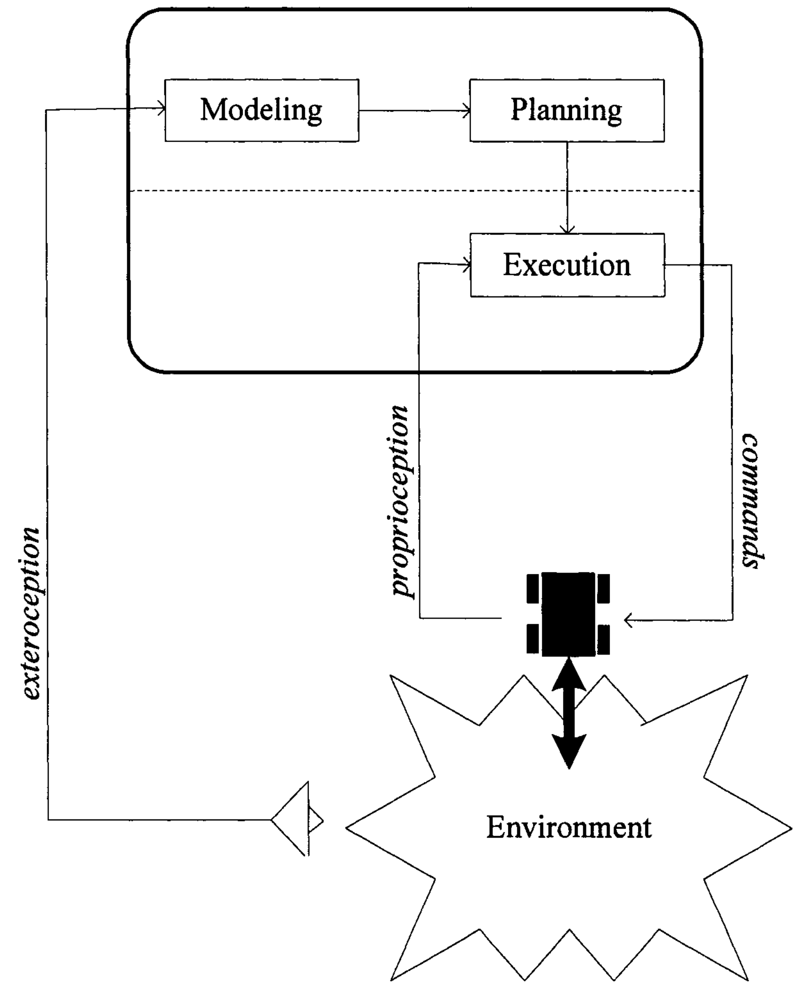


Figure 1 Hierarchical architecture: exteroception sensor readings are fed to the top of the control hierarchy

In the case of a hybrid architecture (Figure 2), data acquisitioned by exteroceptive sensors are used both the higher level functions and the lower level execution layer. This would necessarily imply the need to separate the task into behaviors (actions) and the need to coordinate behaviors according to their priority.

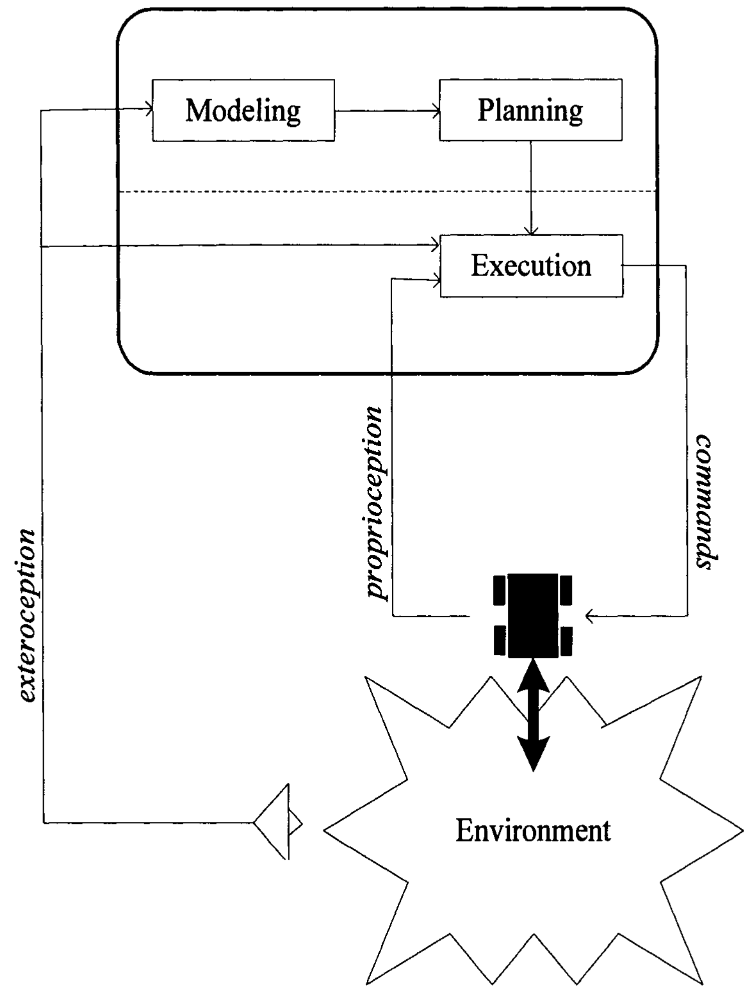


Figure 2 Hybrid architecture: exteroception sensor readings are fed to the top and execution levels of the control hierarchy

In this project, the proposed navigation architecture can be classified as hybrid and is implemented in the form of a state machine depending on sonar perception as input. In case no information about the environment is available, the robot would run in the curvature velocity mode, which is a combination of two different behaviors: goal seeking and obstacle avoidance.

If information about the environment is available, then the higher level planner finds an optimal path towards the user specified position if accessible from the current posture of the robot. Then, a lower level path tracker executes the trajectory. In case the sonar arrays perceive obstacles in the path of the robot, the running routine switches to the curvature velocity mode until the obstacle is bypassed and the waypoint behind the obstacle on the path is reached, in which case the robot switches back to the initial mode and continues executing the initially planned trajectory.

In Chapter 2, the technical specifications of the Pioneer 3AT robot as well as the client interface are detailed. The types of wheeled mobile robots and the modeling of four-wheeled skid-steered robots are discussed in Chapter 3. In chapter 4, a brief discussion of the A* algorithm is presented and the dynamic programming approach is developed. In Chapter 5, a fuzzy path following controller and a path tracking controller based on a Lyapunov approach are examined. In Chapter 6, the implementation of the curvature velocity method for obstacle avoidance is discussed. Finally, a proposed method for the integration of the three aspects of navigation in a complete navigation strategy is discussed in Chapter 7.

CHAPTER 2

TECHNICAL DESCRIPTION OF THE P3-AT MOBILE ROBOT AND ITS CLIENT INTERFACE

2.1 The Pioneer 3-AT Robot

The algorithms and control methodologies developed throughout this project have been tested on the Pioneer 3 All Terrain (P3-AT) mobile robot fitted with two sonar belts and a heading correction gyroscope in addition to the built-in 100 tick motor encoders.

Figure 3 The P3-AT robot in the lab

The P3-AT is a four-wheel drive robot based on skid-steer motion (see Chapter 3 on robot types and modeling). Its high power to weight ratio and its rigid aluminum platform render it robust on rugged outdoor terrain. Its relatively small size (50cm x 49cm x 26cm body and 21.5cm pneumatic wheels diameter and 9Kg with one battery) makes it suitable for indoor applications and navigation in narrow spaces. This also makes the P3-AT easy to handle, repair, and transport. The P3-AT robot can climb sills of 9cm or ramps having a slope of 45%. On flat floor, it can move at speeds of up to 1m/sec and it can rotate in place or it can move in a circle at a radius of 40cm with wheels moving on only one side. At slow speeds and on flat terrain, the P3-AT can carry loads of up to 20Kg. The robot is powered by up to three 12 Volts batteries that have an endurance of three to six hours, depending on charge and terrain. When only one battery is used, it is well recommended to put it in the middle slot of the battery compartment at the rear of the robot to maintain balance. When two batteries are used, one battery is placed in each of the left and right slots.

The four reversible DC motors are powered by a 12 Volts terminal from the batteries and are controlled by a MOSFET H-bridge, while there is a 5 Volts terminal to power the onboard electronics. The robot's power system is protected with fuses to protect all electric and electronic onboard components from eventual current surges. Figure 4 is an upper view of the motor-power interface PCB.

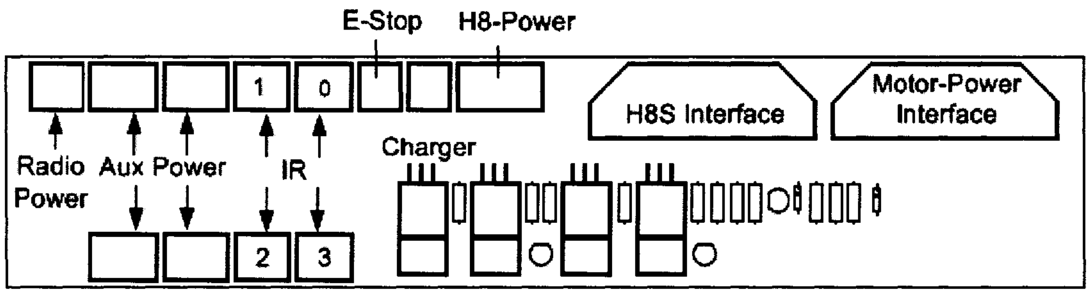


Figure 4   Power terminals on the P3-AT motor-power interface board

For more information on power connections, refer to Appendix B of the 'Operations Manual' (Robotics, 2001).

2.1.1 Wireless Serial Communications

The user control panel on top of the robot (Figure 5) is the user's hardware interface to control the robot. The original plate of the panel has been drilled to branch a 5 Volts power terminal outlet to the power board.

Since the robot used for this project has not been fitted with an onboard computer, a laptop computer fitted with a wireless serial connection has been used to control the robot.

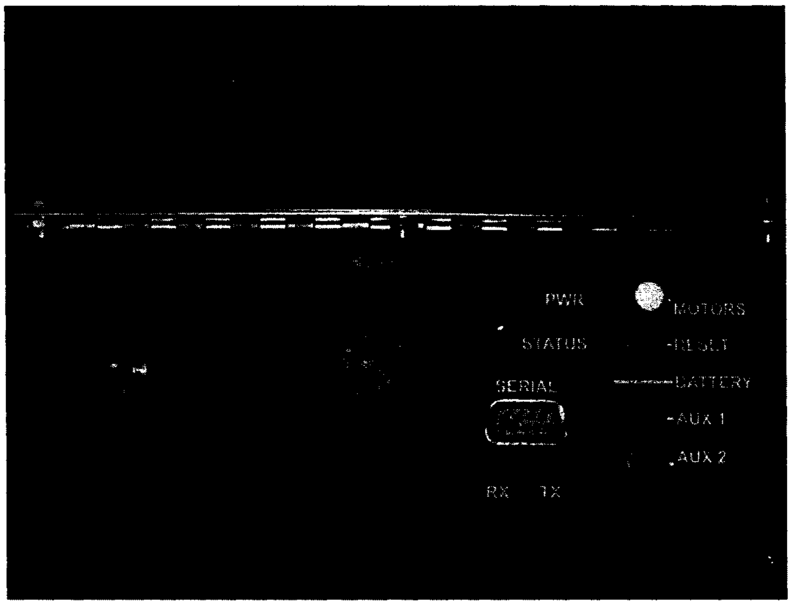


Figure 5 Close view of the user control panel

The wireless serial connection is simply used to replace a serial cable that is less convenient and restricts robot motion. Since the robot is used in a lab or at close proximity from the user, there is no need for an expensive industrial wireless serial connection with ranges of 500 meters. Consequently, it was decided to use a relatively cheap AIRcable Serial (Figure 6) which consists of two serial to serial wireless connector modules: the Data Terminal Equipment (DTE) module and the Data Communications Equipment (DCE) module with a reliable range of ten meters.

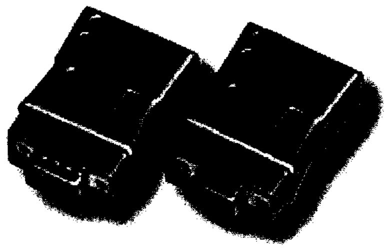


Figure 6 AIRcable Serial DCE and DTE modules

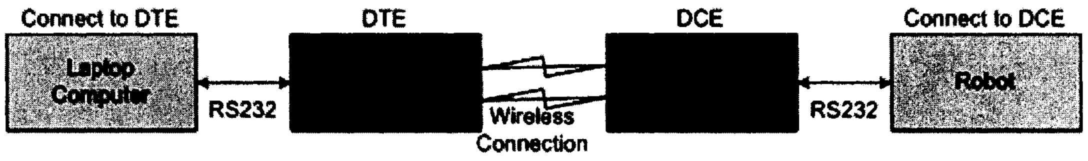


Figure 7 Wireless connection schematic

The DTE has a Sub-D 9 pin female connector that is connected to the laptop's COM1 serial port and the DCE has a Sub-D 9 pin male adapter that is connected to the robot (Figures 7 & 8).

Since the robot does not support hardware handshake, the two modules cannot make a connection. To remedy this problem, pins 7 and 8 of each of the two modules should be shorted. The Request To Send (RTS) and Clear To Request (CTS) pins are shorted (Figure 9). If the RTS signals a request, it is immediately cleared by the CTS without requiring a handshake between the computer and the robot.

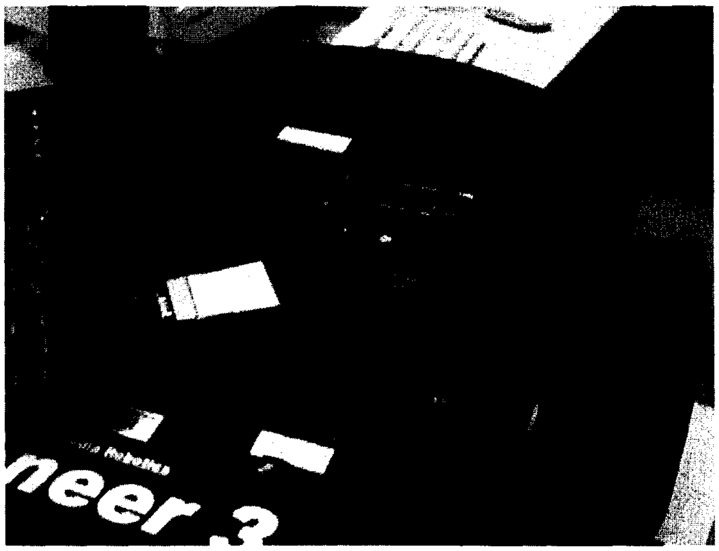


Figure 8 DTE and DCE modules plugged on the laptop computer and the robot respectively

Two DB9 connectors are used to short RTS and CTS for each module (Figure 10). The RTS and CTS of each of the two modules are shorted by internally soldering a wire to the two pins of the DB9 connector that will be plugged to the robot or laptop computer.

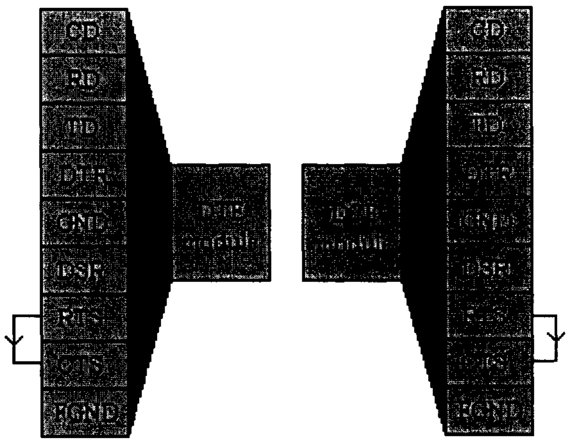


Figure 9 RTS and CTS pins shorted

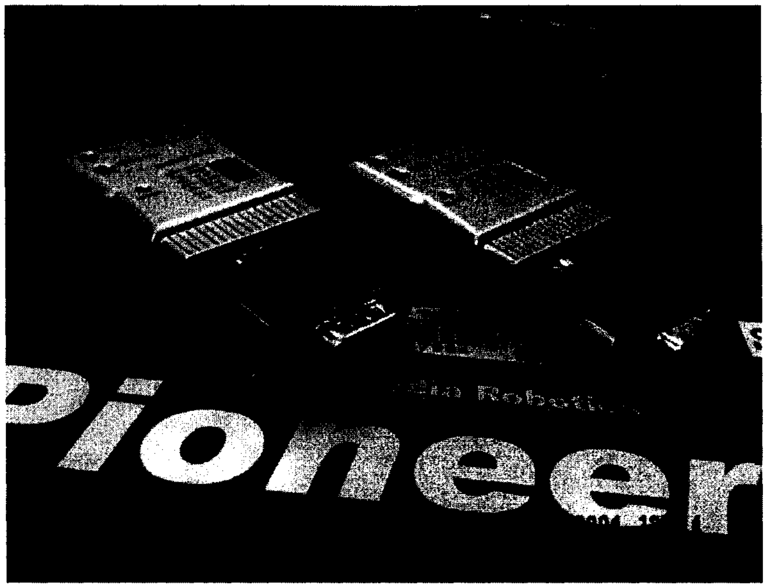


Figure 10 DTE and DCE modules with DB9 plugged into the intermediate DB9 connectors

2.1.2 Sonars

The P3-AT robot used in this project is fitted with two sonar arrays each having eight transducers mounted on the front and rear sections of the robot. Every sonar element returns a range to an object in a certain direction. Obstacles up to more than five meters from the robot can be detected. The sonars in our case are used to detect obstacles and navigate safely. The sonar arrangement in each array is shown in Figure 11.

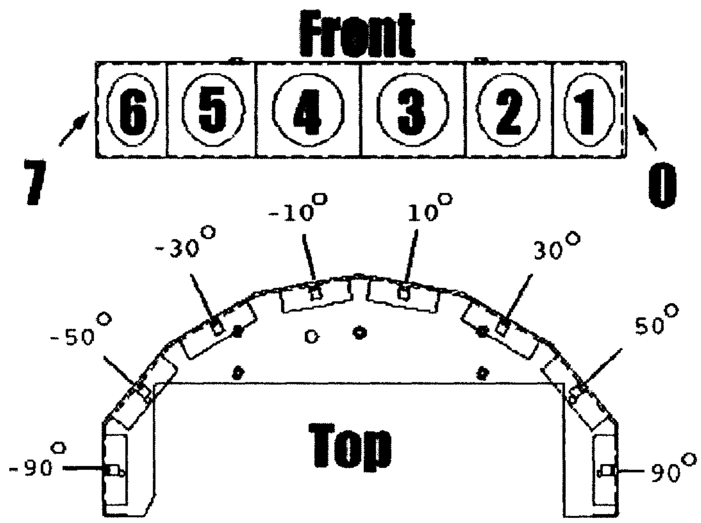


Figure 11 Sonar array front and top views (Courtesy of ActivMedia Robotics, LLC)

Sonars can be used to detect obstacles of reasonable cross section with a sufficient accuracy for navigation purposes. For pinpoint accuracy, a laser range finder can be fitted onboard the robot. It scans in a horizontal plane and can accurately determine the position of the object with respect to the robot. Due to budget limitations, the robot used in this project was not fitted with a laser range finder.

2.1.3 The Activmedia Robotics Operating System (AROS)

The AROS is the operating system that acts as the interface between a client application and the robot. In our case, the client application is an executable program on the laptop computer. The server is the microcontroller and its peripherals that executes the client application commands and provides it with sensor and robot information. The control architecture of the robot is shown in Figure 12. The client application sends packets to the robot server, containing motion commands and information requests on position and velocity as well as sonar readings and receives back the information from the robot server that executes the motion commands.

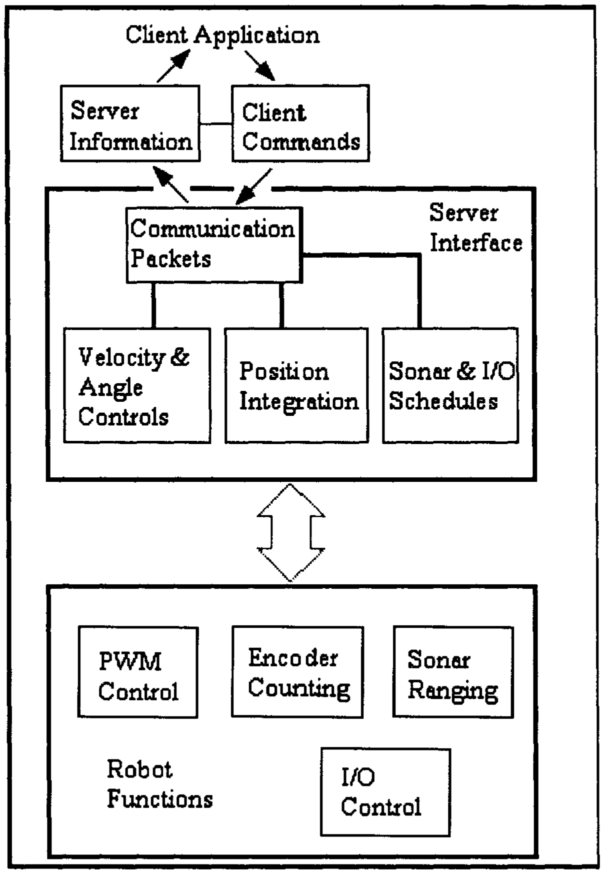
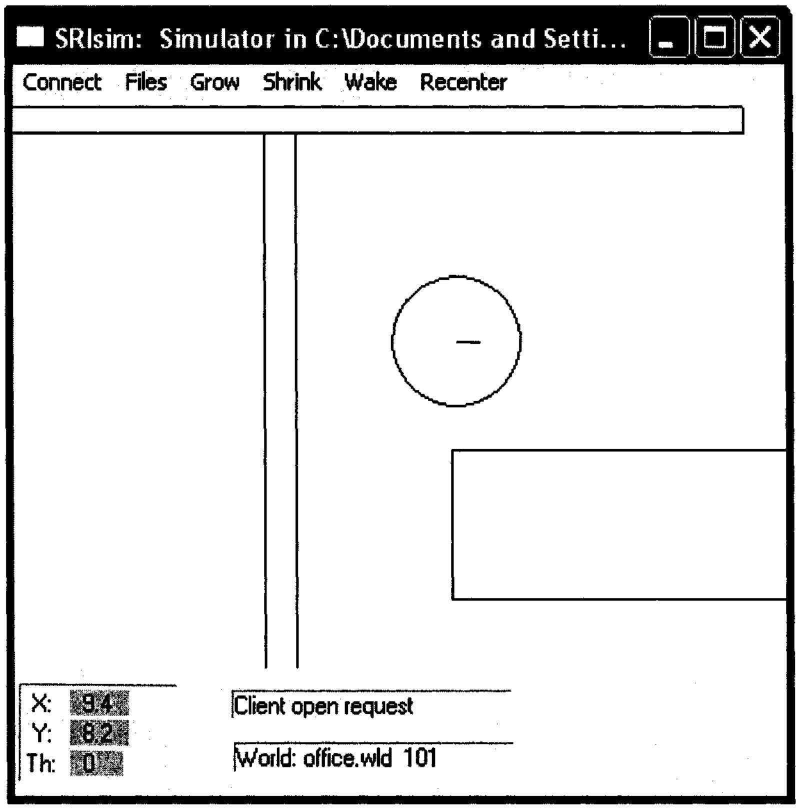


Figure 12 Client-server control architecture

The communication between the control client and the robot is done using special client-server communication packet protocols. In general there are two packet protocols, client command protocol and the server information packet protocol. The client commands and data requests are encapsulated in a packet using the former protocol, while the server information data is encapsulated in the latter. The packets consist of a stream of bytes that are sent through the wireless serial connection from one terminal and received and decoded at the other terminal to determine the type of information in the packet and its numerical arguments, if any. Fortunately, there is no need to deal with low level control and communications, since Activmedia has developed a reliable high level interface to control the robot. Thus the client program would make use of the ARIA library that will be discussed in the next section.

2.2 The SRI Simulator

Developed by the SRI International's Artificial Intelligence Center, the SRIsim is an excellent tool to test and debug client applications and specially those in the autonomous


SRIsim: Simulator in C:\Documents and Setti...
Connect Files Grow Shrink Wake Recenter
X: 9.4
Y: 8.2
Th: 0

Client open request

World: office.wld 101

Figure 13 SRI simulator with the P3-AT parameters loaded snapshot

mode before implementation and testing on the real robot. The SRIsim simulates real robot behavior in regards to odometer and sonar readings. The robot parameters and the environment can be loaded through the Files menu of the simulator window. The client application is connected to the simulator through TCP port 8081 (by default). A custom environment can be easily created using world files that can be loaded through the simulator menu (see Appendix 3 for a sample myWorld.wld file).

2.3 The Activmedia Robot Interface for Applications (ARIA)

To provide developers and researchers with a relatively easy to use interface, the ARIA open source object-oriented user interface was developed in the C++ programming language and is distributed with ActivMedia products. This library is used in the development of a client application as an interface to all the low-level tasks such as establishing communication with the robot. It is highly versatile and flexible and can be used to implement virtually any high level task without worrying about communication with the server and provides easy to use methods to access all sensor information and control the robot. The server on the robot is the AROS operating system that is originally installed on the robot's microcontroller. AROS manages all the low level tasks of motion and motor control and performs sensor information (encoders, gyroscope, and sonar) acquisition. Intelligent tasks such as navigation and sensor data fusion and interpretation are done at the level of the client application.

ARIA provides extensive methods and features for robot control and sensor data acquisition. The ARIA interface spares the user from developing all low level tasks such as packet encoding and decoding and supplies all needed interfaces to the robot. Only those methods and features used in this project will be discussed.

In this discussion, the C++ driver program in Appendix 1 that is based on the "*actionExample*" distributed with the ARIA package is explained. This example contains all the needed functionalities to operate the robot. There are other ways to control the robot; however using actions is the most suitable way for this project. The communication with the robot is done through a serial port as mentioned in the previous section. ARIA provides us with the methods necessary to establish the communication with the robot from the driver program without getting into the communications details. The most suitable method to communicate with the robot is done with the **ArSimpleConnector** method of the ARIA library. If the SRI simulator that is provided with the ARIA package is open, **ArSimpleConnector** will connect to the simulator, even if the robot is connected to the COM1 serial port. If no SRIsim window is open, it tries to connect to the robot through the COM1 serial port on the client computer. By default, **ArSimpleConnector** connects to the robot on COM1, however if another COM port is used, this has to be done by parsing the arguments through a command window. The statements in the code of the **main** function that parses the arguments to **ArSimpleConnector** are as follows:

```
ArSimpleConnector connector(&argc, argv);
Connector.parseargs( );
```

To change the settings as specified by the parsed arguments, the function **ArSimpleConnector::logOptions** is called:

```
If ( argc > 1 ){
        Connector.logOptions( );
        exit(1);
}
```

If the number of arguments *argc* is not greater than 1, meaning no arguments are specified other than the name of the executable, the default settings will be automatically initialized.

To connect to the robot **ArSimpleConnector::connectRobot** is used to establish the connection. This function takes as argument a pointer to an **ArRobot** object. **ArRobot** will be discussed shortly thereafter in what follows. If the robot was correctly associated with the pointer to the **ArRobot** object and a *true* was returned by **ArSimpleConnector::setupRobot**, a blocking connection is established with the robot, or if not a *false* is returned. The function **ArSimpleConnector::setupRobot** is called from **connectRobot**. This function first tries to establish a TCP connection on port 8101 with the simulator, if not it tries to connect to the device through the serial port and returns *true*. If none was available, a *false* is returned meaning no connection could be established. The code in the **main** function in Appendix 1 is:

```
if ( !connector.connectRobot(&robot)) {
        printf("Could not connect to robot ... exiting \n");
        Aria::shutdown( );
        return 1;
}
```

If a connection is established, execution of the main function proceeds, if not, a 1 is returned and program execution ends here.

After this brief discussion about establishing a connection with the robot, we turn now to discuss the **ArRobot** method that is the core of ARIA. As mentioned earlier, the AROS operating system software installed on the microcontroller manages low level tasks such as the execution of motion commands and the acquisition of sensor data. The robot microcontroller acts as a server to a client (user program such as the one in Appendix1), and thus ARIA is a high level interface between the client and the server. **ArRobot** acts as a gateway between client and server communications, is the central database for collection and distribution of state-reflection information coming from the robot, and is the systems synchronization manager (Robotics, 2003).

The client-server communications adhere to packet-based protocols. **ArRobot** handles the low level communications details such as encapsulating and sending data in packets known as client command packets as well as receiving and decapsulating server information packets and information extraction (see section of for more information about packets and packet contents). Some more explanation about how the packets are handled by **ArRobot** are available in Robotics (2003). For even more details, check the header and source files *ArRobot.h* and *ArRobot.cpp* that are available with the ARIA package. The standard server information packets (SIP) containing odometry and sensor information are sent by the server every 100 msecs. The tasks that **ArRobot** performs and are apparent in most client applications source codes are state-reflection as well as motion commands. State reflection consists of determining the state of the robot in what regards position, speed, heading, sensor readings, as well as other stuff related to the state of the robot. Motion commands are velocity and heading correction commands as well as limiting speeds. State-reflection functions such as **ArRobot::getPose**, **ArRobot::getX**, **ArRobot::getY**, **ArRobot::getTh**, **ArRobo::getSonarRange**, and a multitude of other functions are used by the client code to get needed information on the current state of the robot needed to issue motion commands or for other information requests.

At every time interval, **ArRobot** executes a series of interdependent tasks in the following order: SIP handling, sensor interpretation, action handling, and user tasks.

Almost all the other methods of ARIA act as peripherals for **ArRobot** of which **ArSimpleConnector** discussed above is an example. **ArSimpleConnector** uses **ArRobot::blockingConnect** to establish a blocking connection with the robot. This is why a pointer to the **ArRobot** object was needed in the **ArSimpleConnector**.

In the client program of Appendix 1, an **ArRobot** object is first created in the **main** function. Then the sonar range device is added to the robot using **ArRobot::addRangeDevice**. All range devices should be added before actions are added since some actions may require sonar readings to be executed, and thus will not be added to the **ArRobot** object (**robot** in our case) if no sonar device is added to it. If there is a gyroscope on the robot, it is automatically added to **robot**.

The motors are enabled by sending a special packet to the server through **ArRobot::comInt** that makes use of an **ArRobotPacketSender** object in **ArRobot**, that in turn calls packet encoders and sends the packet. Note that the arguments of **comInt** are **ArCommand::ENABLE** (string of type *unsigned char*) and 1 (of type *int*). If 0 was used instead of 1, the motors would be disabled. After some string decoding, **ENABLE** will be interpreted as a command (of type *enum* in **ArCommands**), and will be put in a client command packet and sent to the robot. So in **comInt**, **ArCommands::Enable** is passed to the **ArRobotPacketSender** object that takes care of sending the command to the robot, without the client programmer being concerned about any such details.

Now that the motors are enabled, the robot can be run using the **ArRobot::run** function. If the argument passed to it is *true*, this function calls a synchronous loop that executes until the client program is disconnected from the robot, at which point the function returns and continue into the **main** function. And after that the client program execution is closed.

The commands directly related to **ArRobot** in the **main** function are:

```
ArRobot robot;
robot.addRangeDevice ( &sonar );
robot.comInt ( ArCommands::ENABLE, 1 );
robot.addAction ( &recover,100);
robot.addAction ( &myAction,50);
robot.run ( true );
```

Figure 14 Robot commands

This is what concerns the usage of ArRobot in the client program of Appendix 1. **ArRobot** contains many other functions and utilities from which only the ones used in the action class are pointed out here.

Before explaining the code in the action class, we will briefly mention one more detail in the **main** function, the **ArSonarDevice** method. This method is inherited from **ArRangeDevice** and keeps track of sonar history and cumulative readings. An **ArSonarDevice** object is created and added to the robot in the **main** function. Using

ArRobot::getSonarRange, the distances to objects could be found without the user being concerned about any calculation details involving sonar readings. Other functions for sonar values interpretation are available in **ArRobot**.

The client application controls robot motion using one or more of the following: direct commands, motion commands, and through actions. Through direct commands, the client can send commands directly to the robot server from **ArRobot**. The list of possible commands is specified as of type *enum* in the **ArCommands** method. These are one byte commands with zero or more argument bytes associated with each of them. An example of sending direct commands is the **ArRobot::comInt** that we have used to enable the motors as indicated earlier.

Motion commands on the other hand are one level higher than direct commands in **ArRobot**. Motion commands are built-in motion commands that handle direct motion procedures, such as moving a certain distance from the current position in a straight line using **ArRobot::move**, or setting velocities and headings, or simply to stop the robot.

As stated in the ARIA user manual (Robotics, 2003), it is recommended to use actions instead of direct and motion commands to control robot motion. Actions are implemented in ARIA using the **ArAction** method. The user can define his actions by creating methods inherited from **ArAction** and overloading the **ArAction::fire** member function.

ArAction is very useful to create autonomous tasks in client applications. Some built-in functions are available in ARIA, of which only **ArActionStallRecover** is used in this project. The fire function returns a pointer to an **ArDesiredAction** object. For more information about how ArAction objects are interpreted and called by **ArRobot** and on how the actions are resolved according to their priority, you are kindly requested to refer to the ARIA user manual.

We now go back to the code in Appendix 1 to point out how **ArAction** was used. First, in the **main** function, the action objects are declared with the necessary parameters for the constructor in case needed. Then the actions are added to the robot using **ArRobot::addAction** that takes as argument a pointer to the action object to be added and a priority number. A higher priority number indicates a higher priority for the action to be added. This will be interpreted as such in the priority resolver. The **recover** action undertakes a series of actions to recover in case one of the wheels is stalled. It is given a higher priority then the other action added to **robot**. The **ActionGo** method is the client application action that carries out the task specified by the user. When **addAction** is executed, a pointer to the **ArRobot** object is passed to the action through the **ArAction::setRobot** function in case it was not overloaded in **ActionGo**, or else

```
ArActionStallRecover recover;
ActionGo myAction;
robot . addAction( &recover, 100);
robot . addAction( &myAction, 50);
```

Figure 15 Action declarations

ActionGo::setRobot will be called. In **ActionGo::fire**, member functions of other user specified methods can be called to perform the necessary calculations needed to obtain the arguments for the motion command. We will come back to this later on.

Let's now take a look at the ActionGo definition at the beginning of the client program of Appendix1. ActionGo is inherited from ArAction and the **fire** and **setRobot** functions are overloaded. in **ActionGo**. The user's class that performs the calculations of the desired velocities at the highest level of control and is used in the client program of Appendix 1 is **CurvVel**. A pointer to a **CurvVel** object is declared as a member variable to **ActionGo**. Other important member variables are **mySonar**, **myTime** to keep track of time, and **firstFire**.

An important member variable of the base method **ArAction** is **myRobot** which is a pointer to an **ArRobot** object. In the constructor, a **CurvVel** object is created, and **firstFire** is initialized to 1. To take into account the time delay between establishing the connection with the server and the first time **ActionGo::fire** is called and executed, **myTime** is set to 0 if **firstFire** is 1. Then **firstFire** is set to 0 and remains so all throughout program execution.

```
void ActionGo::setRobot (ArRobot* robot)
{
        myRobot = robot;
        mySonar = myRobot -> findRangeDevice ( "sonar");

        if ( sonar == NULL ) {
                deactivate( ); // function in base class
                return NULL;
        }
}
```

Figure 16 SetRobot member function

In function **ActionGo::setRobot** that will be called by **ArRobot::addAction**, the variable **ArAction::myRobot** is initialized to **robot**, the pointer argument of **ActionGo::setRobot**. Then the **ActionGo::mySonar** variable is initialized to the sonar range device associated with **robot** using **ArRobot::findRangeDevice**. Remember that this association was done in the **main** function. If no sonar range device is found, **ActionGo::mySonar** will be associated with a *NULL* pointer, and the action must be disactivated.

As mentioned previously, **ArAction::fire** returns the desired action to be sent to the server. The desired action must be reset each time **fire** is called. Then again we check for **mySonar** and if it is *NULL* the action must be deactivated and a *NULL* pointer would be returned and the function exits here.

Then state-reflection functions are used to get the parameters needed to determine the desired action. The angles returned are in degrees and distance parameters and sonar ranges in millimeters. The translational and rotational speeds are in millimeters per second and degrees per second. We can convert them to the units of our choice as we deem convenient. The values of the sixteen sonar ranges are obtained using **ArRobot::getSonarRange** and are stored in the **range** vector. All the needed information about the state of the robot is stored in a structure of type **dVInputs**, whose pointer will be passed to **CurvVel::determineVels**. The output of **determineVels** is a structure that consists of two variables, the translational and rotational speeds. These are converted to right and left velocities so as to use one motion command, **ArRobot::setVel2**, and thus avoid delays that would otherwise occur between two motion commands. Then the pointer to the desired action is returned. Function **fire** is called every time **ArRobot** performs the action handling task in the cycle. One more remark to add about the code in the **main** function is concerning the **Aria::init** and **Aria::shutdown**. These two functions should be added at the beginning of the driver function (the **main**) and just before the application ends. **Aria::init** initializes the thread layer and the signal handling method. **Aria::shutdown** shuts down all ARIA processes and/or threads.

2.4 Conclusion

In this chapter, the physical characteristics and the communication links to the P3-AT mobile robot as well as the ARIA application interface were described. An example application was discussed in detail to illustrate the operation of the ARIA interface. In the following chapter, the kinematic and dynamic models of a 4-wheel skid steer mobile robot are developed.

CHAPTER 3

TYPES AND MODELING OF DIFFERENTIALLY STEERED WHEELED MOBILE ROBOTS

3.1 Introduction

As mentioned in Chapter 2, the robot used in this project is the ActivMedia Pioneer 3AT mobile robot available at the GREPCI laboratary. In the first section of this chapter, a brief overview on the types of wheeled mobile robots (WMR) is presented, and is followed by a development of the kinematic and dynamic models of four-wheeled skid-steer WMR in the second section.

3.2 Types of Mobile Robots

A wide range of vehicles used for a wide variety of tasks can be classified under the category of mobile robots. Unmanned aerial vehicles, ground vehicles with various mechanical steering techniques, watercraft, submarine robots, and other types of unmanned vehicles whether remotely controlled, semi-autonomous, or fully autonomous all fall under the category of mobile robots. In this section, the discussion will be limited to WMR and their characteristics.

Holonomic or omni-directional mobile robots can move in any of the set of possible direction from its current posture without having to turn to that direction first. Non-holonomic mobile robots on the other hand can only drive in one direction from their current posture. For example, in a two-dimensional plane, a wheeled mobile robot can only move in the direction of the current orientation of the wheels. Non-holonomic mobile robots are limited by the kinematic constraints that restrict their motion. These constraints are often modeled using equations involving derivatives of the posture variables.

3.2.1 Ackermann Steered WMR

Non-holonomic WMR such as cars adopt the Ackermann steering mechanism and are usually four-wheeled, of which the two front wheels are passive and are used only for steering while the active rear wheels that supply the traction force provide needed to displace the vehicle are fixed and have a common axle. Three wheeled models often referred to as tricycles use the same mechanism but are steered by only one front wheel.

When the front wheels are fixed at a constant angle and the linear velocity is different from zero, the vehicle will follow a circle whose center is the intersection between the axles of the front and rear wheels (Figure 17). Thus, when a vehicle is steered, it will follow a path which is part of the circumference of its

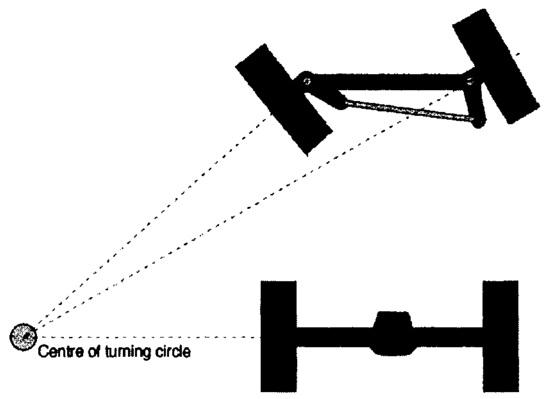


Figure 17 Ackermann steering mechanism

turning circle, that will have a center point somewhere along a line extending from the axis of the fixed axle. The steered wheels must be angled so that they are both at 90 degrees to a line drawn from the circle centre through the centre of the wheel. Since the wheel at the outside of the turn will trace a larger circle than the wheel on the inside, the wheels need to be set at different angles. The Ackermann steering geometry arranges this automatically by moving the steering pivot points inward so as to lie on a line drawn between the steering kingpins and the center of the rear axle. The steering pivot points are joined by a rigid but in length adjustable bar, the tie rod, which is also part of the steering mechanism. This arrangement ensures that at any angle of steering, the centre point of all of the circles traced by all wheels will lie at a common point.

3.2.2 Differentially Steered WMR

Differentially driven WMR are non-holonomic whose active wheels on the left and the right sides of the vehicle are driven by independent motors. Most of these robots have two active front wheels and one passive caster wheel in the back for stabilization purposes. These types of robots have the property that they can turn at the spot by applying equal and opposite forces on the wheels on each side, which makes them suitable in narrow environments cluttered with many obstacles and are usually used for indoor applications.

Another type used mainly in all terrain navigation has four fixed active wheels and is based on skid steering motion. Skid steering is accomplished by creating a differential velocity between the left and right wheels. Figure 18 is an upper view schematic of a four wheel drive robot skid-steering around the center.

The front and rear wheels on each side are synchronized so as to avoid longitudinal slippage. Although this type is usually slower than robots with Ackermann steering, they are more robust and maneuverable on rough terrains. The disadvantage is the control needed to make the robot move in a straight line since the angular speeds of each of the active wheels must be exactly the same.

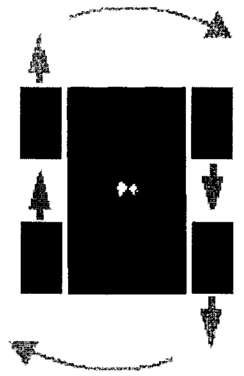


Figure 18 Skid-steering motion around the center

3.2.3 Single Wheel Drive WMR

This non-holonomic WMR is a tricycle but with the front wheel used for both steering and traction. The rear wheels are passive and fixed. This is considered to be the simplest design for a mobile robot. Linear and angular velocities of the robot are completely independent. For straight line motion (Figure 19 (a)), the front wheel axle is parallel to the rear wheel axle. To move in curvilinear motion (Figure 19 (b)), the front wheel is continuously angled depending on the curve to follow.

The robot can also spin around the center midway between the two rear wheels if the angle of the axle of the front wheel is orthogonal to the axle of the two rear wheels (Figure 19 (c)).

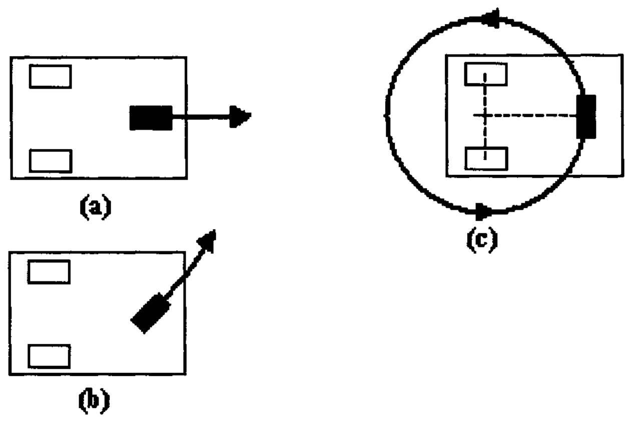


Figure 19 (a) Straight line motion (b) Curvilinear motion (c) Purely rotational motion around the center

3.2.4 Synchronous Drive WMR

The synchronously driven WMR is almost holonomic in that it can move in any direction. All the wheels are active and used for steering. The axles of all the wheels are always parallel to each other and the wheels turn at the same speed and in the same sense. One possible design is to have one motor used for steering chained to all the wheels and one motor for traction geared to all three wheels.

3.2.5 Other Types

Some other types of robots used for delicate missions with extreme conditions that require high mobility, possess some of the features from the different types. The Nomad2000 and the JPL explorer shown in Figure 20 have their left and right wheel drives independent and the steering of each of the wheels is independent. This would ensure maximum displacement capability in rough terrain.

Figure 20 (a) Carnegie Mellon Nomad2000 (Courtesy of Carnegie Mellon) (b) NASA Mars Rover for space exploration (Courtesy of NASA)

3.3 Modeling of Four-Wheeled Skid-Steered Mobile Robots

As mentioned in Chapter 2, the P3-AT mobile robot is a four wheeled skid-steered mobile robot. In this section, the kinematic and dynamic models on a two dimensional planar surface are derived.

3.3.1 Kinematic Modeling

The modeling of the kinematics of differentially steered wheeled mobile robots in a two-dimensional plane can be done in one of two ways: either by Cartesian or polar coordinates. The modeling in Cartesian coordinates is the most widely used and thus we will limit our discussion to modeling in Cartesian coordinates. The robot has four wheels and is differentially driven by skid steer motion. The motors that power the wheels at each side are geared internally to ensure that the velocity of the two adjacent wheels at each side are synchronized (having the same angular velocity) and thus the same velocity at ground contact.

The analysis of the motion of mobile robots is usually done for motion in a flat planar surface using two frames of reference, one local frame of reference fixed on the robot and the other one is a fixed reference from outside the robot as shown in Figure 21. Note that the x-axis of the local frame of reference is parallel to the direction of motion of the wheels and its origin is at the center of the rectangle formed by the centers of the four wheels. The transformation of the coordinates of a certain point P from the fixed frame of reference of the robot to the global frame of reference is given by:

$$\begin{bmatrix} {}^0P_x \\ {}^0P_y \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} {}^rP_x \\ {}^rP_y \end{bmatrix} + \begin{bmatrix} {}^0x_{ref} \\ {}^0y_{ref} \end{bmatrix} \tag{3.1}$$

$$\begin{bmatrix} {}^0P_x \\ {}^0P_y \end{bmatrix} = R(\theta) \begin{bmatrix} {}^rP_x \\ {}^rP_y \end{bmatrix} + \begin{bmatrix} {}^0x_{ref} \\ {}^0y_{ref} \end{bmatrix} \tag{3.2}$$

where θ is the angle between the x-axis of the global frame of reference x_0 and the x-axis of the robot frame of reference x_{ref}. It is positive if it goes in the anti-clockwise sense. ${}^rP_x$ and ${}^rP_y$ are the x and y coordinates of P in the frame of the robot, ${}^0x_{ref}$ and ${}^0y_{ref}$ are the coordinates of the origin of the robot's frame of reference with respect to the global frame of reference. The term $R(\theta)$ is the $2x2$ rotation matrix.

The change of position of the center of the robot (i.e. the origin of the local frame of reference) is characterized by $\dot{x}_{ref}$ and $\dot{y}_{ref}$ with respect to the local frame of reference and $\dot{x}_0$ and $\dot{y}_0$ with respect to the global frame of reference. The relationship between these four terms is as follows:

$$\begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \end{bmatrix} = \begin{bmatrix} \dot{x}_{ref}\cos\theta - \dot{y}_{ref}\sin\theta \\ \dot{x}_{ref}\sin\theta + \dot{y}_{ref}\cos\theta \end{bmatrix} = R(\theta) \begin{bmatrix} \dot{x}_{ref} \\ \dot{y}_{ref} \end{bmatrix} \tag{3.3}$$

The coordinates of the local frame of reference and the angle θ are sufficient to determine the posture of the robot with respect to the global frame of reference. The posture is defined by the $[{}^0x_{ref}\ {}^0y_{ref}\ \theta\ ]^T$ vector and is used to denote the instantaneous position and direction.

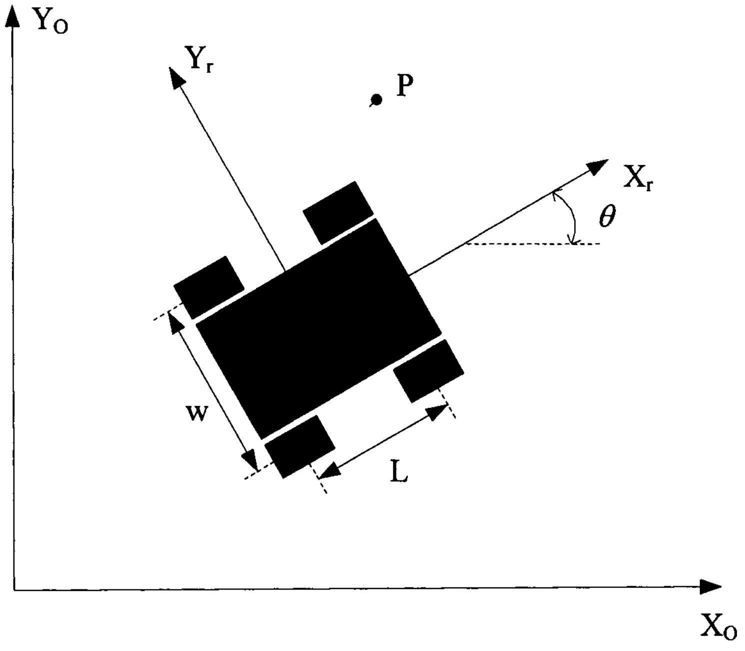


Figure 21 The two frames of reference used in the kinematic model

To get the accelerations, equation (3.3) is derived with respect to time:

$$\begin{bmatrix} \ddot{x}_0 \\ \ddot{y}_0 \end{bmatrix} = R(\theta)\begin{bmatrix} \ddot{x}_{ref} - \dot{y}_{ref}\dot{\theta} \\ \ddot{y}_{ref} + \dot{x}_{ref}\dot{\theta} \end{bmatrix} \qquad \textbf{(3.4)}$$

The motion of the wheels one to four (wheels 1 and 4 being on the left side and wheels 2 and 3 the wheels at the right side of the robot) is given by:

$$\begin{aligned}
\dot{x}_{r1} &= \dot{x}_{r4} = \dot{x}_{ref} - \frac{w}{2}\dot{\theta} \\
\dot{x}_{r2} &= \dot{x}_{r3} = \dot{x}_{ref} + \frac{w}{2}\dot{\theta} \\
\dot{y}_{r1} &= \dot{y}_{r4} = \dot{y}_{ref} + \frac{l}{2}\dot{\theta} \\
\dot{y}_{r2} &= \dot{y}_{r3} = \dot{y}_{ref} - \frac{l}{2}\dot{\theta}
\end{aligned} \tag{3.5}$$

If the rotational speed of each of the four wheels is constant, the robot would turn in a circle around a center whose coordinates with respect to the local frame are given by:

$$\begin{bmatrix} Cx_{ref} \\ Cy_{ref} \end{bmatrix} = \begin{bmatrix} -\dot{y}/\dot{\theta} \\ \dot{x}/\dot{\theta} \end{bmatrix} \tag{3.6}$$

If the speeds vary, as is usually the case, then these coordinates are referred to as the instantaneous center of rotation. In case there is no slippage, $\dot{y}_r$ would be zero and equation (3.3) becomes:

$$\begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \end{bmatrix} = \begin{bmatrix} \dot{x}_{ref} \cos\theta \\ \dot{x}_{ref} \sin\theta \end{bmatrix} = R(\theta)\begin{bmatrix} \dot{x}_{ref} \\ 0 \end{bmatrix} \tag{3.7}$$

In this case, the center of the robot would be always moving in the direction of x_{ref} and the velocity would be denoted by V. It can be proved geometrically that the rotational speed ω around the origin is equal to the derivative of θ with respect to time.

The velocities of each of the four wheels would be:

$$\begin{aligned}
\dot{x}_{r1} &= \dot{x}_{r4} = \dot{x}_{ref} - \frac{w}{2}\dot{\theta} \\
\dot{x}_{r2} &= \dot{x}_{r3} = \dot{x}_{ref} + \frac{w}{2}\dot{\theta} \\
\dot{y}_{r1} &= \dot{y}_{r4} = \frac{l}{2}\dot{\theta} \\
\dot{y}_{r2} &= \dot{y}_{r3} = -\frac{l}{2}\dot{\theta}
\end{aligned} \qquad \textbf{(3.8)}$$

And the most useful equation for controlling the robot at the kinematic level would thus be given by:

$$\begin{bmatrix} \dot{x}_0 \\ \dot{y}_0 \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} V\cos\theta \\ V\sin\theta \\ \omega \end{bmatrix} \qquad \textbf{(3.9)}$$

from which the following motion constraint is imposed:

$$\dot{y}\cos\theta - \dot{x}\sin\theta = 0 \qquad \textbf{(3.10)}$$

Usually, mobile robots have a minimal radius of turn which would impose another constraint on motion. If the angular velocity is zero, the robot will be moving in a straight line and the radius is infinite. Finally the instantaneous radius of rotation would be given by:

$$R = Cy_{ref} = \dot{x}/\dot{\theta} \qquad \textbf{(3.11)}$$

3.3.2 Dynamic Modeling

Through the use of the basic equations of mechanics, the dynamic model can be derived after drawing the free body diagram of forces. The dynamic model presented here is the same as in Caracciolo, de Luca et al. (1999), where the robot is a skid-steer four wheeled mobile robot and is similar to the P3-AT. The free body diagram of forces is presented in Figure 22. F_{xi} are the traction forces applied by the torques τ_i on each of the four wheels. The relation between the torques and the resulting traction forces is:

$$\tau_i = 2rF_{xi} (i = 1,2,3,4) \qquad \textbf{(3.12)}$$

where r is the radius of the wheels. It is to be noted also that the traction forces developed by two wheels on the same side are equal due to the internal gearing of the robot.

So the following relations would be valid:

$$\begin{aligned} \tau_1 &= \tau_4 \Rightarrow F_{x1} = F_{x4} \\ \tau_2 &= \tau_3 \Rightarrow F_{x2} = F_{x3} \end{aligned} \qquad \textbf{(3.13)}$$

The weight distributed on the four wheels is dependent on the position of the center of gravity G. For the analysis of the dynamic model, we take another frame on the robot with the origin at the center of gravity.

R_{xi} are the resistive forces on the wheels in the x_G longitudinal direction. The distribution of the weight on each of the four wheels is as follows:

$$\begin{aligned} F_{w1} &= F_{w2} = \frac{b}{a+b}\frac{mg}{2} \\ F_{w3} &= F_{w4} = \frac{a}{a+b}\frac{mg}{2} \end{aligned} \qquad \textbf{(3.14)}$$

where a and b are the distances shown in Figure 22.

The ratios of the weight of the robot carried by each of the two front wheels on the front are equal. The same applies to the rear wheels. If the coefficient of friction between the wheels and the ground is f_r, then each of the longitudinal resistive force would be given by:

$$R_{xi} = f_r F_{zi}\,\mathrm{sgn}(\dot{x}_i) \tag{3.15}$$

And the total resistive force in the x_G direction would be:

$$R_x = \sum_{i=1}^{4} R_{xi} = fr\frac{mg}{2}\left(\mathrm{sgn}(\dot{x}_1) + \mathrm{sgn}(\dot{x}_2)\right) \tag{3.16}$$

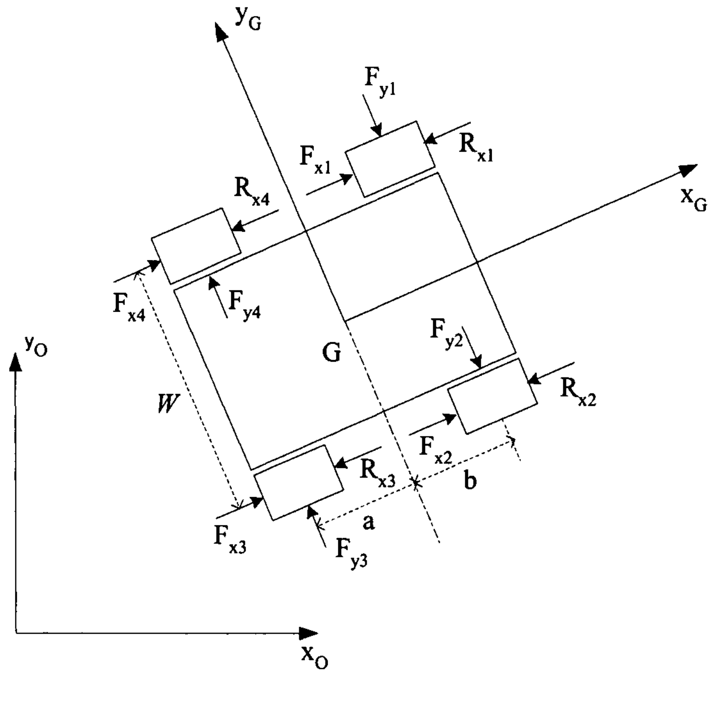


Figure 22 Free body diagram of forces

If the coefficient of friction in the lateral direction is λ, then the lateral friction forces on each of the four wheels is:

$$F_{yi} = \lambda F_{zi}\,\mathrm{sgn}(\dot{y}_i) \tag{3.17}$$

Then the total lateral friction forces would be:

$$F_y = \sum_{i=1}^{4} F_{yi} = \mu \frac{mg}{a+b}\left(b\,\mathrm{sgn}(\dot{y}_1) + a\,\mathrm{sgn}(\dot{y}_3)\right) \tag{3.18}$$

By applying Newton's second law to the forces acting in the x_G and y_G directions of the local frame, we get:

$$\begin{aligned} ma_x &= 2F_{x1} + 2F_{x2} - R_x \\ ma_y &= -F_y \\ I\ddot{\theta} &= w(F_{x1} - F_{x2}) - M_r \end{aligned} \tag{3.19}$$

where the resistive moment M_r around the center of gravity of the robot is calculated around the center of gravity of the robot:

$$\begin{aligned} M_r &= a\left(F_{y1} + F_{y2}\right) - b\left(F_{y3} + F_{y4}\right) + \frac{w}{2}\left[\left(R_{x2} + R_{x3}\right) - \left(R_{x1} + R_{x4}\right)\right] \\ &= \mu \frac{abmg}{a+b}\left(\mathrm{sgn}(\dot{y}_1) - \mathrm{sgn}(\dot{y}_3)\right) + fr\frac{wmg}{4}\left(\mathrm{sgn}(\dot{x}_2) - \mathrm{sgn}(\dot{x}_1)\right) \end{aligned} \tag{3.20}$$

The centrifugal forces acting on the robot are negligible since the robot rolls at relatively low speeds. Writing the dynamic model with respect to the global frame of reference, the following formula would be obtained:

$$M\ddot{q} + C(q,\dot{q}) = E(q)\tau \tag{3.21}$$

where $q = [x \quad y \quad \theta]^T$,

$$M = \begin{bmatrix} m & 0 & 0 \\ 0 & m & 0 \\ 0 & 0 & I \end{bmatrix}, \quad C(q,\dot{q}) = \begin{bmatrix} R_x \cos\theta - F_y \sin\theta \\ R_x \sin\theta + F_y \cos\theta \\ M_r \end{bmatrix},$$

$$E(q)=\begin{bmatrix}\cos\theta/r & \cos\theta/r\\ \sin\theta/r & \sin\theta/r\\ w/2r & w/2r\end{bmatrix} \text{ and } \tau_i = 2rF_{xi}(i = right, left)$$

Given that the wheels are synchronized, only the complete force supplied by two wheels on the right or on the left needs to be considered.

3.4 Conclusion

In this chapter, a brief overview of the different types of mobile robots has been presented, and the kinematic and dynamic models have been derived. In this project, only the kinematic model will be used for path tracking control in Chapter 5. In the following chapter, an optimization technique for path planning based on dynamic programming will be discussed at length.

CHAPTER 4

PATH PLANNING

4.1 Introduction

Robots often navigate in pre-known environments modeled using different tools such as maps. Making use of such information on the environment is advantageous. A path that is optimal in terms of factors such as time, distance, and safety can be planned to reach a desired goal position from the initial position of the robot. The available information can be mapped to a graph whose nodes represent some discrete positions in the environment and are linked by directional arcs with a cost value that is dependent on optimization parameter(s). In this chapter, the A* search algorithm is briefly presented and the iterative dynamic programming technique is described in full detail.

4.2 Cost Map Generation

As mentioned earlier, the path planning algorithm has been developed for outdoor and rough terrain navigation. To be able to find an optimal or near optimal path offline in this case, an aerial image of the terrain or previously stored information on the terrain on which the robot will be operating is needed. The approach mostly used in the literature is to sample the area into nodes at equal intervals in the form of a 2-D image. Given the scale of the image, a regional traversability map is obtained through some calculation procedure and is consequently converted to a cost map. The nodes are assigned a cost in a manner similar to a 2-D black and white image where each pixel corresponds to a certain gray level intensity value.

One simple and efficient method for path planning on natural terrains (Howard and Seraji 2001) makes use of fuzzy logic terrain-based path planning to find traversability, and a traversal cost function was applied to get the cost matrix.

Due to the unavailability of a camera or a laser range finder and a 3-D gyroscope, the cost matrix generation was applied on indoor flat surfaces where obstacles are given a very high cost while traversable regions are given a uniform cost.

4.3 Formulating the Problem

In the previous section the cost map that was obtained by analyzing an aerial image or a certain database of terrain information was obtained. Now the available cost map should be put in a form that can be solved by the optimization techniques that will be discussed later in this chapter. Given a certain position element of a certain node, it will be assumed that the robot can traverse from eight different directions as shown in Figure23. It will also be assumed that the cost to travel in a straight line between two opposite position elements adjacent to the position element in question by passing through it in any of the eight directions is the cost of the node itself. Nevertheless, optimization techniques can still be applied in the general case where the cost to go from a certain node A to an adjacent node B is different from the cost to go from B to A.

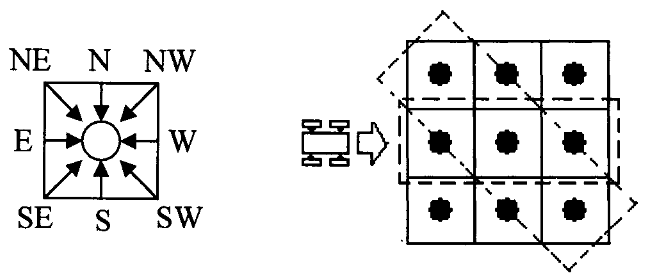


Figure 23 Eight passage directions through a node

The position elements will now be represented by nodes linked in two ways to all adjacent nodes (Figure 24(b)).

Each node will have eight links, one towards each of the adjacent nodes (Figure 24(a)). The nodes now are put in the form of a general graph.

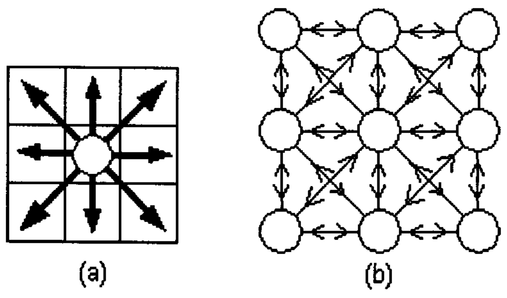


Figure 24 Graphical illustration of connections between nodes

4.4 The A* algorithm

The most commonly used algorithm for path finding in mobile robot applications is the A* algorithm. This algorithm has the advantage that it is highly efficient in calculation time compared to the other methods. The convergence to the optimal path has been proven in Hart, Nilsson, et al (1971); and its extensive use has proved it to be reliable if the proper estimation function referred to as the heuristic is used to approximate the cost to reach the goal. In the following discussion, N_i refers to the node i and the term L_{ij} refers to link j of node i. The terms and variables that will be used will be defined and the pseudo-code of the algorithm will be presented.

G: The cost to move from the starting point S to a certain given node by following the path generated to reach this node.

H: The estimated cost to move from that given node to the destination node. The actual cost from the given node to the destination node is not yet known. This heuristic is calculated by a function that can be dependent on the distance to the node point or some other parameter. The only difficulty in applying the A* algorithm is in the choice of a suitable heuristic function to determine H. A heuristic function is an estimate and generally cannot estimate the exact value of the variable in question (the cost H in this case). The choice of the heuristic function affects the speed of convergence and the probability of having an optimal solution. The choice of the heuristic function is application dependent.

C: Total cost to reach the destination from the starting node by passing through the current node. Since the actual cost from the current node in question is unknown, the estimate H will be used. So:
$C = G + H$

Open list: Current list of open nodes. This is the list of the nodes that have been opened but not yet processed.

Closed list: Current list of closed nodes. The nodes in *Closed* cannot be put in the *Open* list again. These nodes are the candidates to be on the final optimal path to reach the final destination node.

A pointer to the parent node is associated with each processed node. Initially the *Closed* and *Open* lists are both empty. The steps below are iterated over and over again until the destination node is put in the *Open* list (Pruski, 1996):

```
1. Put $N_0$ in Open and associate with it the cost $F(N_0) = H(N_0)$ and a NULL
   pointer, since $N_0$ is at the top of the hierarchy
2. If Open is empty, no solution and exit
3. Choose the node $N_i$ in Open that has the lowest cost F
4. Put $N_i$ in Closed
5. For all nodes $N_k$ that can be reached directly from Ni

        if      $N_k$ is in Closed

                if      $C(N_k)$ is lower than the cost of $N_k$ in Closed

                        Put Nk in Open and associate it with a pointer to Ni

                else    if      $N_k$ is in Open

                                if      $C(N_k)$ is lower than the cost of node $N_k$
        in Open

                                        Associate $N_k$ with the new cost and dot it
                                        with a pointer to $N_i$

                                else    Put $N_k$ in Open without modification

        if      $N_k$ is the destination node $N_f$

                Exit

        else

                Go to step 2
```

Figure 25 A* search algorithm pseudo code

The optimal path back from N_f is the inverse of the chained list going from the destination node to the start node N_0.

4.5 Dynamic Programming

4.5.1 Classical Dynamic Programming

The notation that will be used here is the same as in Hillier (1990). Given $N+1$ layers, including a layer for the final node, and node N is a node on the last layer before the final node. The decision variables x_n (n=1 ,2,...,N) are the immediate destination on the nth layer. The total cost of the best solution of the remaining layers (n, n+,...,N) towards the goal node is given by $f_n(s, x_n)$. For an arbitrary node s on layer $n+1$ the optimal solution from s to the goal by taking a decision x_{n^*} that minimizes $f_n(s, x_n)$ will be determined. The total cost is given by:

$$f_n^*(s) = \min f_n(s, x_n) = f_n(s, x_n^*) \qquad \textbf{(4.1)}$$

and

$$f_n(s, x_n) = Csx_n + f_{n+1}^*(x_n) \qquad \textbf{(4.2)}$$

where Csx_n is the cost between s and x_n and $f_{n+1}^*(x_n)$ is the minimum future cost (layer $n+1$ through $N+1$). We start first at layer N and continue through layers $N-1$, $N-2$, ...,1. The optimal costs for the nodes of layer N is $f_n(s, x_n) = Csx_n$ which is the cost of the single links between the nodes on layer N and the goal node.

The classical dynamic programming approach proceeds to find the optimal paths for all the nodes of all layers, including the node representing the actual state.

4.5.2 Dynamic Programming: A Novel Approach

When the cost matrix that corresponds to the map obtained by the aerial image is available, all inaccessible zones will be disregarded so as to minimize the number of nodes, and thus the number of calculations. This is done through a recursive procedure starting from the current position of the robot. Once the cost matrix of the terrain nodes is available, and given the initial or current position and the final or desired position, a modified version of dynamic programming is used to find the optimal path that leads to the node that corresponds to the desired destination position. Given that the node that corresponds to the final destination is *G*, all nodes will be put in layers starting from node *G* as follows: The first layer contains nodes that have a direct link to node *G*, the second layer contains the nodes that can reach *G* through a minimum of two links, the third layer contains the nodes that can reach *G* through a minimum of three links, and the *k*th layer contains the nodes that can be linked to *G* through a minimum of *k* links. By applying conventional dynamic programming on this graph, the optimality of the path cannot be guaranteed since it does not account for the links between nodes that belong to the same layer and for links going from lower to higher layers.

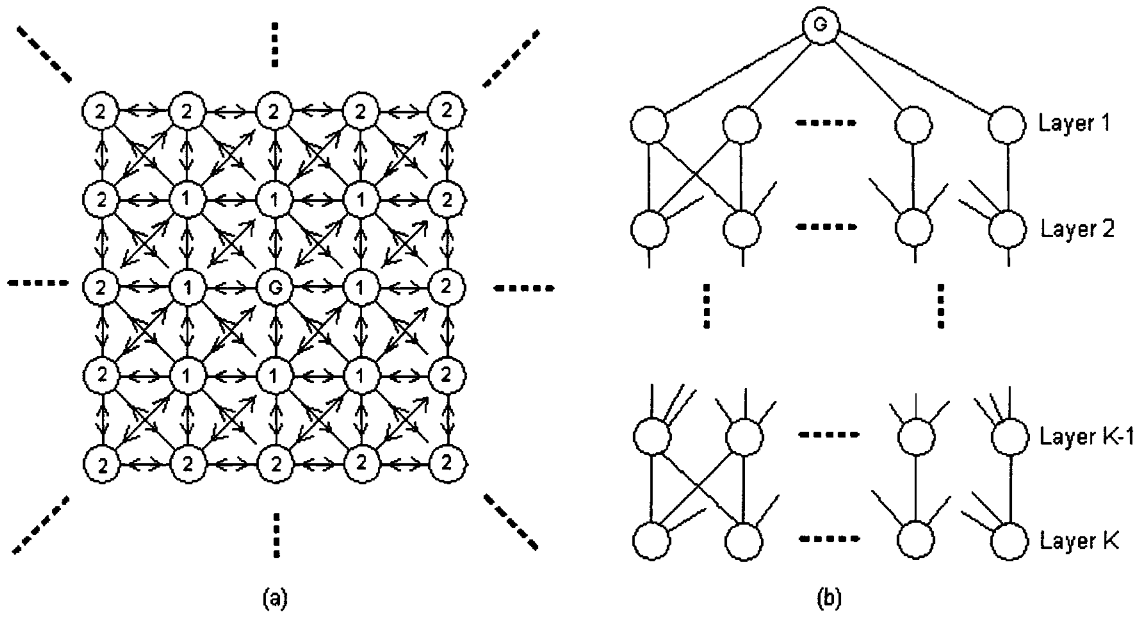


Figure 26 (a) Layers emanating from the goal node (b) Layers arranged from the goal node downwards

Nodes belonging to the same layer can only have links to nodes on the same layer, to nodes in the layer that immediately precedes it, and to nodes in the layer just after it (Figure 26).

In other words, if the node that corresponds to the initial position of the robot belongs to layer *I*, and the optimal path to the node that corresponds to the desired goal position passes from the initial position to another node on the same layer *I*, or goes through the higher layer *I+1* and back to *I* and towards the goal, then the classical dynamic programming approach cannot be used to find an optimal path that can go back and forth until it reaches the goal position.

The proposed approach is a generalization of the classical approach in that it makes it possible to have links between nodes belonging to the same layer, as well as links directed from an inferior to a superior layer. The classical approach will be used at the beginning to find a suboptimal solution and then iterations similar to the classical approach are applied to nodes on the same layer and superior layers to find a globally optimal solution.

In Figure 27, node S represents the initial position of the robot and node G represents the final goal position. In classical dynamic programming, the analysis starts from the goal node and upwards towards the higher layers, until the optimal path is found for all nodes including the node that corresponds to the initial position. The first layer contains nodes that have a direct link with node G.

There are five different layers in Figure 27. The start node S is on the first layer, however the optimal solution passes through the nodes 1 to 24 pointed to by the arrows.

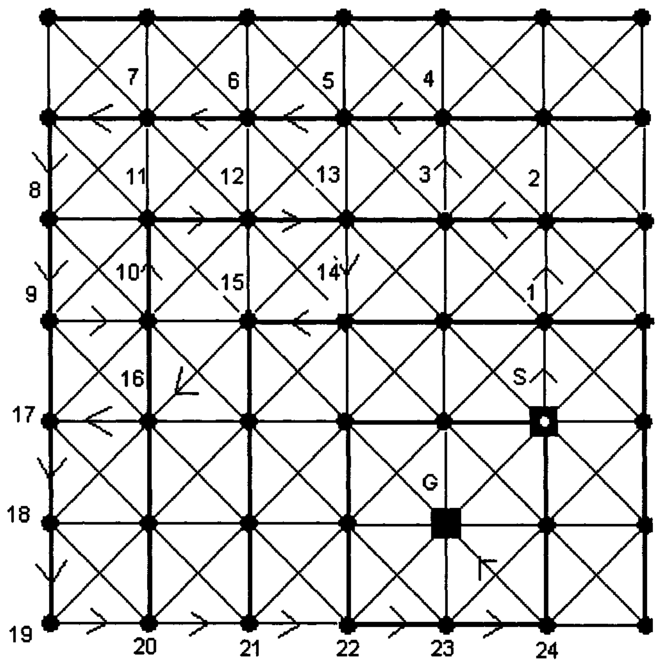


Figure 27 Graph with nodes

By applying classical dynamic programming, the optimal solution would be to go from S to G directly since it is the only solution available. The direct path from S to G might be very costly, and it might be more interesting to go backwards and then go towards G. As mentioned previously, the nodes on the same layer cannot have links between each other. In Figure 28, the problem is reshaped to be solved using dynamic programming. Clearly, the optimal path goes from S to node 1 on the second layer and then to node 2 on the second layer also, then to node 3 on the third layer, then to nodes 4 through 9 on the fourth layer, then goes forth to node 10 on layer 3, and so on. In this case, the fifth layer was discarded for clarity since the optimal path does not pass through it. However when the optimal path was calculated, it was tested and taken into account. The approach must guarantee a globally optimal solution.

A global optimal solution is required in the case of a labyrinth where only the globally optimal solution is acceptable, since all the other solutions pass through walls.

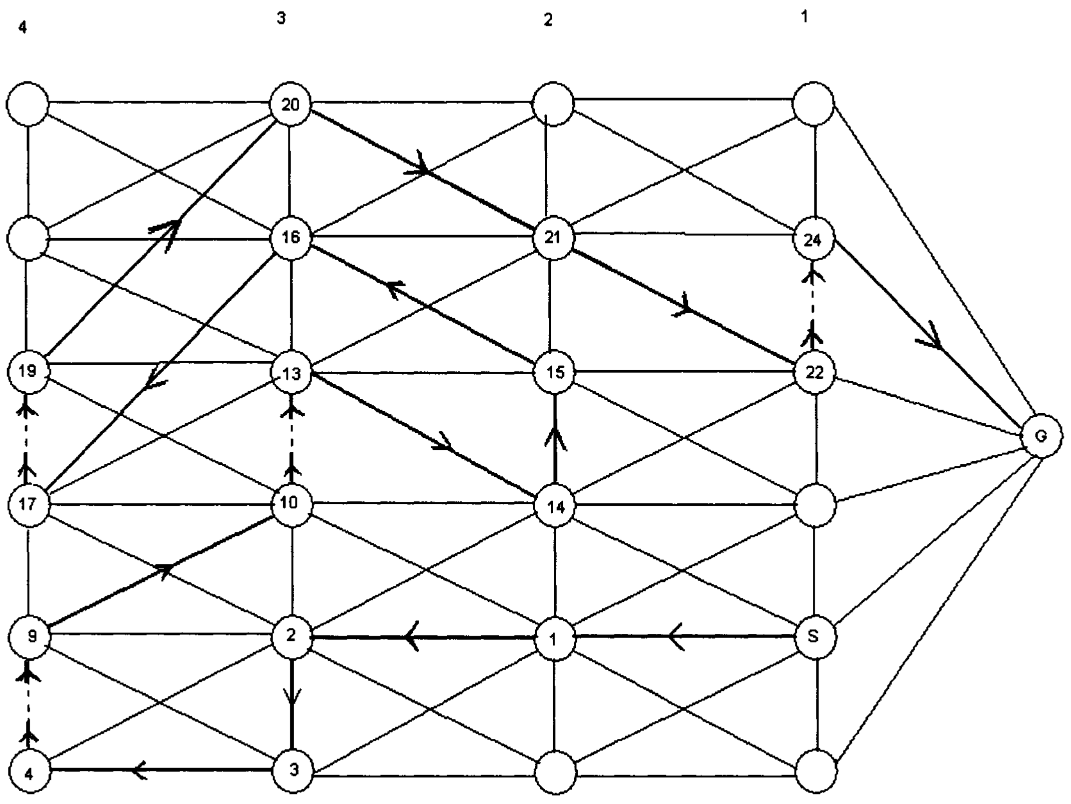


Figure 28 Nodes aligned in stages

Compared to classical dynamic programming in which the links between nodes are unidirectional and go from a superior layer to an inferior layer, and the nodes that belong to the same layer have no links between them, the new approach can take all the links and their sense into account. Moreover, the cost between adjacent nodes can depend on the direction of the link. That is, given two adjacent nodes *i* and *j*, the cost *Cij* to go from node *i* to node *j* can in general be different from the cost *Cji* from *j* to *i*. This flexibility is useful in the case of 3D terrain navigation. The cost to go up a slope is always different from the cost of descending it. The approach that is used to find a globally optimal solution starts by applying classical dynamic programming on the links allowable by this approach, that is from superior towards inferior layers only and to the goal position finally. Then links between nodes belonging to the same layer are tested to determine if a more optimal solution can be found. At the end of the process, links towards the superior layer are tested to determine if an even more optimal solution can be found. This process is repeated *(N+1)xM* times where *N* is the number of layers and M the number of nodes belonging to the layer that contains the maximum number of nodes.

4.5.2.1 Convergence to the Optimal Solution

In general, if there are *N* nodes other than the goal node, the maximum number of nodes that a path can pass through before reaching the goal node is *N*, or all the nodes other than the goal node, thus passing only once at each node. The goal is to find the absolutely optimal procedure to reach the goal node among all possibilities. To this end, the layered solution graph will be introduced. To simplify the graph, the connections of the nodes from a certain layer *i* to a layer *j* are represented by a vector link.

The vector link in Figure 29(b) represents only links going from nodes on layer *i* to layer *j* in one direction from *i* to *j*. Nodes on a certain layer can be either connected to other nodes in the same layer, to nodes in the next higher layer and to nodes at an immediately inferior layer. So the difference between *i* and *j* can only be 1, 0, or -1. To represent the links between nodes on the same layer, the nodes are duplicated as in Figure 30 (a) and represented by a vector link from *i* to *i* (Figure 30(b)).

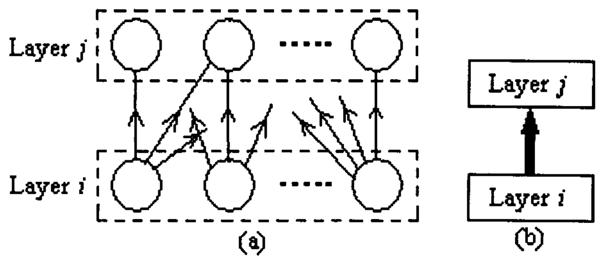


Figure 29 (a) Links going from nodes on layer *i* to layer *j* (b) are represented by a vector link from i to j

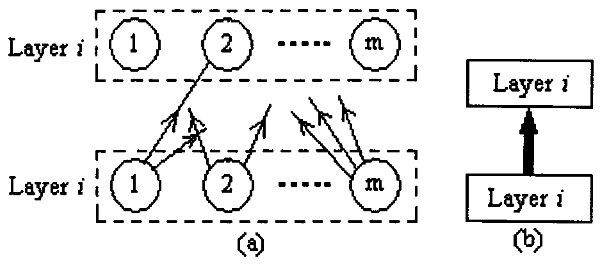


Figure 30 Inter layer connections

At layer 1, all the nodes are connected to the goal node, and these links are represented by a vector link from layer 1 to the goal node *G* as in Figure 31 below.

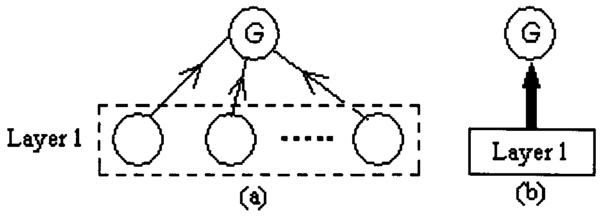


Figure 31 Connections to the final goal layer

Now that some simplifications are made, the layered solution graph can be presented (Figure 32). On top is the goal node with layer 1 directly below it and connected to it with a vector link. The links from the goal node to nodes of layer 1 will not be used since the objective is only to reach the goal node. In the next step, layers 1 and 2 are inserted below layer 1 and also linked to layer 1. This should not cause any confusion. One can imagine layer 1 on level 2 as having different nodes from layer 1 on level 1 but having the same connections and costs. At the *K*th level, all the layers would be contained. Levels *K+1* to *N-1* contain all the layers in order with each layer having vector links with maximum three layers (layers 2 to *K-1* and two vector links for layers 1 and *K*) at the level above it as in Figure 32. More specifically, a certain layer *k* at level *n* is connected to its duplicate at level *n-1* and also to layers *k-1* and *k+1* at level *n-1* if they exist. The layered solution graph contains all the possible solutions (paths) to reach the goal node from any other node. Classical dynamic programming can now be applied to those virtual layers stuffed in levels to find an optimal solution.

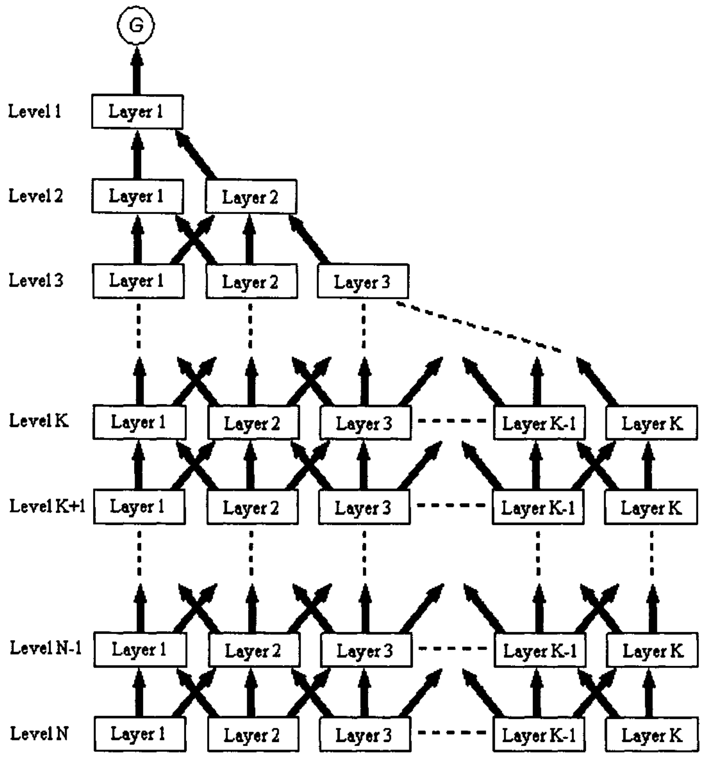


Figure 32 Layered solution graph

Let the operation of DP (i, j) denote the operation on all nodes of layer i of equations (4.1) and (4.2) with layer j being the layer that contains the next destination nodes. The first operation would be DP (1, G) at level 1 to the goal node G. The next step would be DP (1, 1) from level 2 to level 1, followed by DP (2,1) also on level 2, and so forth until the whole graph has been analyzed. The number of DP operations is equal to the number of vector links V on a layered solution graph and is given by:

$$\begin{aligned} V &= (N-K-1)(3K-2)+\sum_{k=0}^{K-1}(2+3k)+1 \\ &= (N-K-1)(3K-2)+2K+3K(K-1)/2+1 \qquad \textbf{(4.3)} \\ &= 3NK-2N-3K^2+K+3K(K-1)/2+3 \end{aligned}$$

The optimal solution from a certain starting node to the goal node would be the minimum of the cost between all its virtual duplicates at the levels 1 through N. It is guaranteed that at the optimal solution, the path cannot pass twice by the same node since all the costs of going from one node to another are positive and the minimum will not pass through the same node twice. In other words, if L_{ij} is used to denote level i and layer j, and some node n belongs to layer j, then the cost C^* of the optimal solution to go from n to G would be given by:

$$C^* = \min(n_i \in Lij)\Big|_{i=1,\ldots,k} \qquad \textbf{(4.4)}$$

where n_i is the duplicate of node n at level i.

4.5.2.2 Iterative Procedure

Putting all these layers in a layered solution graph is very costly in terms of memory space. But since the layers at the different levels are all duplicates, if each node was associated with a current optimal cost C^+ to reach G and a pointer p to the next node to follow that lie on the current optimal path towards G, with DP (i, j) being executed in the same order as in the layered solution graph. After every DP (i, j) operation, only the costs needs to be determined

So only the graph of Figure 26 (b) rearranged in layer-vector link representation in Figure 33 is needed and an iterative procedure would be followed to determine the optimal solution.

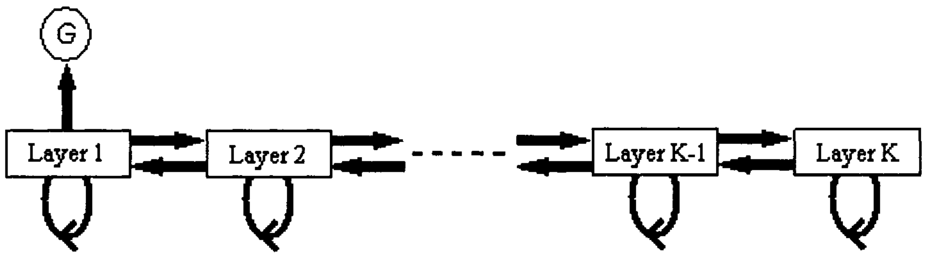


Figure 33 Layer-vector link representation of the graph

The pseudo code to perform all the *V* operations needed to reach the optimal solution is presented in Figure 34.

```
DP(1,G)
for i = 1 to K
        for j = 1 to i
                if j == 1
                        DP (j , j)
                        if i ≠ 1
                                DP( j , j+1 )
                        end
                else if j == i – 1 && j ≠ 1
                        DP (j , j-1 )
                        DP (j , j)
                else if j == i
                        DP (j , j-1 )
                else
                        DP (j , j-1 )
                        DP (j , j)
                        DP (j , j+1 )
                end
        end
end

for i =1 to N-K
```

```
        for $j$ = 1 to $K$
            DP ($j$, $j$)
            if $j \neq 1$
                DP ($j$, $j$-1)
            end
            if $j \neq K$
                DP ($j$, $j$+1)
            end
        end
    end
```

Figure 34 Pseudo code of the iterative procedure

4.6 Implementation on Parallel Processors

Going back to the solution graph (Figure 32) and supposing that the DP operations have reached a certain level L_i and the DP operations (DP (j, j-1), DP (j, j), DP (j, j+1)) are to be done on a certain layer j at L_i. For these three DP operations on layer j corresponding to L_i to be executed, all the DP operations for all levels L_k $(k \in [1, i-2])$ and for all layers between $[\max(1, j-(i-k)), \min(j+(i-k), K)]$ must have been executed.

Conversely, if at a certain level L_i, all the DP operations have been done till a certain layer j, then all the DP operations for all levels L_k from L_{i+1} up to L_{i+j-1} for layers $[1, j\text{-}k]$ in these levels can be executed independently of the rest of the DP operations corresponding to the rest of the solution graph.

Furthermore, if all the DP operations till a certain level L_i have been executed, then all the DP operations for all the layers in level L_{i+1} can be executed independently from each other.

These properties of dynamic programming make its implementation on parallel processors attractive. The parallel processor can be exploited to the maximum with this algorithm.

4.7 Implementation on MATLAB

Since the new dynamic programming algorithm was developed before the robot was available, the MATLAB technical computing software was used to implement and test the algorithm. Although less flexible and powerful than the C++ programming language in terms of speed and performance, the ease of implementation of the algorithm in MATLAB made it the right choice. Since most MATLAB operations and data are represented as matrices, all the data structures have been represented as matrices.

The program (script) where the cost maps in matrix forms are defined and the plotting is done is the 'main.m' file. The user is prompted to select one of six maps, and to select the start and goal positions. The function **'getOptimal'** takes **'startPosition'**, **'goalPosition'**, the **'MAPCOSTS'** matrix, and **'scale'** as input arguments and returns **'WayPoints'** as well as **'OPTIMAL'**. Let's start first by the input arguments:

> **'startPosition'** : *1x2* input vector containing the *x* and *y* coordinates of the initial position of the robot in meters. The origin is at the lower left corner of the map.
> **'goalPosition'** : same as **'startPosition'** but with the goal coordinates.
> **'MAPCOSTS'**: *MxN* matrix of costs as explained in section 4.1.
> **'scale'**: Is the horizontal or vertical distance in meters between two adjacent nodes in **'MAPCOSTS'**. In all the tests, the scale was assumed to be *1* meter.

And the output arguments:

> **WayPoints**: *Wx2* matrix containing the coordinates of the nodes that lie on the optimal path from **'startPosition'** to **'goalPosition'** inclusive, and in order.

OPTIMAL: *Nx3* matrix whose first column contains the number of all accessible nodes on the map from *1* to *N* and whose second column contains the next node towards the target goal for all nodes of the first column respectively. The third column contains the cost of going from the node in the first column towards the goal node. The goal node is the only node having a cost of zero in the third column.

The rest of the code in **'main.m'** is for plotting the graph of the map with the obstacles and the optimal path from the start position to the goal position.

The function **'getOptimal'** is the interface to use the dynamic programming technique that has been developed to obtain the optimal solution. This function and its sub functions will be now briefly discussed. The first step in **'getOptimal'** is to get the **MAPCOSTS** matrix indices corresponding to the start and goal positions. Then the function '**determineNodes**' finds all accessible positions from the start position, assigns to them a node number, and determines the connections between nodes. The goal position should be chosen in an accessible region. The **OPTIMAL** matrix is calculated using the function '**dynamicProg**'. At the end, the **WayPoints** matrix is calculated from **OPTIMAL**.

The function '**determineNodes**' takes as arguments **MAPCOSTS** and **startPositionIndices** determined in '**getOptimal**'. In this function, two matrices are calculated: **NODES** and **MAPINDEXROW**. **NODES** is the matrix that contains the nodes and the nodes they are connected to along with the costs. **MAPINDEXROW** is the *Nx3* matrix containing the node number as well as their corresponding index in **MAPCOSTS**. The recursive function **findAccessibleRegions** is used to calculate these two matrices. Inaccessible nodes are disregarded.
In the function **'getOptimal'**, the function **'dynamicProg'** calculates the matrix **OPTIMAL** and is the bulk of the program. This function takes as input arguments the

following variables that have already been calculated: **NODES**, **startNode**, **targetNode**, **MAPCOSTS**, and **MAPINDEXROW**. Let's take a look inside the function **'dynamicProg'**.

Starting from **targetNode**, all the nodes are classified and ordered into layers as explained. The function **'findStages'** uses a sorting routine to return the matrix **stages** whose rows contain the nodes of each of the layers. The number of columns in stages is determined by the number of nodes in the layer that contains the maximum number of nodes. The rows are filled starting from the beginning, and the remaining unused slots are filled by zeros. Although this is not efficient in terms of memory usage, no better performance can be achieved using MATLAB, since MATLAB offers no pointer usage as in C or C++.

The three dimensional matrix **NODESLowMedUp** is calculated using **NODES** and **stages**. **NODESLowMedUp** orders the connections to nodes in **NODES** into connections to the next lower, the same, and the next upper layer for all nodes. The pseudo code in Figure 34 is then implemented using the function **'DP'**. Finally the matrix **OPTIMAL** is returned to **'getOptimal'**, where the matrix **WayPoints** is calculated and returned to **'main'** along with **OPTIMAL**. Figure 35 is a hierarchical diagram of the functions used to implement the dynamic programming algorithm.

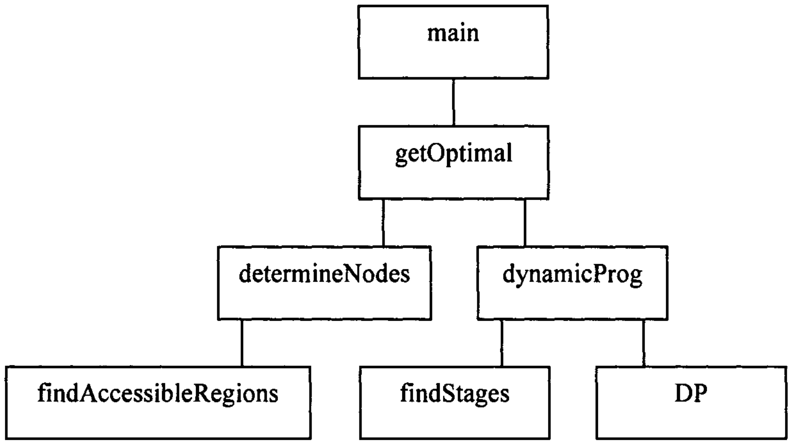


Figure 35 Hierarchy of functions

4.8 A* and Dynamic Programming: A Comparison

Both the A* algorithm and the dynamic programming technique presented in this chapter yield optimal solutions to a graph. The advantages and disadvantages of the dynamic programming technique relative to the A* algorithm can be summarized as follows:

Advantages:

- The dynamic programming algorithm determines the optimal solutions for all the nodes of the graph towards the goal node, which is advantageous in a multi-robot environment or in case the robot deviates from its original path due to a real time dynamic obstacle.

- The dynamic programming algorithm can be executed on several computational resources in parallel while the A* algorithm can only be executed sequentially.

- The dynamic programming algorithm is generic and its performance is not dependent on the choice of any heuristic function as is the case with the A* algorithm.

Disadvantages:

- The only disadvantage of the dynamic programming algorithm with respect to the A* is in the calculation time which is significantly much higher if implemented sequentially on a single computational unit.

4.9 Performance and Results

The dynamic programming algorithm has been tested on numerous cases which were compared to the solutions obtained using the A* algorithm. The solutions were exactly identical using both techniques and the graphic results presented here were obtained using both techniques. In Figure 36, the environment is a 2-D environment where obstacles are represented by grey cases and free space by the white cases. The cases are essentially the nodes of the graph. It was assumed that the cost of displacement in free space from one case to another adjacent case horizontally or vertically is 1 and that the cost of moving in diagonals is 1.41. It is important to note that displacement is limited to eight directions: it can be either horizontal, vertical or along diagonals at slopes of 45 degrees. This implies that displacement is valid only between adjacent cases. The cost is thus proportional to the distance traveled. The start position and the desired destination are represented by an 'x'. The optimal path was obtained in both cases.

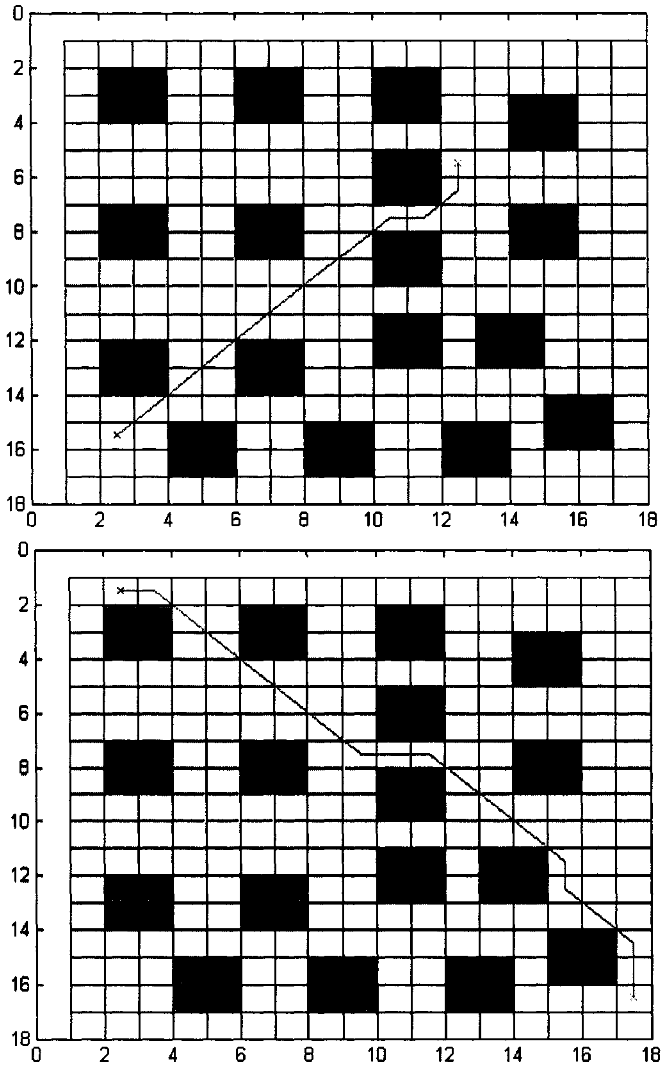


Figure 36 Two test cases in a 2-D environment

In Figure 37, the cost between the nodes is variable and not only dependent on distance, as is the case in a 3-D terrain. All nodes in this environment are accessible. The cost of the displacement from one node to an adjacent node was chosen to be the value assigned to the adjacent node. This allows for testing the dynamic programming algorithm in a general graph where the costs of displacement between two nodes in the opposite senses are not necessarily the same. The white cases were given a cost value of 1. The other cases were given values of 3, 10, 15, and 40 from lightest to darkest. To better visualize the solution, the environment was created such that the optimal solution is purely on the white cases. The concentric rectangles emanating from the goal node join nodes that belong to the same layer. The node that represents the initial position of the robot lies on the fourth layer. It is clear that the optimal solution passes through nodes on layers higher than the initial node and by nodes that lie on the same layer.

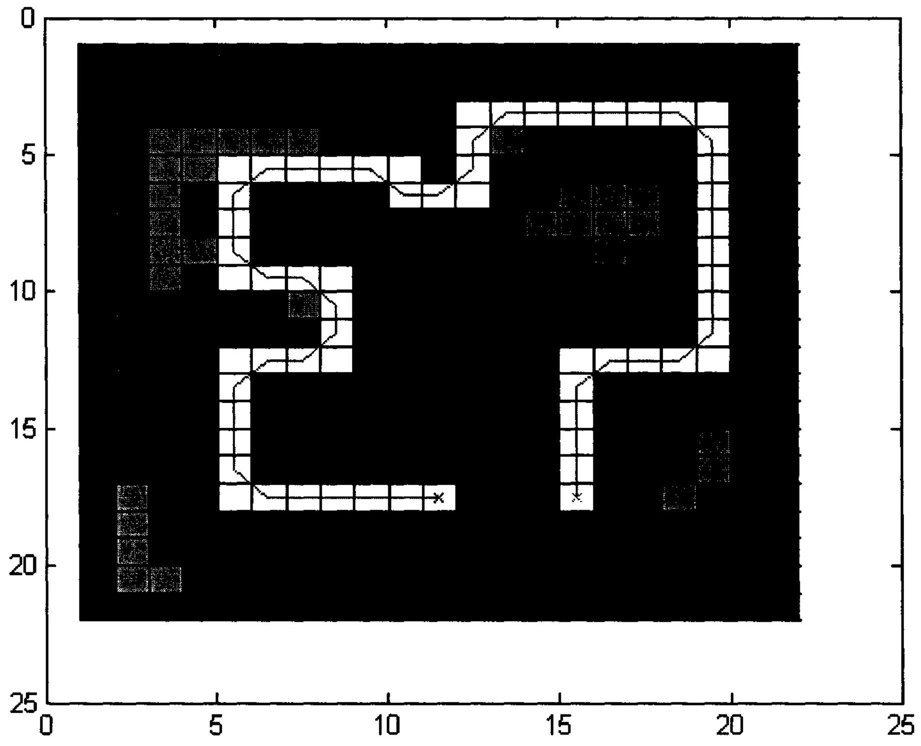


Figure 37 Optimal solution in a 3-D environment

Figure 38 is the result of another case in a general 3-D environment. All the nodes are accessible and the cost ranges from 1 to 40 from lightest to darkest cases. With some inspection, it is obvious that the solution is optimal. In Figure 39, the dynamic programming technique was used to find the solution to a labyrinth. The graphical results displayed in this section demonstrate the effectiveness of the dynamic programming algorithm in finding an optimal solution to a graph.

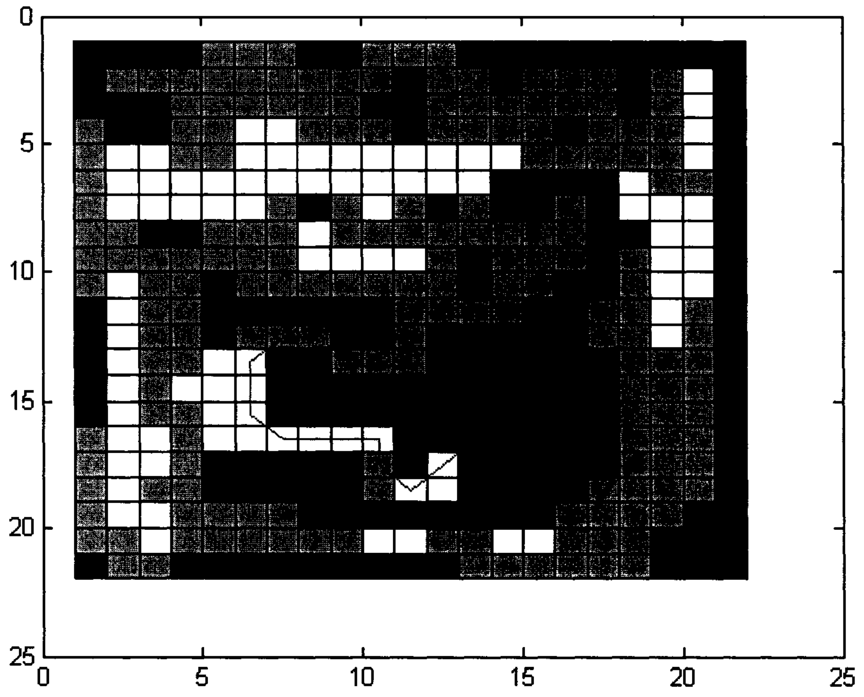


Figure 38 3-D environment test case

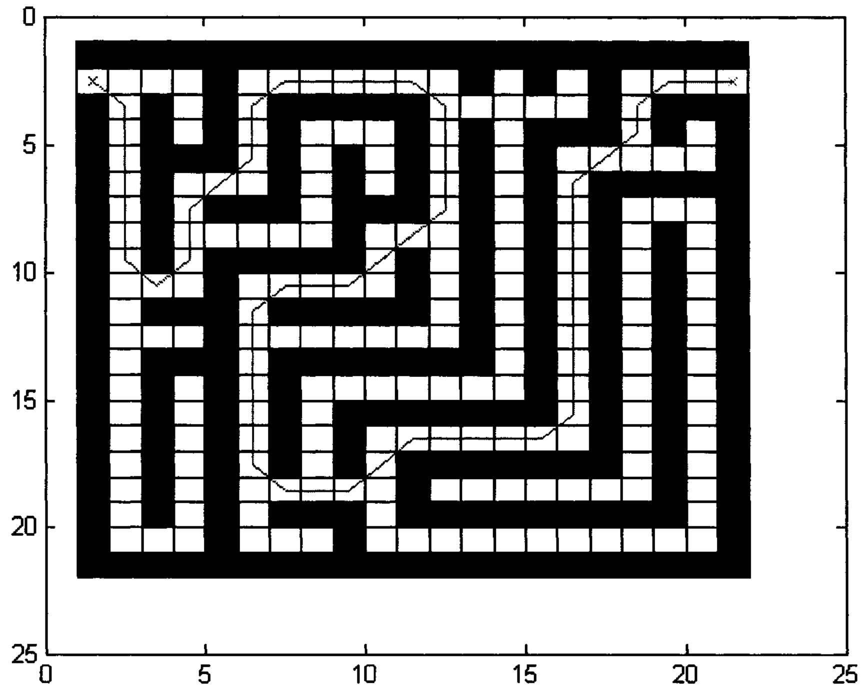


Figure 39 Solution to a labyrinth

4.10 Conclusion

The dynamic programming algorithm that was developed in this chapter has some interesting advantages over the widely used A* algorithm. Nevertheless, the calculation time of the A* algorithm remains a big advantage. A lot of algorithms and techniques that determine an optimal solution in a graph have been developed by many researchers, yet the A* algorithm remains the most efficient since its inception. The purpose of developing the dynamic programming algorithm was to create a new technique that has some advantages and to implement it. A more exhaustive study can be done to assess the execution of the dynamic programming algorithm on parallel computational resources. The optimal path is obtained in the form of waypoints corresponding to the positions of the nodes. In the following chapter, two path tracking techniques for controlling the robot motion along the waypoints of the optimal path are described and analyzed.

CHAPTER 5

HIGH LEVEL PATH TRACKING AND TRAJECTORY FOLLOWING

5.1 Introduction

In the literature review in Chapter 1, a variety of approaches for path tracking control of wheeled mobile robots have been presented. As discussed there, most of path tracking approaches controlling the robot at the dynamics level, and thus cannot be implemented if the robot dynamics are inaccessible. In this project, two types of controllers for path tracking and trajectory following were developed and tested in real time. In both approaches, control is done at the kinematic level. The controller in the first approach is based on fuzzy logic control (FLC). In the second approach, a classical control law was derived from a Lyapunov function. The control variables in this case are the desired translational and rotational speeds. The speeds are varied depending on the variations in the path and on the posture of the robot. The implementation on the P3-AT proves the performance of both approaches.

5.2 Fuzzy Logic Path Tracking Controller

In this section, a fuzzy logic controller (FLC) for the path tracking of a wheeled mobile robot based on controlling the robot at a higher level is presented. Motion is controlled by the translational and rotational velocities. The speeds are varied depending on the variations in the path and posture of the robot. The heuristic rules of the FLC are based on an analogy with a human driver and the optimization of the controller is based on experimentation.

5.2.1 Path Tracker Parameters

The path tracking controller that was implemented here is based on the controller in Driankov and Saffiotti (2001) but with some major changes in the inputs and outputs of the Fuzzy Logic Controller (FLC) and the rules, as well as in the path representation. The controller equation is as follows:

$$\begin{bmatrix} V \\ \omega \end{bmatrix} = \begin{bmatrix} f1(C, dR, d\theta, V_c) \\ f2(C, dR, d\theta, V_c) \end{bmatrix} \tag{5.1}$$

where V and ω are the translational and rotational velocities of the robot, C is the *look ahead curvature* (LAC) which is a feed forward input, dR the distance from the actual position of the robot to the next desired position, $d\theta$ the difference between the angles of the line joining the current position to the next desired position and the actual heading of the robot, V_c the current linear velocity (see Figure 40).

The functions *f1* and *f2* are the control laws of a Sugeno type fuzzy controller. Sugeno controllers take in fuzzy inputs and discrete outputs. The outputs are calculated separately. At first let's describe the parameters used for the controller. C is obtained using another fuzzy logic module whose inputs are $a1$ and $a2$. In Figure 40, the input parameters of the controller are illustrated for a certain posture of the robot. The trajectory is described by a set of discrete node positions N_1 to N_{Final} linked to each other starting from the initial position to the final desired position.

The task of the robot is to pass at the proximity of these points in the required order in a continuous and smooth manner. A continuous trajectory can be discretized as needed. The behavior of the controller is such that if the discrete points are close to each other, high precision but lower speeds will result.

If less precision is needed, the discrete points are selected further apart and the robot will move at higher speeds. The current node N_i is defined as the node whose position is nearest to the robot's current position. The next node N_{i+1} is the next node in the list of nodes on the trajectory and N_{i+2} is the one next to N_{i+1}. The angles $\alpha 1$ and $\alpha 2$ are the angles between the lines N_iN_{i+1} and $N_{i+1}N_{i+2}$, and between the lines $N_{i+1}N_{i+2}$ and $N_{i+2}N_{i+3}$, respectively. If $\alpha 1$ and $\alpha 2$ are large, then the robot must speed down to be able to make a smooth turn. C is a parameter that is function of angles $\alpha 1$ and $\alpha 2$ used to indicate the steepness of the curvature. The path is represented by a linked list of nodes starting with the start node and ending with the destination node. A pointer to the current node N_i points initially to the first node of the list. Whenever the robot gets nearer to the next node position N_{i+1}, the pointer would point to it, and it becomes the new current node. Consequently, the robot always heads in the direction of the node next to the current node. The pointer to the current node can only change incrementally starting from the beginning of the list.

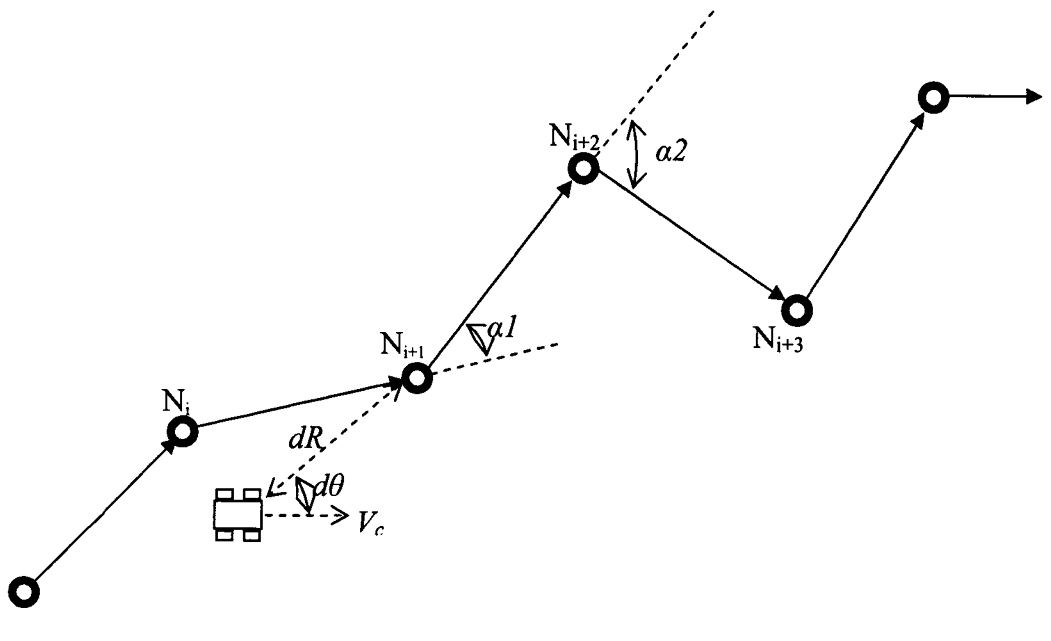


Figure 40 FLC parameters

If a robot has a high current velocity V_c, and needs to make a sharp turn $d\theta$, then it must first slow down while turning smoothly. When it has slowed down sufficiently, the robot can start making turn in response to the curvature. All the parameters to the input of the controller can be calculated knowing the current node and the robot's current position and heading, as well as its current velocity. The block diagram of Figure 41 is the general structure of the control loop.

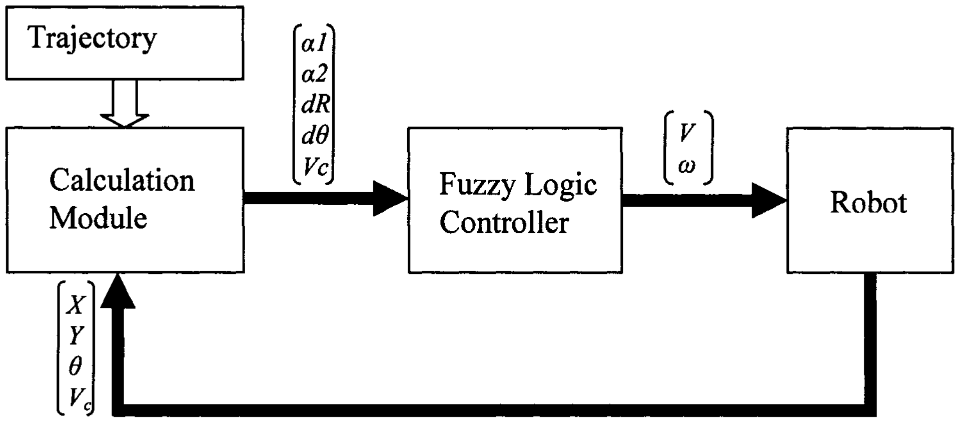


Figure 41 Control diagram

The calculation module takes in the position, heading, and current velocity of the robot, determines the current node state of the robot with respect to the trajectory, and calculates the controller parameters. The fuzzy logic controller then determines V and ω so that the robot follows the trajectory in a smooth and efficient manner.

5.2.2 Fuzzy Path Tracking Controller

The task of the path tracking fuzzy controller is to direct the robot to follow the trajectory in a smooth and continuous manner as precisely as possible. It might not be necessary that the robot passes exactly through the points on the trajectory, but at least pass at their proximity and arrive to the final destination. The closer the discrete points are to each other, the more precise the robot will be in executing the trajectory but at a lower speed. Figure 42 shows the schematic of the FLC. The first module determines the look-ahead curvature (LAC) value and the path tracker module determines the linear velocity and angular speed to be output to the robot. Both modules are Sugeno type fuzzy inference systems of order zero. The document International Technical Commission (IEC) (1997) contains a brief and practical introduction to fuzzy control. For a more detailed analysis on fuzzy control, Farinwata (2000) provides a more in-depth theoretical study. The LAC uses the angles *a1* and *a2* to determine the value of *C*. The membership functions of each of the parameters are shown in Figure 43. The membership functions of *a1* are *Straight1*, *High1*, and *VeryHigh1* and those of α2 are *Straight2*, *High2*, and *VeryHigh2*. The membership functions of C are singletons that take values between zero and five. The zero value indicates that there is no curvature meaning that *a1* and *a2* are small and the robot will follow a straight line at the current state. If the curvature is high then the robot must speed down to make the sharp turn. The inference rules map the membership functions of the input parameters to the membership functions of the output.

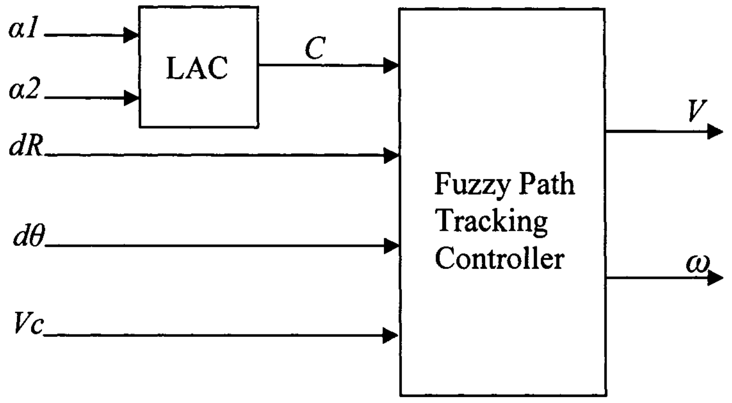


Figure 42 FLC Schematic

For example the rule:

IF *a1* is *Straight1* AND *a2* is *VeryHigh2* THEN *C* is *cHigh*

Maps membership functions *Straight1* and *VeryHigh2* of the inputs to membership function *cHigh* of the output. If the two conditions of the inputs are satisfied for this rule, then the value four corresponding to *cHigh* is returned for this rule. The truth value for the rule is obtained by using the product of the truth values of *Straight1* and *VeryHigh2*:

$$A_i = \mu(Straight1(\alpha 1))\mu(VeryHigh2(\alpha 2)) \quad \textbf{(5.2)}$$

Where μ is a value between zero and one that indicates the truth value that an input value belongs to some membership function. The values of all rules are returned and the output returned by the LAC module is defuzzified using the center of gravity method for singleton (COGS) (International Technical Commission (IEC), 1997). The formula of this method is:

$$U(t_k) = \frac{\sum_{i=1}^{p} U_i A_i(t_k)}{\sum_{i=1}^{p} A_i(t_k)} \quad \textbf{(5.3)}$$

Where A_i are the singleton values (i.e. truth values) of the individual rules, and U_i their corresponding outputs. Figure 44 shows the input-output characteristics of the LAC. It is clear that the output rises faster as *a1* increases. The rule base is shown in Table 1.

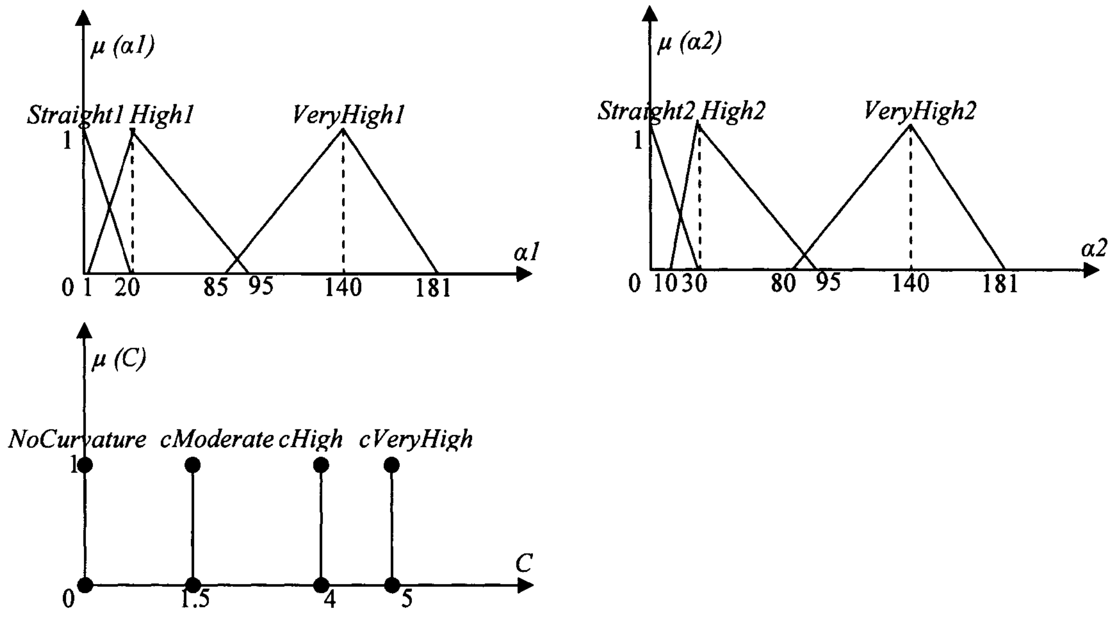


Figure 43 LAC parameters and their corresponding membership functions

Table I

Inference rules for LAC

α1 / α2	Straight1	High1	VeryHigh1
Straight2	NoCurvature	cHigh	cVeryHigh
High2	cModerate	cHigh	cVeryHigh
VeryHigh2	cHigh	cHigh	cVeryHigh

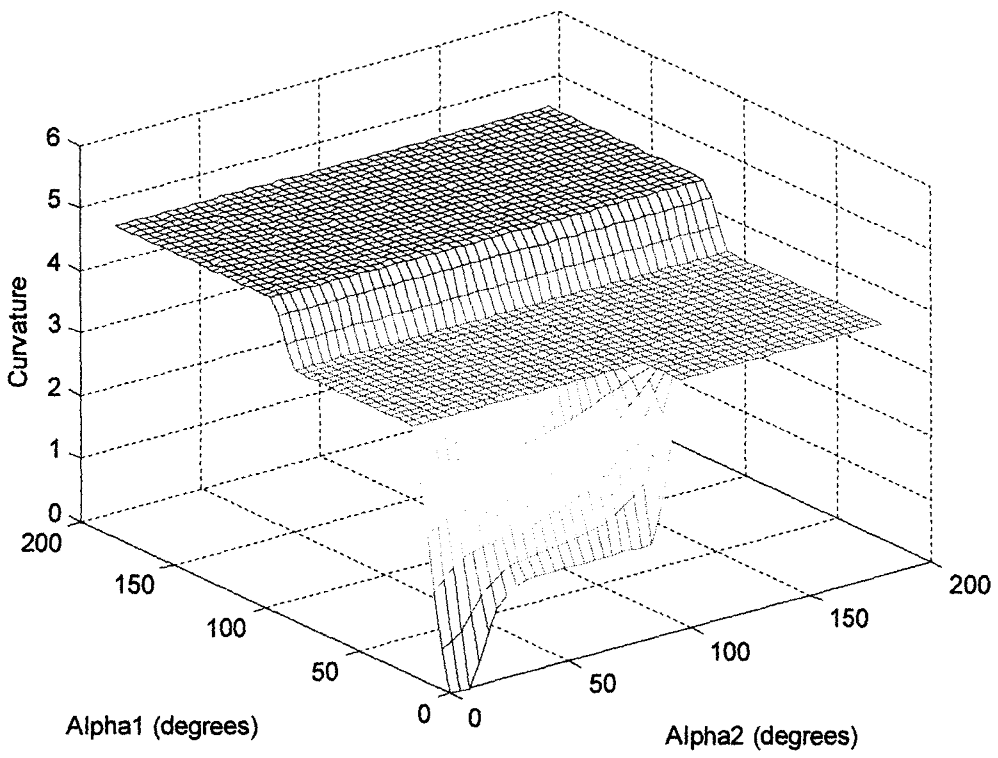


Figure 44 Input-Output surface for LAC

The value C is then fed to the path tracking controller along with dR, $d\theta$, and V_c. The membership functions of each of the input and output parameters are shown in Figure 45. As expected, C ranges from zero to five.

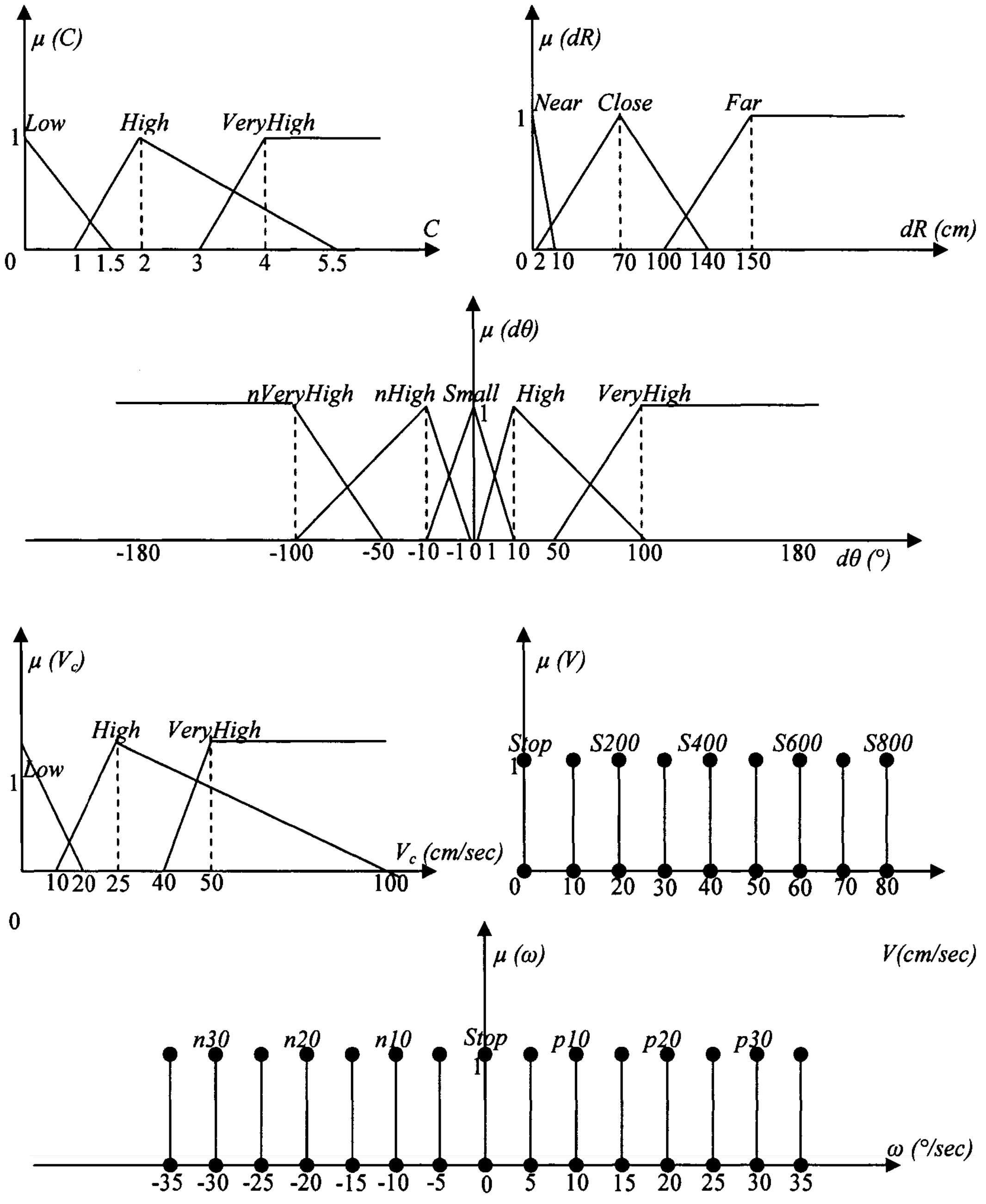


Figure 45 Membership functions of path tracker parameters

Input *dR* ranges between zero and 7000 mm, *dθ* ranges from -180 to +180 degrees, and Vc from zero to 1000 mm/sec. The linear velocity output ranges from zero to 800 mm/sec and the angular speed from -30 to +30 degrees/sec.

The inference rules maps the input membership functions to the output membership functions. The behavior of the controller is such that it changes linear velocity and angular speed in a smooth and almost continuous manner. When the curvature is sharp, the controller decreases the speed and outputs the needed rotational speed in the right direction to make the turn smoothly. When the curvature is smooth, the robot will speed up and the rotational speed is small so as to stay on track. For example the rule:

IF *C* is *High* THEN *V* is *S300*

sets the speed to 300mm/sec when the curvature value is high no matter what the other values at the inputs are if it is the only inference rule that is activated by the inputs. If not, the COGS defuzzification method mentioned above is used again to calculate the outputs. The value of rotational speed is not affected by this rule. Another example is when the current velocity V_c is high and either $d\theta$ or C is high, the robot should slow down first before making the turn. Figures 46 and 47 show the input-output characteristics at C and V_c both set to zero. Note the symmetry with respect to the plane that satisfies $d\theta=0$.

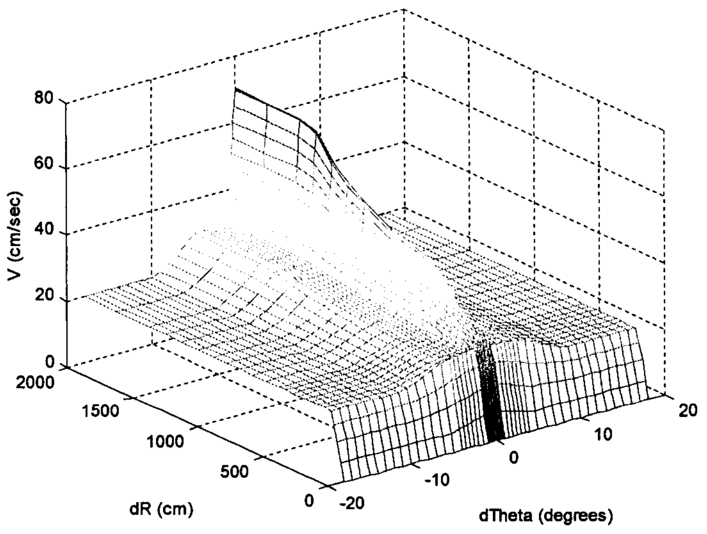


Figure 46 V output when C and Vc are zero

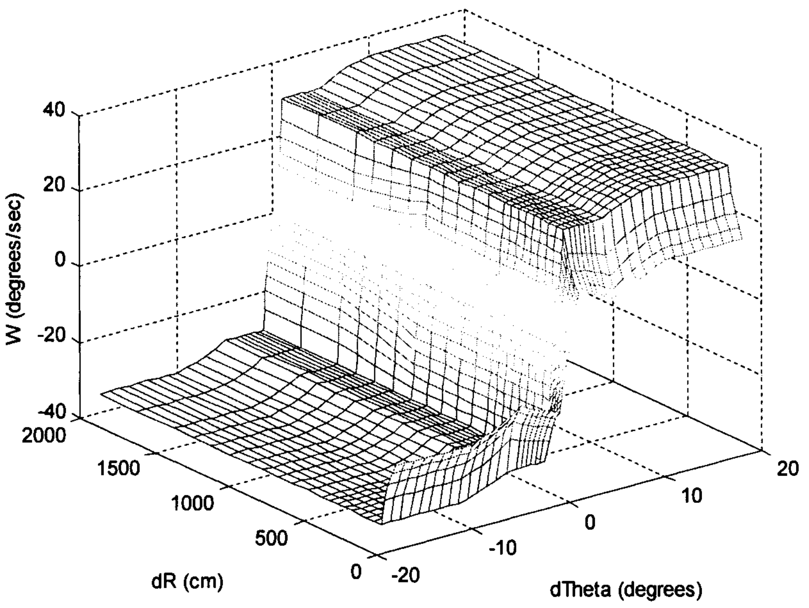


Figure 47 Angular speed output for C and Vc are zero

5.2.3 Real Time Implementation

The path tracking FLC shown above is implemented using the C++ *Free Fuzzy Logic Library* (FFLL) along with the *Activmedia Robotic Interface for Application* (ARIA) library that provides extensive methods to control the Pioneer robot, communicate with it, and obtain its sensor information. The fuzzy controllers are specified in a Fuzzy Control Language (FCL) files using the IEC 61131-7 industrial standard (International Technical Commission (IEC), 1997). The FFLL contains methods to read FCL files that contain the input and output information and membership functions, the defuzzification method(s), and the inference rules. The FFLL also contains the methods needed to calculate the outputs once the FCL files are read. A new interface with the robot that provides 3D views was used. The real time implementation was tested on the Pioneer 3AT four-wheel differentially driven mobile robot used for all terrain navigation. The lower level controllers of the motors are separate with a PID controller for each motor alone. Only the values of the gains of the PID controllers can be modified to cope with the weight of the robot, but only reference speeds or positions can be controlled.

5.2.3.1 FFLL and FCL Files

To implement the fuzzy logic path tracker and the look-ahead curvature (LAC) approximation, the free fuzzy logic library (FFLL) developed for artificial intelligence (AI) applications implemented in C++ was used. FFLL is easy to use and is compliant on the "basic level" with the fuzzy control language (FCL) as stated in table 6.1_1 of IEC 61131-7 [ref]. It is distributed for free and is open source. As specified by its developers, FFLL was designed to be fast in performing fuzzy calculations. FFLL makes use of look-up tables thus trading memory for some significant gain in speed. All details on FFLL are available on Site (2002). Only the application programmer interface (API) functions used in this project will be described as well as the FCL files. The API header file *FFLLAPI . h* contains the prototypes of all the functions that are used by a user application. To compile, build, and execute an application that makes use of FFLL, the API header file along with the *FFLLAPI . LIB* and *FFLLAPI . DLL* are the only files needed from the FFLL package that can be downloaded from Site (2002).

We will now discuss how the API functions are used in a program. First a model is created using **ffll_new_model** and an *int* value that refers to the created model is returned. Then the FCL file (file that contains all the description of the fuzzy logic module, such as input and output membership functions as well as the defuzzification technique and the fuzzy inference functions) is loaded into the model using **ffll_load_fcl_file**. Then a child object of the model is created using **ffll_new_child**. More than one child can be created for the same model, in the case that two or more exactly identical fuzzy inference systems are used for the same application. The child object will be used to implement the fuzzy module, and thus takes the inputs and returns the outputs of the fuzzy outputs. All these functions are for initialization and are used only once for each fuzzy logic module. Function **ffll_set_value** takes as arguments the feedback inputs as well as the child object number returned when **ffll_new_child** was used. After setting the inputs, **ffll_get_output** is used to return the outputs. These two functions will be used in a sequence of two all throughout program execution as specified by the user. The usage of these functions to implement the fuzzy logic path tracker will be presented in the next subsection.

The three FCL files corresponding to the three fuzzy modules are available in Appendix 2. An FCL file is a standard used for fuzzy logic industrial applications, each with its own application interface. FFLL is actually an interface for two FCL files for a certain control applications. The first keyword in an FCL file is *FUNCTION_BLOCK*, and a name of the *FUNCTION_BLOCK* can be specified. Its delimiter is the end *FUNCTION_BLOCK* at the end of the file. The *VAR_INPUT* and *VAR_OUTPUT* are used to indicate the list of inputs and outputs respectively. Both must end with the *END_VAR* indicator. *FUZZYFY* followed by the name of the variable specified in *VAR_INPUT* or *VAR_OUTPUT* is used to specify the membership functions for each of the inputs and outputs.

The member functions are specified as a set of discrete points corresponding to points on the actual continuous curve using the *TERM* keyword. The defuzzification method and the interpretation of the *AND* of membership functions are indicated after *DEFUZZIFY* followed by *AND* and *METHOD*. And finally the fuzzy inference rules are listed after *RULEBLOCK*. Special attention to the format of the FCL file should be made for all terms to be put correctly. Comments are inserted between (* and *) characters and are disregarded by the FCL interface. The three FCL files corresponding to each of the fuzzy logic modules used in the path tracker application are available in Appendix 2.

5.2.3.2 C++ Implementation

For the purpose of modularity, all the inner functionalities of the path tracking controllers have been implemented outside the **ArAction** method of Appendix 1. The inner functionalities have been implemented in the **PathTracker**. All what needs to be done **ArAction** is to create a **PathTracker** object and call the **PathTracker :: FLPathTrackerC** function along with all inputs required to calculate the feed back inputs to the fuzzy models, as well as **PathTracker :: setTrajectory** in order to load the waypoints into the **PathTracker** object. The initialization and creation of the FFLL models and their corresponding childs, as well as the input and output setting and recuperation on each of the child objects are implemented in the **PathTracker** method.

The **PathTracker :: currentState** variable is a pointer to the current nearest next waypoint. In the constructor, it is initialized to the first waypoint of the trajectory. **PathTracker :: PathNodesList** is the list of waypoints of the trajectory. The fuzzy models and child objects are also created in the constructor using **PathTracker :: initializeFuzzyControllers**. The type *int* variables corresponding to the models and childs are member variables of **PathTracker**, and are used in **PathTracker :: initializeFuzzyControllers** as well as in input and output extraction to refer to each of the fuzzy child objects. **PathTracker :: findCurrentState** is used to find the instantaneous next waypoint. **PathTracker :: FLPathTrackerC** is the function used as the interface to obtain the output from a set of inputs. This function calculates all inputs to the fuzzy modules and returns a pointer to the **VelParameters** structure containing the translational and rotational velocities. The functions **PathTracker::findCurvature**, **PathTracker::velocityOutput**, and **PathTracker::omegaOutput** make use of the FFLL-API functions **ffll_set_value** and **ffll_get_output** to send input and get the output from the FFLL child objects.

5.2.4 Simulation and Experimental Results:

The path tracking controller has been simulated on the SRI simulator with the parameters of the P3-AT robot as well as on the robot itself. The controller corrects the path of the robot relying on the position returned to it by the encoders and therefore some error due to localization will be unavoidable. To evaluate the performance of the controller, tests were conducted on trajectories having some sharp turns and some others that are smoother. The first such one is shown in Figure 48, a straight line path discretized at 25cm intervals with the robot initially at a distance of 1m from the path. The robot joins the path and traverses at the proximity of the node points with an error inferior to 1cm when it reaches the straight line. This very small error is due to the dynamics of the system.

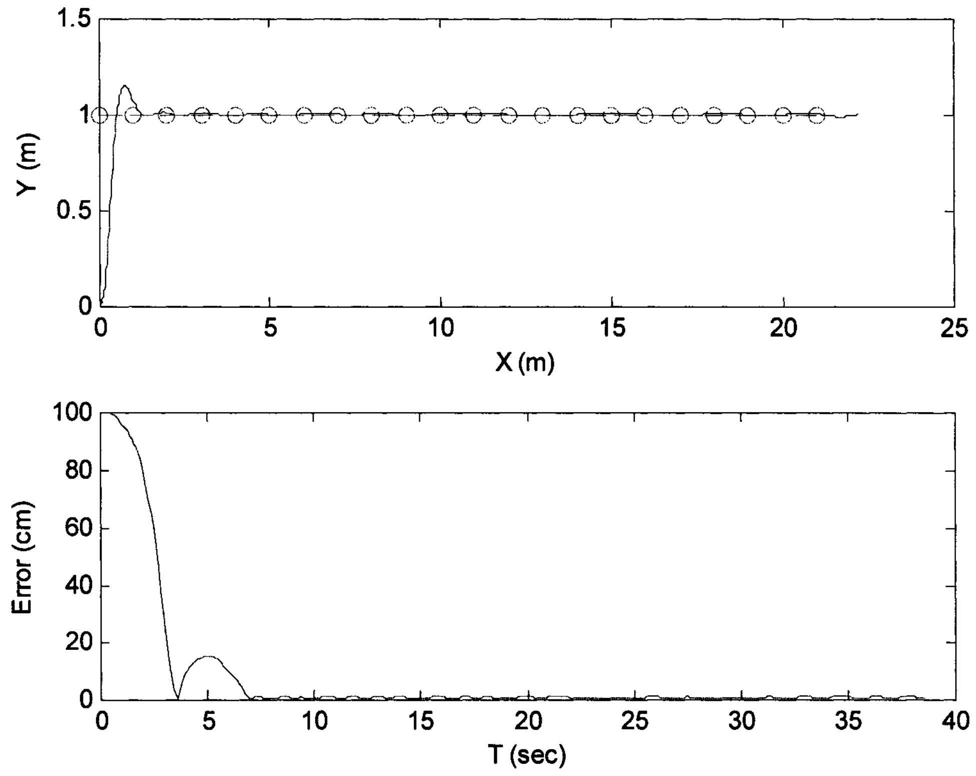


Figure 48 Straight line with the robot 1meter away

Figure 49 shows the performance on a sine wave trajectory with a period of two meters discretized at about 25cm. The initial error of about 15cm is due to the initial orientation of the robot. When the robot is on track the error was less than 2cm. The desired and actual trajectories almost coincide with each other.

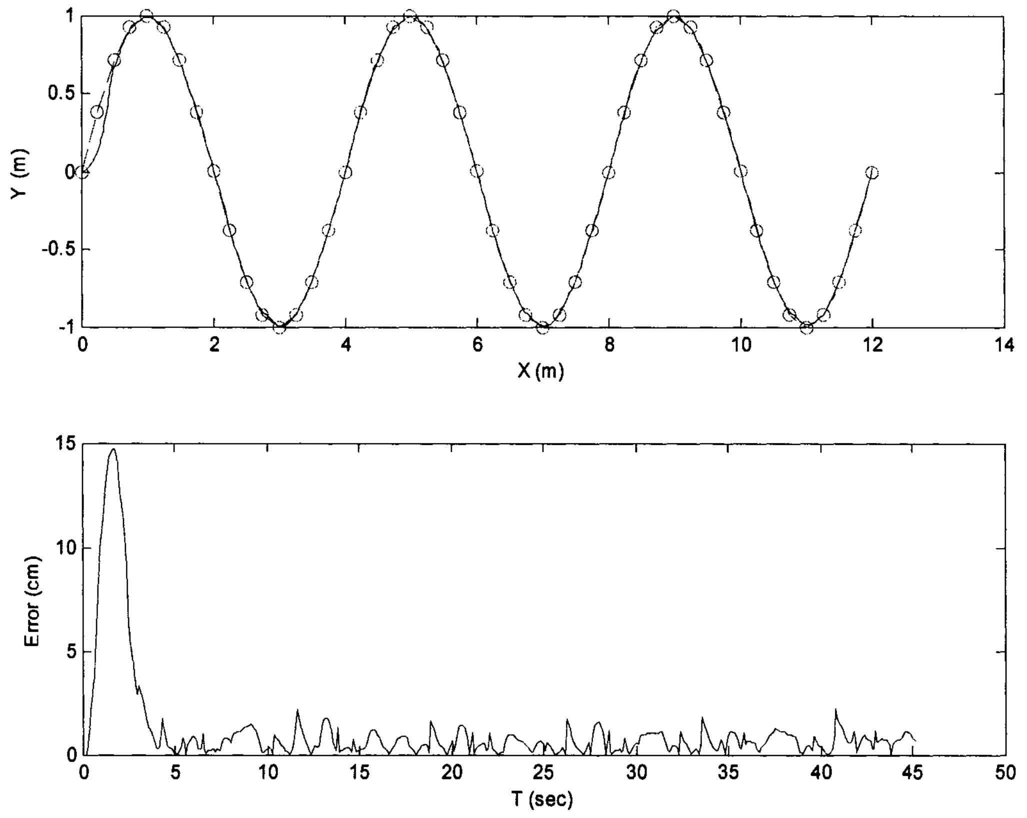


Figure 49 Sine wave trajectory at 25 cm discretization

Figure 50 shows the results of simulations performed on a general trajectory with some very sharp turns in an area of 10mx10m. The way points are interspaced at more than one meter apart. The error was mostly inferior to 30cm. The sharp discontinuities in the errors are due to the fact that the errors are measured by the distance fron the line joining two waypoints to the actual position. This is due to the fact that there is no continuous reference trajectory. The results are acceptable for most mobile robots applications.

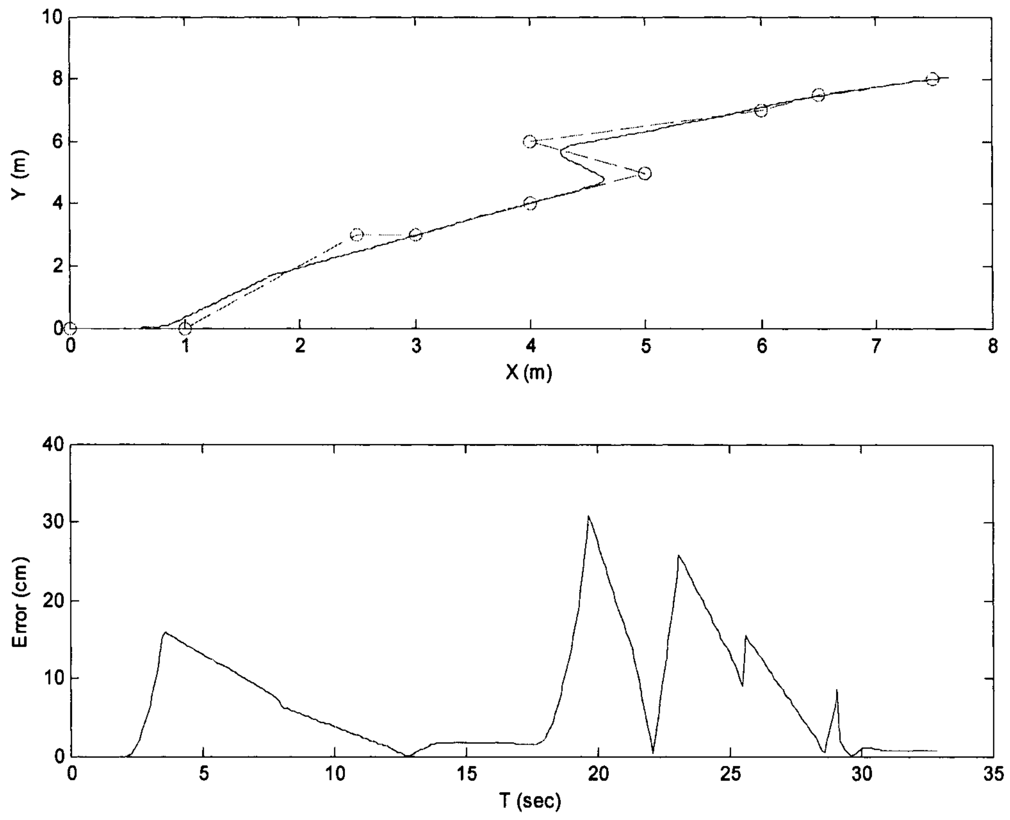


Figure 50 General trajectory

The behavioral fuzzy logic path tracking controller that we have implemented proved to be very reliable and robust in terms of precision and speed. Despite the fact that fuzzy logic control is not based on a precise mathematical model, it is robust and flexible. A lower level controller can be implemented independent from the path tracking problem or other behaviors that can be independently integrated.

5.3 Lyapunov Based Control Law Approach to the Problem of Path Tracking

In this section, a more theoretical approach based on a Lyapunov function was used to implement a path tracking controller. The approach used is the same as in Kanayama, Kimura et al. (1990) and is based on the error model of the kinematic model.

5.3.1 The Control Technique

Knowing that the kinematical model of a differentially steered wheeled mobile robot in Cartesian coordinates is given by the following equation:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix} \qquad \textbf{(5.4)}$$

The objective is to track a reference robot. The relation between the velocities of the reference robot V_r and ω_r, and its posture by x_r, y_r, and θ_r is as follows:

$$\begin{bmatrix} \dot{x}_r \\ \dot{y}_r \\ \dot{\theta}_r \end{bmatrix} = \begin{bmatrix} \cos\theta_r & 0 \\ \sin\theta_r & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_r \\ \omega_r \end{bmatrix} \qquad \textbf{(5.5)}$$

Then, three error variables e_x, e_y, and e_θ that correspond to the instantaneous errors in posture variables are chosen as:

$$\begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_r - x \\ y_r - y \\ \theta_r - \theta \end{bmatrix} \qquad \textbf{(5.6)}$$

These errors would be the errors in posture with respect to the local frame of reference of the robot. The transformation matrix converts global coordinates to local coordinates.

Calculation of the derivatives of the errors using the constraint $\dot{x}_r \sin\theta_r = \dot{y}_r \cos\theta_r$ and with $\theta_e = e_\theta = \theta_r - \theta$ gives (Kanayama, Kimura et al., 1990):

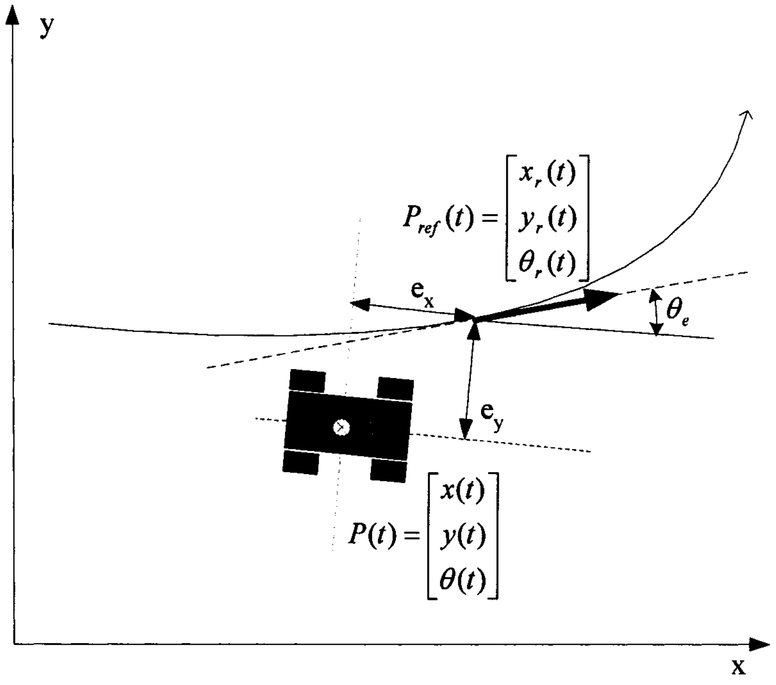

y
x
$P_{ref}(t) = \begin{bmatrix} x_r(t) \\ y_r(t) \\ \theta_r(t) \end{bmatrix}$
e_x
e_y
θ_e
$P(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}$

Figure 51 Current and reference postures and posture errors

$$\begin{aligned}
\dot{e}_x &= (\dot{x}_r - \dot{x})\cos\theta + (\dot{y}_r - \dot{y})\sin\theta - (x_r - x)\dot{\theta}_c \sin\theta + (y_r - y)\dot{\theta}\cos\theta \\
&= e_y\omega - v + \dot{x}_r \cos\theta + \dot{y}_r \sin\theta \\
&= e_y\omega - v + \dot{x}_r \cos(\theta_r - \theta_e) + \dot{y}_r \sin(\theta_r - \theta_e) \\
&= e_y\omega - v + \dot{x}_r(\cos\theta_r \cos\theta_e + \sin\theta_r \sin\theta_e) + \dot{y}_r(\sin\theta_r \cos\theta_e - \cos\theta_r \sin\theta_e) \\
&= e_y\omega - v + (\dot{x}_r \cos\theta_r + \dot{y}_r \sin\theta_r)\cos\theta_e + (\dot{x}_r \sin\theta_r + \dot{y}_r \cos\theta_r)\sin\theta_e \\
&= e_y\omega - v + v_r \cos\theta_e
\end{aligned} \tag{5.7}$$

and

$$\begin{aligned}
\dot{e}_y &= (\dot{x}_r - \dot{x})\sin\theta + (\dot{y}_r - \dot{y})\cos\theta - (x_r - x)\dot{\theta}_c\cos\theta + (y_r - y)\dot{\theta}\sin\theta \\
&= -e_x\omega + \dot{x}\sin\theta + \dot{y}\cos\theta - \dot{x}_r\sin\theta + \dot{y}_r\cos\theta \\
&= -e_x\omega + \dot{x}_r\sin(\theta_r - \theta_e) + \dot{y}_r\cos(\theta_r - \theta_e) \\
&= -e_x\omega + \dot{x}_r(\sin\theta_r\cos\theta_e - \cos\theta_r\sin\theta_e) + \dot{y}_r(\cos\theta_r\cos\theta_e + \sin\theta_r\sin\theta_e) \\
&= -e_x\omega + (\dot{x}_r\cos\theta_r + \dot{y}_r\sin\theta_r)\sin\theta_e + (\dot{x}_r\sin\theta_r + \dot{y}_r\cos\theta_r)\cos\theta_e \\
&= -e_x\omega + v_r\sin\theta_e
\end{aligned} \tag{5.8}$$

Which we put in matrix format:

$$\begin{bmatrix} \dot{e}_x \\ \dot{e}_y \\ \dot{e}_\theta \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} v + \begin{bmatrix} e_y \\ -e_x \\ -1 \end{bmatrix} \omega + \begin{bmatrix} v_r\cos e_\theta \\ v_r\sin e_\theta \\ \omega_r \end{bmatrix} \tag{5.9}$$

From equation (5.9) above, the aim of a control law is to make the errors converge to zero. The proposed velocity inputs v_f and ω_f of the control law (Kanayama, Kimura et al,. 1990) are:

$$\begin{aligned}
v_f &= v_r\cos e_\theta + K_x e_x \\
\omega_f &= \omega_r + V_r K_y e_y + K_\theta \sin e_\theta
\end{aligned} \tag{5.10}$$

By substituting v_f and ω_f in the errors of equation (5.9), we get:

$$\begin{bmatrix} \dot{e}_x \\ \dot{e}_y \\ \dot{e}_\theta \end{bmatrix} = \begin{bmatrix} e_y(\omega_r + v_r(K_y e_y + K_\theta\sin e_\theta)) - K_x e_x \\ -e_x(\omega_r + v_r(K_y e_y + K_\theta\sin e_\theta)) + v_r\sin e_\theta \\ -v_r(K_y e_y + K_\theta\sin e_\theta) \end{bmatrix} \tag{5.11}$$

The Lyapunov energy function V_0 is chosen as such:

$$V_0 = \frac{1}{2}(e_x^2 + e_y^2) + \frac{1-\cos e_\theta}{K_y} \tag{5.12}$$

If the proposed control law was used, there will be a stable equilibrium at $\bar{e} = 0$ if $v_r > 0$,

where $\bar{e} = \begin{bmatrix} e_x \\ e_y \\ e_\theta \end{bmatrix}$

By deriving V_0 with respect to time, we get:

$$\dot{V}_0 = e_x\dot{e}_x + e_y\dot{e}_y + \left[-\left(\omega_r + v_r\left(K_y y_e + K_\theta \sin\theta_e\right)\right)x_e + v_r \sin\theta_e\right]e_y + \left[-v_r\left(K_y y_e + K_\theta \sin\theta_e\right)\right]\sin\theta_e / K_y = -K_x e_x^2 - \frac{K_\theta \sin^2 e_\theta}{K_y} \le 0 \tag{5.13}$$

Given that K_x, K_y, and K_θ are all positive constants, the above inequality would be satisfied and the system with the control law would be stable.

Furthermore, if v_r and ω_r are continuous and bounded, then $\bar{e} = 0$ is uniformly asymptotically stable. The error system is linearized around $\bar{e} = 0$, to get a linearized system of the form $\dot{\bar{e}} = A\bar{e}$.

The nonlinear system is of the form:

$$\dot{\bar{e}} = \begin{bmatrix} f_1(\bar{e}) \\ f_2(\bar{e}) \\ f_3(\bar{e}) \end{bmatrix} = f(\bar{e}) \tag{5.14}$$

To linearize the system using the Taylor equation for derivatives, the matrix A would be calculated as follows:

$$A = \left.\frac{\partial f}{\partial \bar{e}}\right|_{\bar{e}=0} = \left.\begin{bmatrix} \frac{\partial f_x}{\partial e_x} & \frac{\partial f_x}{\partial e_y} & \frac{\partial f_x}{\partial e_\theta} \\ \frac{\partial f_y}{\partial e_x} & \frac{\partial f_y}{\partial e_y} & \frac{\partial f_y}{\partial e_\theta} \\ \frac{\partial f_\theta}{\partial e_x} & \frac{\partial f_\theta}{\partial e_y} & \frac{\partial f_\theta}{\partial e_\theta} \end{bmatrix}\right|_{\bar{e}=0} = \begin{bmatrix} -K_x & \omega_r & 0 \\ -\omega_r & 0 & v_r \\ 0 & -v_r K_y & -v_r K_\theta \end{bmatrix} \tag{5.15}$$

The eigenvalues of A can be obtained by the following equation:

$$\det(A - \lambda I) = 0 \quad \textbf{(5.16)}$$

which gives an equation of the form:

$$a_3\lambda^3 + a_2\lambda^2 + a_1\lambda + a_0 = 0 \quad \textbf{(5.17)}$$

where

$$\begin{aligned} a_3 &= 1 \\ a_2 &= K_x + v_r K_\theta \\ a_1 &= K_x K_\theta v_r + v_r{}^2 K_y + \omega_r{}^2 \\ a_0 &= K_x K_y v_r{}^2 + \omega_r{}^2 v_r K_\theta \end{aligned} \quad \textbf{(5.18)}$$

Since all terms are positive, the Routh-Hurwitz criterion can be used to determine that all the eigenvalues λ are negative, and thus the system is asymptotically stable at $\bar{e} = 0$.

5.3.2 Generating the Reference Trajectory:

Given the Cartesian coordinates of the waypoints of a trajectory planned offline, it is required to find the linear and angular velocity, and the posture (position and orientation) as a function of time so that the robot can pass through the waypoints in a certain predefined time t_f starting from t_0 at the first waypoint.

Given N waypoints, the time to pass from a waypoint i to the next waypoint $i+1$, is calculated by the following:

$$\Delta t_i = \left| K\Delta\theta_{i+1}\frac{d_i}{V_{\max}} + \frac{d_i}{V_{\max}} \right| \quad \textbf{(5.19)}$$

where d_i is the distance between waypoints i and $i+1$, $\Delta\theta_{i+1}$ is the change in direction at waypoint $i+1$, K is a constant factor, and V_{max} the maximum linear velocity the robot can attain.

The higher the $\Delta\theta_{i+1}$, the more time will be allotted to make a turn and change the direction. If $\Delta\theta_{i+1}=0$, then minimum time is allowed so that the robot rolls at V_{max}. All Δt_i are then rounded to the nearest 100 milliseconds, knowing that the robot sample time is 100 milliseconds. To obtain the time from t_0 till waypoint i, we sum all the Δt before i,

$$t_i = \sum_{j=1}^{i} \Delta t_i \qquad \textbf{(5.20)}$$

We use a cubic spline to interpolate the coordinate x(t) with respect to t, and y(t) with respect to t, using the two sets $\{ t_0, \ldots t_f; x_0, \ldots x_f \}$ and $\{t_0, \ldots t_f; x_0, \ldots x_f\}$respectively. What we get is a matrix of coefficients for each of the two sets:

$$A = \begin{bmatrix} a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ \ldots & \ldots & \ldots & \ldots \\ a_{N0} & a_{N1} & a_{N2} & a_{N3} \end{bmatrix}_{N\times 4} \text{ and } B = \begin{bmatrix} b_{10} & b_{11} & b_{12} & b_{13} \\ b_{20} & b_{21} & b_{22} & b_{23} \\ \ldots & \ldots & \ldots & \ldots \\ b_{N0} & b_{N1} & b_{N2} & b_{N3} \end{bmatrix}_{N\times 4}.$$

and thus we have:

$$x(t) = \begin{cases} a_{10} + a_{11}t + a_{12}t^2 + a_{13}t^3, t \in [t_0, t_1[\\ a_{20} + a_{21}t + a_{22}t^2 + a_{23}t^3, t \in [t_1, t_2[\\ \ldots\ldots \\ a_{N0} + a_{N1}t + a_{N2}t^2 + a_{N3}t^3, t \in [t_{N-1}, t_N] \end{cases} \qquad \textbf{(5.21)}$$

and

$$y(t) = \begin{cases} b_{10} + b_{11}t + b_{12}t^2 + b_{13}t^3, t \in [t_0, t_1[\\ b_{20} + b_{21}t + b_{22}t^2 + b_{23}t^3, t \in [t_1, t_2[\\ \ldots\ldots \\ b_{N0} + b_{N1}t + b_{N2}t^2 + b_{N3}t^3, t \in [t_{N-1}, t_N] \end{cases} \quad \textbf{(5.22)}$$

Therefore *x(t)* and *y(t)* are independent, and it is simple to calculate their derivatives $\dot{x}(t)$ and $\dot{y}(t)$ respectively.

$$\dot{x}(t) = \begin{cases} a_{11} + 2a_{12}t + 3a_{13}t^2, t \in [t_0, t_1[\\ a_{21} + 2a_{22}t + 3a_{23}t^2, t \in [t_1, t_2[\\ \ldots\ldots \\ a_{N1} + 2a_{N2}t + 3a_{N3}t^2, t \in [t_{N-1}, t_N] \end{cases} \quad \textbf{(5.23)}$$

and

$$\dot{y}(t) = \begin{cases} b_{11} + b_{12}t + b_{13}t^2, t \in [t_0, t_1[\\ b_{21} + b_{22}t + b_{23}t^2, t \in [t_1, t_2[\\ \ldots\ldots \\ b_{N1} + b_{N2}t + b_{N3}t^2, t \in [t_{N-1}, t_N] \end{cases} \quad \textbf{(5.24)}$$

The linear velocity *V(t)* and the orientation angle *θ(t)* can then be calculated as follows:

$$V(t) = \sqrt{\left(\dot{x}^2(t) + \dot{y}^2(t)\right)} \quad \textbf{(5.25)}$$

and

$$\theta(t) = ATAN2(\dot{y}(t), \dot{x}(t)) \quad \textbf{(5.26)}$$

where *atan2* refers to the inverse tangent that takes into account the signs of the *sin* and *cos* so as to determine a unique angle. And finally the angular velocity *ω(t)* is calculated simply by deriving *θ(t)* with respect to *t*, and we get:

$$\omega(t) = \frac{1}{1+\left(\dot{y}(t)/\dot{x}(t)\right)^2}\frac{\ddot{y}(t)\dot{x}(t)-\ddot{x}(t)\dot{y}(t)}{\dot{x}^2(t)} \tag{5.27}$$

There is a kinematics constraint on the radius of curvature. The robot can turn around itself with a zero radius of curvature; otherwise its radius of curvature must be greater than a certain R_{min}. The instantaneous radius of curvature is:

$$R(t) = \frac{V(t)}{\omega(t)} \tag{5.28}$$

If R is smaller than R_{min}, we can limit $\omega(t)$ to $V(t)/R_{min}$ so as to respect this kinematics constraint. Given $x(t)$, $y(t)$, $\theta(t)$, $V(t)$, and $\omega(t)$, the reference trajectory is fully defined.

5.3.3 Real Time Implementation

To implement the controller described in this section, the same logic for implementing the fuzzy controller in the previous section was used, but without using FFLL. Instead, a vector containing the coefficients of the trajectory intervals returned by the **cSpline** function as well as the controller gains are initialized. In this technique, the time factor is used since the robot has to follow a reference trajectory. The reference position, velocities, and accelerations are calculated using the coefficients stored and the reference time. The function **cSpline** is defined in *Spline.h*. The member function that returns the desired velocities is **PathTracker::PathTrackerC**. The header files and their corresponding source files are available in Appendix 2.

5.3.4 Simulation Results

The approach has been implemented and tested on the P3-AT in real time. The gains of the control law were chosen according to the recommendations of Kanayama, Kimura et al. (1990), $K_x=2.5$, $K_y=0.75$, and $K_\theta=1.41$. The robot successfully followed its path and reached its target destination. The testing results for three different cases are displayed in

Figures 52 to 57. In Figure 52 the reference trajectory is a unit step function with the initial position of the robot at the origin and the initial heading is horizontal at zero degrees. The response to a sharp discontinuous change in reference position is slower than the response of the fuzzy logic controller. The steady state errors in distance are inferior to 20 cm. In Figure 53 the reference and actual velocities are displayed. To note is the large errors in velocities at the beginning when the robot is converging to the trajectory.

In the case of a sine wave trajectory with the robot initially at the origin and heading horizontally at zero degrees (Figure 54), the error is also inferior to 20 cm compared to 3 cm of error for the fuzzy logic controller. We note however the fast response to the discontinuous change in heading at the beginning when compared to the fuzzy logic controller. The velocities errors (Figure 55) are comparable to those of the step response.

Finally Figures 56 and 57 display the results for a general trajectory with some sharp turns. The error in distance is always inferior to 20 cm.

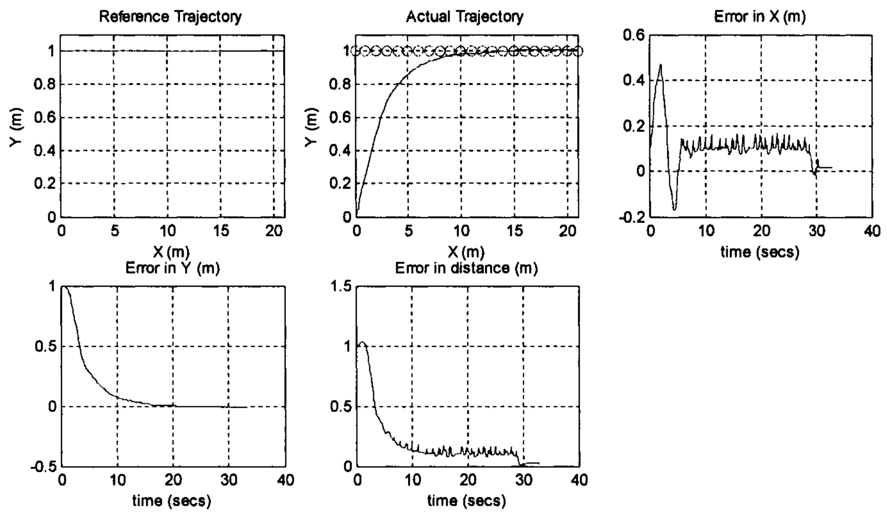


Figure 52 Position and position errors

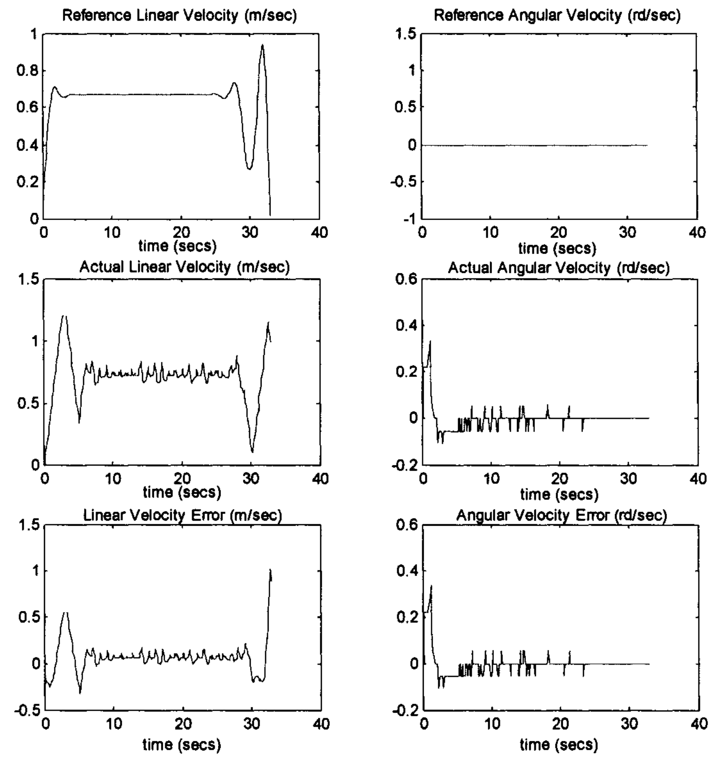


Figure 53 Velocities and velocity errors

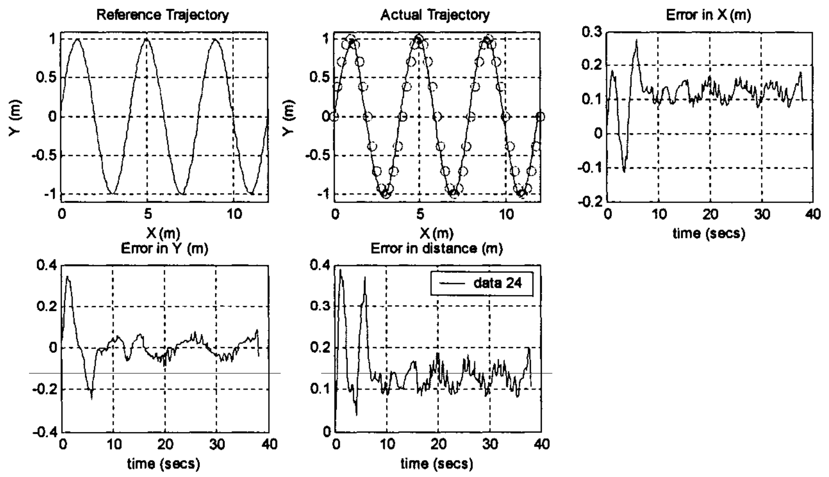


Figure 54 Position and position errors

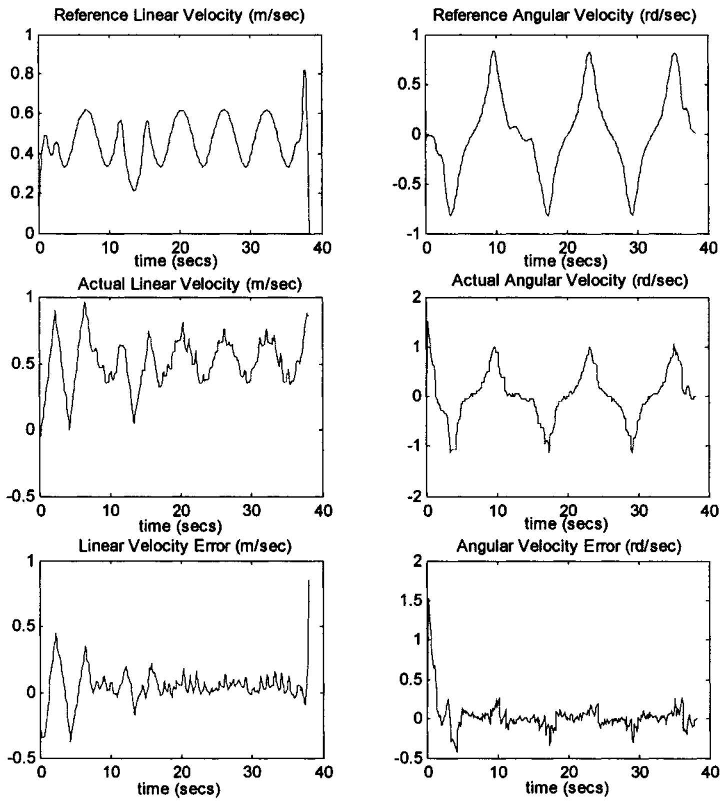


Figure 55 Velocities and velocity errors

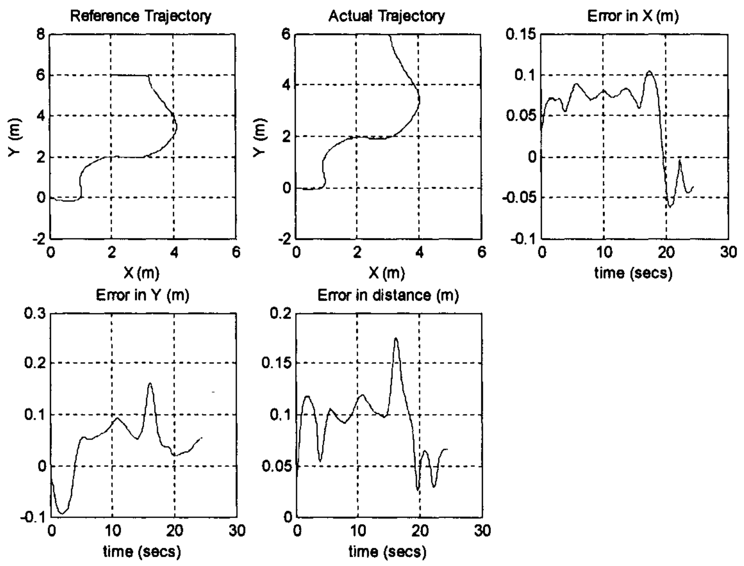


Figure 56 Position and position errors

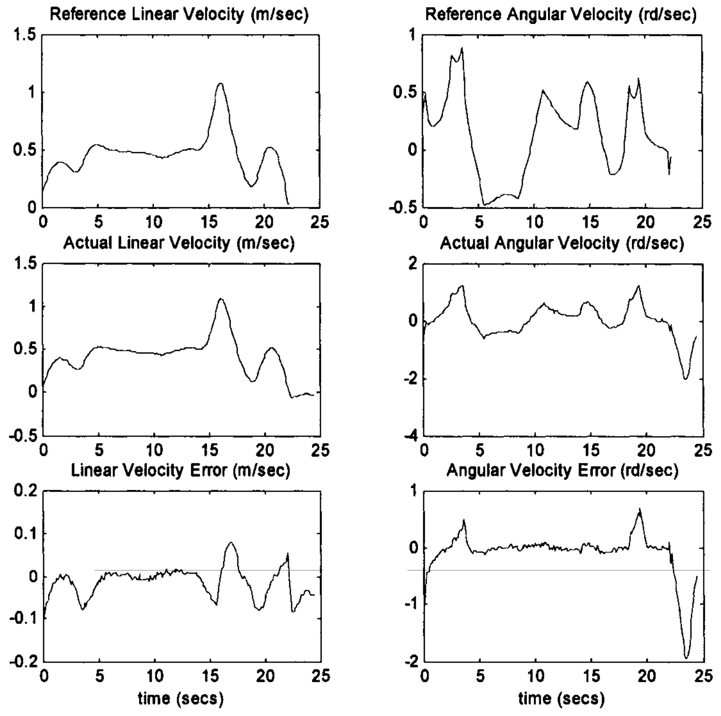


Figure 57 Velocities and velocity errors

5.4 Conclusion

The fuzzy logic controller for path following described in this chapter proved to be highly reliable and robust. It has several advantages over the classical Lyapunov based approach for trajectory following. The fuzzy controller controls robot motion along the discrete waypoints of an optimal path planned by the dynamic programming algorithm of Chapter 4. Position control is performed successively along the waypoints till the desired position that corresponds to the last waypoint on the path is reached.

In the Lyapunov derived approach, a continuous reference trajectory that connects the waypoints of the optimal path is calculated. The reference trajectory is a time dependent variable, and reference position, heading, and velocities are calculated at every time instant of the trajectory. The Lyapunov based controller is thus used to regulate three variables in addition to position. This imposes several constraints that result in a lower performance in position control relative to the fuzzy controller.

If the purpose of the navigation is to reach a desired destination along the optimal path without imposing any time constraints, the fuzzy controller is way more efficient and robust. The fuzzy inference rules are set to allow maximum speed while respecting navigation safety. If the robot is required to follow an exact trajectory that is function of time, the Lyapunov derived controller can be satisfactorily used. The performance of the Lyapunov based approach in real time is not as outstanding as the fuzzy position control approach due to the high friction of the wheels of the P3AT designed for rough terrain navigation being used in an indoor test environment. This becomes more evident in real-time implementation when controlling several variables. Furthermore, it is more recommended to control robot motion at the dynamic level if higher accuracy and exactitude are desired. The P3AT dynamics controls are not directly accessible. In the following chapter, the curvature velocity method for obstacle avoidance will be described.

CHAPTER 6

REAL TIME OBSTACLE AVOIDANCE

6.1 Introduction

Obstacle avoidance in real time is one of the crucial aspects in mobile robot navigation. The technique implemented in this project is the curvature velocity method (CVM) (Simmons, 1996). This technique is widely mentioned in the literature and is said to be reliable and computationally efficient. In this chapter, the CVM is described, and the modifications done to adapt it so that it can be implemented using sonar perception instead of a laser range finder are pointed out.

6.2 Obstacle Representation in Real Time

Since the curvature velocity method for obstacle avoidance (Simmons, 1996) deals mainly with curvatures, it is convenient that obstacles be modeled as circles represented by the coordinates of their center with respect to the local frame of reference (with the X and Y axes of the local frame introduced in Chapter 3 interchanged to stay in line with the article notation) and their radius (Figure 58).

The positions of the centers of obstacles are determined using a simple geometric interpretation of the readings of the two sonar arrays. The obstacles are enlarged by adding the radius *r* of the robot to the radius of the obstacles so as to accommodate for the width of the robot, since the motion of the robot is represented only by the center of the robot. The sonar array elements are fixed to the robot and their readings can only be interpreted to be in their corresponding discrete directions, unlike the laser range finder that scans in a plane and can achieve highly accurate models of obstacles surrounding the robot. Since the obstacles cannot be accurately determined, all obstacles will be assumed to have a radius of half a meter. In the case of a wall, when the robot gets close enough, the obstacles would be overlapping and the modifications to the original curvature velocity method would take this effect into account (more on this later).

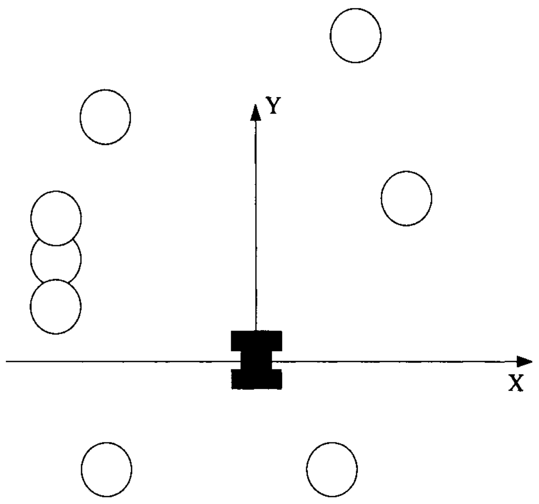


Figure 58 Robot local frame of reference and obstacles

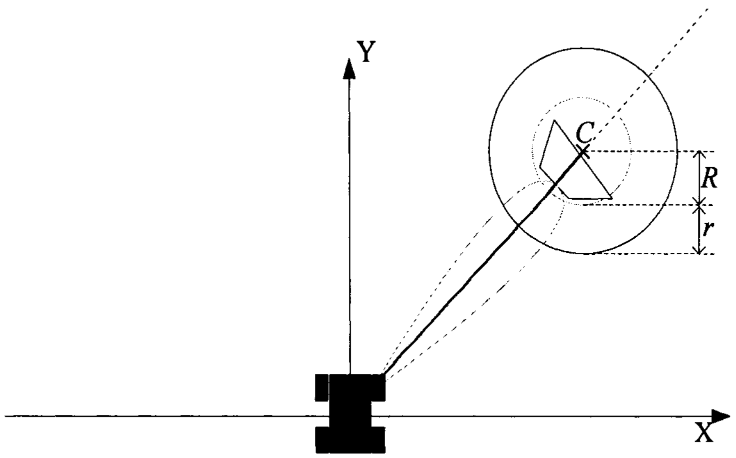


Figure 59 Obstacle representation in real time

If a sonar element returns a reading that is inferior to three meters, the obstacle would be assumed to be centered on the line in the direction of the sonar reading at a distance *R* away from the presumed point of impact of the sonar wave with the obstacle (Figure 59).

If a reading superior to three meters is returned by a sonar element, no obstacle will be associated with the reading. The significance of representing obstacles will be clearer as the curvature-velocity method is discussed in detail.

6.3 The Curvature-Velocity Method

The curvature-velocity method is an obstacle avoidance technique that optimizes a linear objective function in the velocity space of the robot with respect to the requirements and specifications mentioned in Simmons, (1996). The velocity space for a wheeled mobile robot (WMR) such as the P3-AT operating on a flat planar surface consists of the translational and rotational velocities. As indicated in the kinematic modeling in Chapter 3, a differentially driven WMR moves along a circle whose radius is the ratio of the translational velocity (TV) and the rotational velocity (RV) ($Radius = TV / RV$) in case TV and RV are constant. The curvature is defined to be the inverse of the radius. Each point in the velocity space maps to a curvature in the Cartesian space (Figure 60).

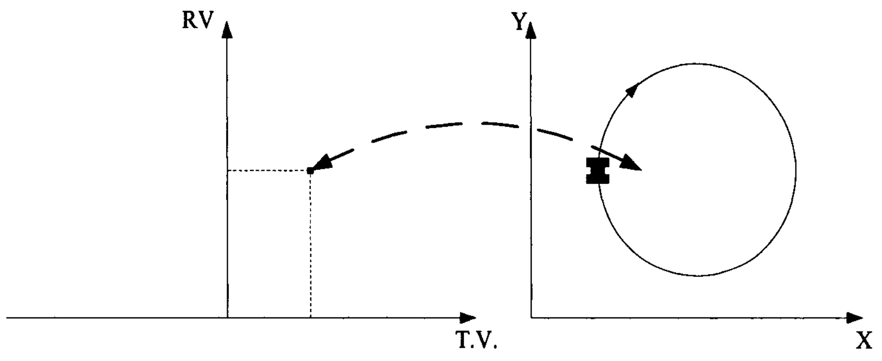


Figure 60 A point in velocity space maps to a curvature in Cartesian space

A positive curvature corresponds to clockwise motion of the robot relative to the global frame of reference. Furthermore, the set of TV and RV forming a line emanating from the origin (origin excluded) of the velocity space corresponds to the same curvature in Cartesian space due to the fact that the curvature is the ratio of TV to RV.

The distance d_c that the robot would travel before hitting an obstacle along a curvature c can be calculated using simple geometrical formulas. First it is to be noted from the geometry of the motion (Figure 61) that the center of the arc d_c lies on the abscissa of the local frame of reference. The angle θ of the arc is obtained and d_c can be calculated using:

$$\theta = \begin{cases} \tan^{-1}(y_i/(x_i - 1/c)), c < 0 \\ \pi - \tan^{-1}(y_i/(x_i - 1/c)), c > 0 \end{cases} \qquad \textbf{(6.1)}$$

$$d_c(c, obs) = \begin{cases} y_i, c = 0 \\ |1/c|\theta, c \neq 0 \end{cases} \qquad \textbf{(6.2)}$$

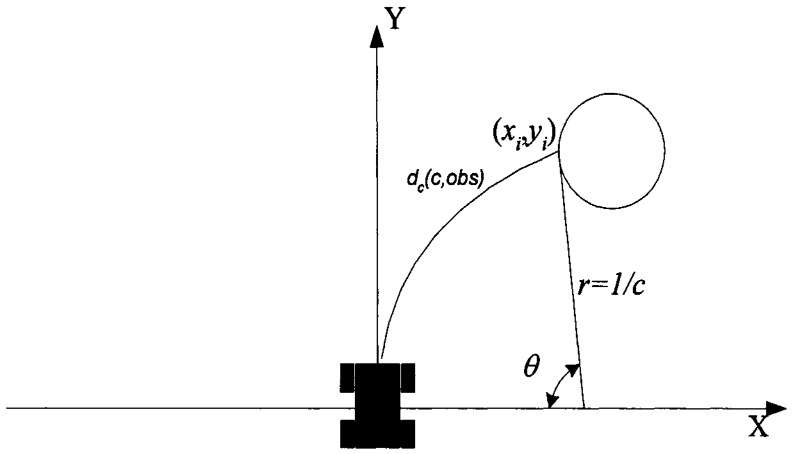


Figure 61 Calculation of travel distance before collision

The distance function for an obstacle *obs* in velocity space is defined as:

$$d_v(tv,rv,obs)=\begin{cases} d_c(rv/tv,obs), tv \neq 0 \\ \infty, otherwise \end{cases} \quad \textbf{(6.3)}$$

And the cumulative distance for a set of obstacles *OBS* affected by a (*tv*, *rv*) pair is given by:

$$D_L(tv,rv,OBS)=\min\left(L,\min_{obs\in OBS} d_v(tv,rv,obs)\right) \quad \textbf{(6.4)}$$

where L is a limiting distance (three meters as in Simmons (1996)). The distances D_L constitute a set of constraints in the velocity space. Another set of constraints is introduced by the translational and rotational velocity and acceleration limits:

$$\begin{aligned} & tv \le tv_{\max} \\ & tv \ge tv_{\min} \\ & rv \le rv_{\max} \\ & rv \le rv_{\max} \\ & rv \ge rv_{curv} - (ra_{\max} \times T_{accel}) \\ & rv \le rv_{curv} + (ra_{\max} \times T_{accel}) \\ & tv \le tv_{curv} + (ta_{\max} \times T_{accel}) \end{aligned} \quad \textbf{(6.5)}$$

where tv_{max}, tv_{min}, rv_{max}, and rv_{min} correspond to the minimum and maximum attainable translational and rotational velocities, rv_{curv} and tv_{curv} are the instantaneous rotational and translational velocities respectively, ra_{max} and ta_{max} are the maximum rotational and translational accelerations respectively, and T_{accel} is the sampling time which corresponds to the cycle time of the robot (see Chapter 2 on actions). An objective function *f(tv,rv)* that takes into consideration the different performance criteria is optimized while taking the constraints into consideration.

$$\begin{aligned} & f(tv,rv)=\alpha_1 speed(tv)+\alpha_2 dist(tv,rv)+\alpha_3 head(rv) \\ & speed(tv)=tv/tv_{\max} \\ & dist(tv,rv)=D_L(tv,rv,OBS)/L \\ & head(rv)=1-\left|\theta_g - rv\times T_c\right|/\pi \end{aligned} \quad \textbf{(6.6)}$$

The *speed* variable favors higher translational speeds, the *dist* variable favors longer travel along the curvature defined by *tv* and *rv* without colliding with obstacles and is thus dependent on D_L, and finally the *head* variable which favors moving towards the goal position, with θ_g being the angle of the goal heading measured in the local frame of reference and T_c being a time constant (taken as one second) and is used to determine the heading of the robot if it spins at *rv* for T_c seconds. Note that all the terms of the objective function are normalized to be between zero and one. The constant coefficients (α_1, α_2, and α_3) are chosen to have a sum of one. The optimal (*tv*, *rv*) pair is the pair that maximizes f in the allowable space. To find the optimal pair of (*tv*, *rv*) in the space limited by the constraints, some approximation technique such as simulated annealing can be used to determine the maximum of f.

6.4 Modifications for Real Time Implementation

The problem with the above explained approach is that it is not computationally efficient in real time. This problem is addressed by approximating D_L with a finite set of intervals with each interval being assigned a constant distance to an obstacle. The distance $d_v(c, obs)$ will be assumed constant between c_{min} and c_{max}, the minimum and maximum curvatures at the boundaries of the obstacle interval (Figure 62).

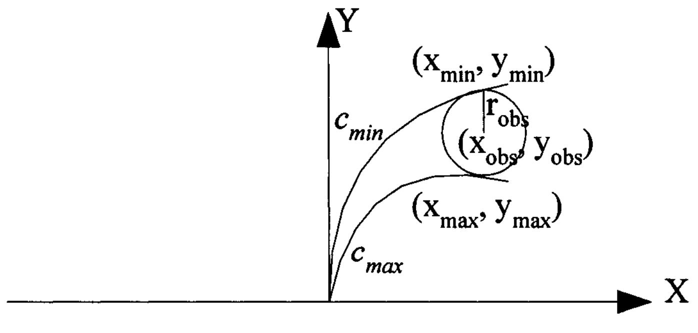


Figure 62 Curvatures at the boundaries of an obstacle

Given the x_{obs} and y_{obs}, the coordinates of the center of an obstacle, c_{min} and c_{max} can be determined by:

$$\begin{aligned} c_{min} &= 2\ (x_{obs} - r_{obs}) / (x_{obs}^2 + y_{obs}^2 - r_{obs}^2) \\ c_{max} &= 2\ (x_{obs} + r_{obs}) / (x_{obs}^2 + y_{obs}^2 - r_{obs}^2) \end{aligned} \qquad \textbf{(6.7)}$$

The circles formed by c_{min} and c_{max} are tangent to the circle representing the obstacle and intersect with it at (x_{min}, y_{min}) and (x_{max}, y_{max}) respectively, which are determined as follows:

$$\begin{aligned} x_{min} &= 1/c_{min} + \mathrm{abs}(1/c_{min}).(x_{obs} - 1/c_{min}) / (-r_{obs} + \mathrm{abs}(1/c_{min})) \\ y_{min} &= \mathrm{abs}(1/c_{min}).y_{obs} / (-r_{obs} + \mathrm{abs}(1/c_{min})) \\ x_{max} &= 1/c_{max} + \mathrm{abs}(1/c_{max}).(x_{obs} - 1/c_{max}) / (r_{obs} + \mathrm{abs}(1/c_{max})) \\ y_{max} &= \mathrm{abs}(1/c_{max}).y_{obs} / (r_{obs} + \mathrm{abs}(1/c_{max})) \end{aligned} \qquad \textbf{(6.8)}$$

The distance d_c can be calculated using equation (6.8) above and d_v is given by:

$$d_v = \begin{cases} \min(d_c(c_{\min}, obs), d_c(c_{\max}, obs)), & c_{\min} \le rv/tv \le c_{\max} \\ \infty, & ,otherwise \end{cases} \qquad \textbf{(6.9)}$$

D_{limit} can be determined through the use of the min-union of the intersection between obstacle intervals as will be explained in the next subsection.

6.4.1 Modification of Curvature Intervals

The curvature intervals are described by a curvature interval data structure ($<c_1,c_2>$, $d_{1,2}$) with $c_1 \le c_2$ based on some rules. The distance $d_{1,2}$ is the distance associated with the interval $<c_1, c_2>$. After the first curvature interval corresponding to the first obstacle is determined, every new interval that will be added will result in the modification of already available intervals or in the new interval being modified or in both consequences. Suppose a new curvature interval ($<c_{min}, c_{max}>$, d_i) is to be added and

($<c_1, c_2>$, $d_{1,2}$) is an already existing interval. In the case of overlapping intervals, the intervals are divided such that there is no intersection between any two intervals. The modifications are applied according the following set of rules:

- *If the intervals $<c_1, c_2>$ and $<c_{min}, c_{max}>$ are disjoint then no modifications to neither intervals is done.*
- *If $<c_{min}, c_{max}>$ contains $<c_1, c_2>$, $c_{\min} \le c_1$ and $c_2 \le c_{\max}$, then d1,2 is replaced by the minimum between $d_{1,2}$ and d_i.*
- *If $<c_{min}, c_{max}>$ is contained by $<c_1, c_2>$ ($c_1 \le c_{\min}$ and $c_{\max} \le c_2$) and $d_i<d_{1,2}$, , then $<c_1,c_2>$ is divided into three intervals: ($<c_1, c_{min}>$, $d_{1,2}$), ($<c_{min},c_{max}>$, d_i), and ($<c_{max},c_2>$, $d_{1,2}$). If $d_i>d_{1,2}$ then $<c_{min}, c_{max}>$ will be eliminated and not compared to the other existing intervals.*
- *If $<c_1, c_2>$ and $<c_{min}, c_{max}>$ overlap with $c_1 \le c_{\min}$, then the two intervals are modified as follows:*
    - *If $d_i < d_{1,2}$, then $<c_1,c_2>$ is replaced by ($<c_1, c_{min}>$, $d_{1,2}$) and $<c_{min},c_{max}>$ is not changed.*
    - *If $d_{1,2} < d_i$, then $<c_{min},c_{max}>$ is replaced by ($<c_2, c_{max}>$, $d_{1,2}$) and $<c_1,c_2>$ is not changed.*
- *If $<c_1, c_2>$ and $<c_{min}, c_{max}>$ overlap with $c_{\max} \le c_2$, then the two intervals are modified as follows:*
    - *If $d_i < d_{1,2}$, then $<c_1, c_2>$ is replaced by ($<c_{max}, c_2>$, $d_{1,2}$) and $<c_{min},c_{max}>$ is not changed.*
    - *If $d_{1,2} < d_i$, then $<c_{min}, c_{max}>$ is replaced by ($<c_2, c_{max}>$, d_i) and $<c_1,c_2>$ is not changed.*

The interval $<c_{min}, c_{max}>$ is compared to all the existing intervals, and this is done to all obstacles. At the end, the distances of each of the intervals would correspond to the D_{limit} distance. The initial curvature is always taken to be ($<-\infty,\infty>$,L), and other intervals are added afterwards. The significance of this approach and its efficiency in calculating the optimal (*tv*, *rv*) pair will be clearer later on.

In the example of Figure 63, the use of these rules is illustrated. The initial curvature is ($<-\infty,\infty>$,L). Starting with the first obstacle interval $<c_2,c_1>$, $d_{v2,1}$ corresponds to the minimum between d_1 and d_2 corresponding to c_1 and c_2 respectively, which is d_1 in this case. The second rule applies in this case when comparing (<c_2,c_1>, d_1) with (<-∞,∞>,L). So we will now have three intervals: (<-∞,c_2>,L), (<c_2,c_1>, d_1), and (<c_1,∞>,L). The interval <c_4,c_3> is added with d_3 taken as $d_{v4,3}$ (using $min(d_3,d_4)$). The intervals <c_2,c_1> and <c_4,c_3> overlap, having $d_{v4,3}<d_{v2,1}$ along with $c_3<c_1$, thus the conditions of the first criterion of the last rule is satisfied. Hence (<c_4,c_3> ,d_3) will not be modified while interval (<c_2,c_1>, d_1) would be replaced by (<c_3,c_1>, d_1).

The interval <c_6,c_5> is to be added. Compared to <c_3,c_1> and to <c_4,c_3>, it is found to be disjoint. The final set of intervals would be (<-∞,c_6>,L), (<c_6,c_5>, d_6), (<c_5,c_4>, L), (<c_5,c_4>, d_3), (<c_3,c_1>, d_1) and (<c_1,∞>,L).

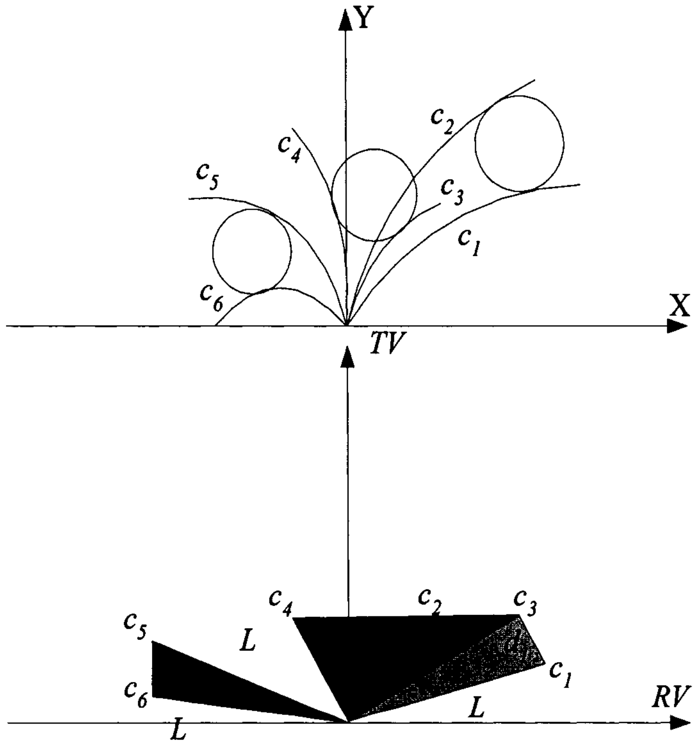


Figure 63 Curvature intervals and piecewise constant approximation of D_{limit}

6.4.2 Optimization of the Objective Function

After all the curvature intervals are obtained, the optimal pair (*tv*,*rv*) lies in the region of the intervals bounded by the velocity and acceleration constraints. Since the objective function increases linearly with *tv*, the optimal (*tv*,*rv*) of the objective function lies on the boundaries drawn by the constraints. So the objective function will only be calculated at the vertices on the upper boundaries (Figure 63) of each of the curvature intervals. The objective function is also calculated at the vertex that corresponds to moving directly towards the goal position, which lies at $rv=\theta_g/T_c$. Then the (*tv*,*rv*) pair that yields the maximum value of the objective function would be the optimal set of commands to control the robot motion.

The objective and its variables are as shown below:

$$\begin{aligned}
&f(tv,rv) = \alpha_1 speed(tv) + \alpha_2 dist(tv,rv) + \alpha_3 head(rv) \\
&speed(tv) = tv / tv_{\max} \\
&dist(tv,rv) = D_L(tv,rv,OBS)/L \\
&head(rv) = 1 - \left|\theta_g - rv \times T_c\right| / \pi
\end{aligned} \tag{6.10}$$

6.5 Real Time Implementation

The implementation in real time was done using modular blocks so as to facilitate debugging and testing. Sonar range acquisition was done by using the member function **ArRobot::getSonarRange ( )** in **ActionGo::fire( )** in the *'main.cpp'* file. The calculation of obstacle positions and the establishment of a list of obstacles are implemented in *'SonarCalc.h'*. The list of obstacles is passed to a **CurvVel** object. The **CurvVel** class contains member functions to calculate the curvature velocity intervals and determine the optimal solution. There was a problem using the standard libraries in respecting the 100ms cycle time of the robot control system; however the calculation time has been reduced to an acceptable level through a special use of pointers.

6.6 Conclusion

In this chapter, the curvature velocity method (CVM) for obstacle avoidance was described and adapted for implementation in real time. This method optimizes an objective function in the curvature velocity space with velocity and acceleration constraints. The CVM was coded and tested in real time. The approach however was not fully tested and debugged due to time limitations. The code is functional with a probability close to 60% of the cases when the robot passes next to an obstacle. In the rest of the cases the robot hits the obstacle due to unidentified bugs. In the next chapter a navigation strategy is proposed to combine all three navigation aspects developed throughout this project.

CHAPTER 7

PROPOSED NAVIGATION STRATEGY

7.1 Introduction

To obtain a fully autonomous navigation system, path planning, path tracking, and obstacle avoidance need to be integrated into a navigation strategy that coordinates and synchronizes them. In this chapter, a hybrid control architecture in the form of a state machine is proposed.

7.2 Navigation Strategy

The purpose of a navigation strategy is to coordinate the different behaviors that execute the desired task. The desired task in this project is to displace the robot from its current position to another position specified by the user, while avoiding obstacles that might get in its way and at the least possible calculation cost. For this end, the navigation controller constantly monitors the environment and the robot posture and activates the action that best leads to the execution of the desired task. At first, if the environment is known then the dynamic programming algorithm discussed in Chapter 4 can be used to find an optimal trajectory. Otherwise, the robot must be capable of finding its way to the goal position while avoiding obstacles and making a model of the environment for future use.

As long as the robot is connected to its client, the sonar signals (exteroceptive sensors) will continuously be read and used to update the model of the environment in the proximity of the robot. The position and velocities (proprioceptive sensors) are also continuously checked. In case the robot was in navigation mode, the information coming from the proprioceptive and exteroceptive sensors is used to switch command to the appropriate mode of action. The four states of the finite state machine are used to represent four modes of action. The modes are:

Stop mode: The robot is stopped (both translational and rotational velocities are set to zero).

Path Planning mode: Command switches to this mode only after being at stop mode, since it might take a few seconds to plan a new trajectory, and it is safer if the robot is stationary when a new trajectory is being planned. In this project the dynamic programming algorithm of Chapter 4 is used in the path planning mode.

Path Tracking Mode: In this mode, the robot tracks the trajectory that was planned when the robot was in the path planning mode.

Obstacle Avoidance mode: In this mode, the curvature velocity method discussed in Chapter 6 is activated. This method integrates both the seek goal and avoid obstacle behaviors.

The model of the environment is represented by the cost matrix (see Chapter 4), and will be updated based on sonar readings and robot position and posture with respect to the environment. Since the robot will be traveling in a two dimensional environment, the obstacles would be modeled as in Chapter 6, and the matrix elements included in the circle used to represent the obstacle would be assigned very high values. (Figure 64).

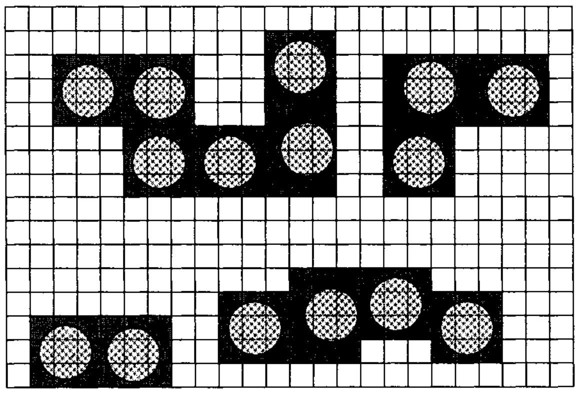


Figure 64 Matrix elements eclipsed by obstacles

The robot at the beginning is usually in the stop mode waiting for commands. When the user specifies a destination, the application program checks if a model of the environment is available and if the position of the robot with respect to the model is known. Both conditions are satisfied, command switches to the path planning mode. The optimal path is determined, and command switches to the path tracking mode, and remains so until an obstacle lying on one of the next three waypoints on the path is detected or the destination was reached.

If an obstacle is detected, command switches to the obstacle avoidance mode, and the intermediate goal would be set to the waypoint where no obstacle is present. When the intermediate goal position is reached and no obstacle stands in the way, command is switched back to trajectory tracking mode. Throughout the navigation process, multiple switches can occur between trajectory tracking mode and obstacle avoidance.

The application program keeps track of the trajectory and constantly checks if the robot passes through the same region twice to check if the robot is stuck in a local minimum. In such a case, the robot would be put in the stop mode, and then in trajectory planning mode to plan a new optimal path and then switches to trajectory tracking mode. This process will continue until the robot reaches goal position. In case the goal position cannot be reached, the robot is put in stop mode and will wait for user commands.

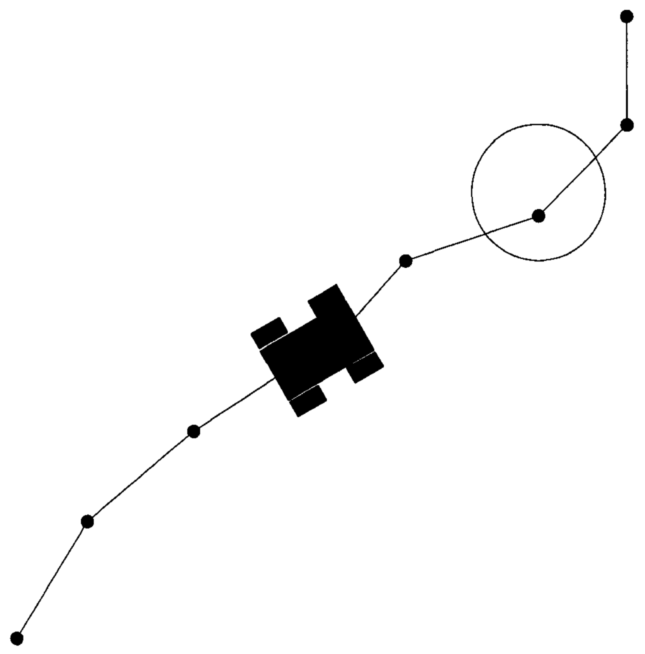


Figure 65 Obstacle on the path of the robot

The state machine diagram in Figure 66 summarizes the whole process of the navigation strategy. The implementation of the navigation strategy as a state machine simplifies the navigation procedure and is very efficient and practical.

If no model of the environment was available at the beginning, the robot is initially in the stop mode and would then switch to the obstacle avoidance mode. As the robot starts navigating, a model of the environment would be established.

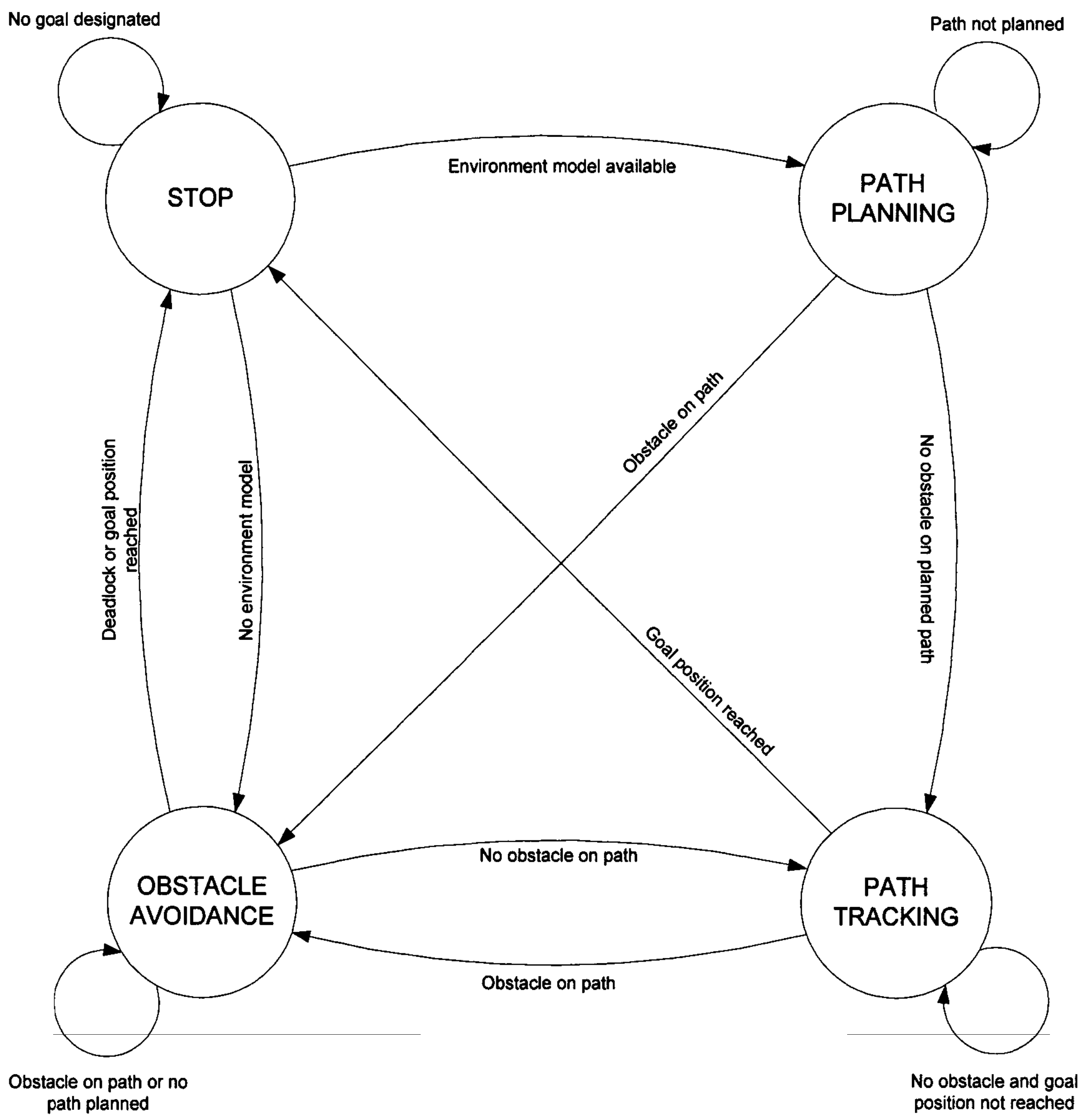


Figure 66 Navigation strategy state machine

The application program constantly keeps checking for a local minimum, and in case one is detected, the robot is put in stop mode and a trajectory is planned using the partially available model of the environment. All the regions not yet modeled will be assumed obstacle free. The robot would then switch to the trajectory tracking mode, and the process continues.

7.3 Conclusion

The navigation strategy proposed in this chapter works supposedly well, but has some limitations due to localization and positioning errors due to slippage. This error increases as the robot moves and the model of the environment would become increasingly erroneous. A global positioning system (GPS) device and a compass would solve the problem of localization.

CONCLUSION

The work presented in this research project is a solid foundation for a fully autonomous navigation system for a wheeled mobile robot. The navigation system that was proposed is flexible and can be further expanded to include more functionalities and to integrate the usage of additional sensors and instruments as well as additional computational resources. The technical details have been fully documented taking into consideration the possibility of further development and research. Three functionalities crucial to an autonomous navigation system for a wheeled mobile robot have been developed and implemented in real time on the P3-AT mobile robot. The localization and sonar perception routines available with the ARIA interface have been used without any modifications. The main focus of this project was on path planning, path tracking, and obstacle avoidance implemented using an application program that controls the robot and is run on a laptop computer that is connected to the robot through a wireless serial communication link.

The iterative dynamic programming algorithm for path planning is a sound contribution in the field of optimization algorithmics. It has some interesting advantages over the widely used A* search algorithm. The dynamic programming algorithm determines the optimal solutions for all the nodes of the graph towards the goal node, which is advantageous in a multi-robot environment. This is also advantageous if the robot deviates from its original path due to a real time dynamic obstacle, since there would be no need to plan a new optimal path from the current position in real time. The dynamic programming algorithm can be executed on several computational resources in parallel while the A* algorithm can only be executed sequentially. The dynamic programming algorithm is generic and its performance is not dependent on the choice of any heuristic function as is the case with the A* algorithm. Its only disadvantage with respect to the A* is in the computation time which is significantly much higher if implemented sequentially on a single computational unit. A more exhaustive study can be done to assess the execution of the dynamic programming algorithm on parallel computational resources.

The environment is represented by a graph whose nodes represent some discrete positions sampled at regular intervals. The nodes are linked by directional arrows associated with a cost of traversability. If the robot operates in a 2-D indoor environment, the cost would be a function of the distance between the positions represented by two nodes. If the environment is a rugged 3D terrain, a methodology is needed to assess the traversability of the terrain and calculate the cost accordingly. The cost would depend on terrain characteristics such as slopes and terrain roughness in addition to distance. The cost of traveling between two nodes needs not be the same in both directions. To minimize calculations, nodes pertaining to positions inside obstacles or unsafe regions are given a very high cost of traversability and are eliminated using a recursive function similar to the branch and bound technique.

The fuzzy logic controller for path following was designed based on an analogy with a human driver. The path is a discrete set of waypoints in the form of a linked list whose first and last elements are the initial and destination positions respectively. As is usually the case, it is the optimal path obtained by using the iterative dynamic programming technique. The controller drives the robot at the proximity of those discrete waypoints without requiring a continuous reference trajectory. This characteristic enhances the reliability and the robustness when used for real life situations. The results of real time implementation proved this controller to have a high performance.

A classical controller for trajectory following derived from a Lyapunov function of errors in position and heading and their derivatives was implemented and tested in real time. The reference trajectory is continuous and is determined by interpolating the discrete set of waypoints by a cubic spline. The reference trajectory is a function of time. Reference velocities and heading are derived from the reference trajectory. The controller is used to regulate the robot position as well as velocities and heading. With this approach, the robot would be actually following a virtual reference robot.

Although there is no solid basis to compare the two approaches analytically, the fuzzy controller has proved to be more reliable and robust. It can be made to follow a trajectory that is more precise by inserting intermediate waypoints between the waypoints. The performance of the classical controller varies if different reference trajectories are used. The gains that yield optimal performance with a reference trajectory that is a straight line are not necessarily the same to have an optimal performance of a sinus reference trajectory. If the reference path changes in real time, the fuzzy controller would not require cubic splines to interpolate the waypoints to obtain a continuous reference trajectory as is the case with the classical controller. The classical controller can be effectively used if an application requires that the robot tracks a trajectory with a predetermined timing.

The curvature velocity method (CVM) for obstacle avoidance is valid theoretically and has been implemented in real time. This approach has the advantage that the robot can be used to explore an unknown environment. Due to time limitations, the program could not be fully tested and debugged, but it works in almost 60% of the cases. Simpler methods for obstacle avoidance can be more easily implemented nevertheless.

Several architectures have been developed in the literature to coordinate all aspects and behaviors of navigation systems. For this project, a state machine with four modes comparable to the hybrid architecture has been proposed to control the robot.

RECOMMENDATIONS

In this project, an autonomous navigation system was developed, partially implemented, and tested on a flat surface indoor environment. While scanning through the mobile robot navigation, a lot of inspiring ideas and suggestions for the development of interesting applications and take maximum advantage of the P3-AT capabilities come to mind. The P3-AT has the capability of being equipped with a multitude of sensors and can be used in rough and unstructured terrain navigation outdoors.

For navigation on a rough terrain, the trajectory planning algorithm needs not to be changed. However a technique is required to assess terrain and generate a cost matrix for a 3D terrain. Refer to section 2 of Chapter 4 for more detail.

For trajectory following, it is recommended to add a terrain smoothness factor as input to the fuzzy controller to take account of terrain irregularities. For this purpose a camera and a 3D gyroscope can be used to make an assessment of the terrain while making use of an image processing technique.

Furthermore, the navigation aspects developed throughout this project can be adapted for some specific applications and tasks. For example a manipulator can be added on top the robot to perform certain tasks, and this might require some modifications in path planning, path tracking, and obstacle avoidance, as well as in the overall navigation strategy.

The usage of a camera and a laser range finder can be useful for a precise modeling of the environment as well as to track a certain object, and this is important for applications in a multi-robot environment such as a soccer team or robot convoys.

APPENDIX 1

Client Driver Program

This is the driver program based on *actionExample* that is provided with the ARIA package examples. This program is the driver and the interface between the robot control techniques and the robot or simulator. The code of this program is often referred to in the explanation of the ARIA interface in Chapter 2.

```
#include "Aria.h"
#include "CurvVel.h"
#include <fstream>
#include <iostream>
#include <list>
#include <vector>

using namespace std;

/*
      This demonstrates how to make actions and how to use them …
*/


class ActionGo : public ArAction
{
public:
      // constructor, sets myMaxSpeed and myStopDistance
      ActionGo(double maxSpeed, double stopDistance);
      // destructor, its just empty, we don't need to do anything
      virtual ~ActionGo(void) {delete CurvVelPtr;};
      // fire, this is what the resolver calls to figure out what this
action wants
      virtual ArActionDesired *fire(ArActionDesired currentDesired);
      // sets the robot pointer, also gets the sonar device
      virtual void setRobot(ArRobot *robot);
protected:
      // this is to hold the sonar device form the robot
      ArRangeDevice *mySonar;
      // what the action wants to do
      ArActionDesired myDesired;

      // initialize PathTracker;
      //PathTracker* PathTrackerPtr;
      CurvVel* CurvVelPtr;
      // maximum speed
      double myMaxSpeed;
      // distance to stop at
      double myStopDistance;
      ArTime myTime;
      short firstFire;
};
```

```
/*
This is the constructor, note the use of constructor chaining with the
ArAction... also note how it uses setNextArgument, which makes it so
that
other things can see what parameters this action has, and set them.
It also initializes the classes variables.
*/
ActionGo::ActionGo(double maxSpeed, double stopDistance) :
ArAction("Go")
{
      mySonar = NULL;
      myMaxSpeed = maxSpeed;
      myStopDistance = stopDistance;
      setNextArgument(ArArg("maximum speed", &myMaxSpeed, "Maximum
speed to go."));
      setNextArgument(ArArg("stop distance", &myStopDistance, "Distance
at which to stop."));
      CurvVelPtr = new CurvVel();
      firstFire=1;
}

/*
Sets the myRobot pointer (all setRobot overloaded functions must do
this),
finds the sonar device from the robot, and if the sonar isn't there,
then it deactivates itself.
*/
void ActionGo::setRobot(ArRobot *robot)
{
      myRobot = robot;
      mySonar = myRobot->findRangeDevice("sonar");
      if (mySonar == NULL)
            deactivate();
}


/*
This fire is the whole point of the action.
*/
ArActionDesired *ActionGo::fire(ArActionDesired currentDesired)
{
      double timeD;

      if (firstFire==1){
            myTime.setToNow( );
            firstFire=0;
      }

      timeD=(myTime.mSecSince( ));

      // reset the actionDesired (must be done)
      myDesired.reset( );
```

```
        // if the sonar is null we can't do anything, so deactivate
        if (mySonar == NULL)
        {
              deactivate();
              return NULL;
        }

        double myCurrentHeading = myRobot -> getTh( )*pi/180 ;


        VelParameters* myCurrentVel=new VelParameters;

        myCurrentVel->velocity = ( myRobot -> getVel( ) ) / 1000 ;
        myCurrentVel->omega = ( myRobot -> getRotVel( ) ) * pi/180 ;



        Position myCurrentPosition ( (myRobot -> getX( ))/1000,(myRobot -
>getY())/1000);

        Position* myCurrentPositionPtr = &myCurrentPosition ;

        Position goalPosition(100,0);

        if ( myCurrentPosition.findDistance(goalPosition) < .5)
        {
              myRobot->setVel2 (0,0);
              return &myDesired;
        }


        cout<<"myCurrentPosition: \t"<<myCurrentPosition;
        cout<<"myCurrentHeading: \t"<<myCurrentHeading<<"\t
myCurrentVelocity
             \t"<<myCurrentVel->velocity<<endl;

        static VelParameters* myDesiredVel;
        static VelParameters myDesiredV;


        static vector<double> range(16);

        // get the range of all sonars 0 through 15

        //double myRobotRadius=myRobot->getRobotRadius( )/2;

        //cout<<endl<<myRobotRadius<<endl;

        for(int i=0;i<16;i++)
        {
             range[i] = ( myRobot->getSonarRange(i) ) * .001 +
myRobotRadius  ;
```

```
			cout<<"Range["<<i<<"] = "<<range[i]*1000<<endl;
		}

		static dVInputs mydVInputs;

		mydVInputs.currentHeading = myCurrentHeading;
		mydVInputs.currentPosition = myCurrentPosition;
		mydVInputs.currentVel=(*myCurrentVel);
		mydVInputs.sonarReadings = range;

		static dVInputs* mydVInputsPtr = & mydVInputs;



		// set goal position
		CurvVelPtr->setGoalPosition(goalPosition);

		myDesiredV = CurvVelPtr->determineVels(mydVInputsPtr);

		myDesiredVel = &myDesiredV;

		static double leftVelocity, rightVelocity;

		leftVelocity = myDesiredV.velocity - (myDesiredV.omega)*490/2;
		rightVelocity = myDesiredV.velocity +(myDesiredV.omega)*490/2;

		cout<<"velocity \t"<<myDesiredVel->velocity<<"omega \t"<<
			myDesiredVel->omega<<endl;

		cout<<"leftVelocity \t"<<leftVelocity<<"\t rightvelocity
		\t"<<rightVelocity<<endl;

		myRobot->setVel2 ( leftVelocity , rightVelocity );/**/


		delete myCurrentVel;
		// return a pointer to the actionDesired, so resolver knows what
to do
		return &myDesired;
}

int main(int argc,char** argv)
{
		ofstream trajectory ("Trajectory.txt",ios::trunc);
		ofstream RefTrajectory ("RefPosition.txt",ios::trunc);
		ofstream RefVels ("RefVels.txt",ios::trunc);
		ofstream RobotVels ("RobotVels.txt",ios::trunc);

		// the robot
		ArRobot robot;
		// the sonar device
		ArSonarDevice sonar;
```

```
      // some stuff for return values
      std::string str;

      // the behaviors from above, and a stallRecover behavior that
uses defaults

      ActionGo go(500, 350);
      ArActionStallRecover recover;


      // this needs to be done
      Aria::init( );
      ArSimpleConnector connector(&argc, argv);
      connector.parseArgs ( );

      if (argc > 1)
      {
            connector.logOptions( );
            exit(1);
      }


      // add the range device to the robot, you should add all the
range
      // devices and such before you add actions
      robot.addRangeDevice(&sonar);

      // do a blocking connect, if it fails exit
      if (!connector.connectRobot(&robot))
      {
            printf("Could not connect to robot... exiting\n");
            Aria::shutdown();
            return 1;
      }

      // enable the motors, disable amigobot sounds
      robot.comInt(ArCommands::ENABLE, 1);
      robot.comInt(ArCommands::SOUNDTOG, 0);

      // add our actions in a good order, the integer here is the
priority,
      // with higher priority actions going first
      robot.addAction ( &recover, 100);
      robot.addAction ( &go, 50);

      // run the robot, the true here is to exit if it loses connection
      robot.run(true);

      // now just shutdown and go away
      Aria::shutdown( );
      return 0;

}
```

APPENDIX 2

Fuzzy Control Language Files (FCL)

2.1 Fuzzy Control Language (FCL)

This appendix contains the three FCL files for each of the three fuzzy modules described in Chapter 5. Comments are included between '(* *)' characters. The VAR_INPUT and VAR_OUTPUT statements are indicators to locate the inputs and outputs respectively. The statement FUZZIFY followed by an input name is an indicator to the definition of the member functions of this input. The statement DEFUZZIFY followed by an output name is an indicator to the defuzzification method used to defuzzify the weights obtained from the inference rules. And finally the rules are added after the statement RULEBLOCK.

2.2 Look Ahead Curvature (LAC) FCL

```
FUNCTION_BLOCK

VAR_INPUT
      alpha1      REAL; (* RANGE(0 .. 180) *)
      alpha2      REAL; (* RANGE(0 .. 180) *)
END_VAR

VAR_OUTPUT
      Curv  REAL; (* RANGE(0 .. 6) *)
END_VAR

FUZZIFY alpha1
      TERM Straight1 :=(0,0) (0, 1) (20, 0);
      TERM High1 := (1, 0) (20, 1) (95, 0) ;
      TERM VeryHigh1 := (85, 0) (140, 1) (180, 0) ;
END_FUZZIFY

FUZZIFY alpha2
      TERM Straight2 :=(0,0) (0, 1) (30, 0);
      TERM High2 := (10, 0) (20, 1) (95, 0) ;
      TERM VeryHigh2 := (80, 0) (140, 1) (180, 0) ;
END_FUZZIFY

FUZZIFY Curv
      TERM NoCurvature := 0 ;
      TERM Moderate := 1.5 ;
      TERM cHigh := 4 ;
      TERM cVeryHigh := 5 ;
END_FUZZIFY
```

```
DEFUZZIFY Curv
      AND:PROD;
      METHOD: COGS;
END_DEFUZZIFY

RULEBLOCK first
      AND:PROD;
      ACCUM:MAX;
      RULE 0: IF Straight1 AND Straight2 THEN NoCurvature;
      RULE 1: IF Straight1 AND High2 THEN Moderate;
      RULE 2: IF Straight1 AND VeryHigh2 THEN cHigh;
      RULE 3: IF High1 AND Straight2 THEN cHigh;
      RULE 4: IF High1 AND High2 THEN cHigh;
      RULE 5: IF High1 AND VeryHigh2 THEN cHigh;
      RULE 6: IF VeryHigh1 AND Straight2 THEN cVeryHigh;
      RULE 7: IF VeryHigh1 AND High2 THEN cVeryHigh;
      RULE 8: IF VeryHigh1 AND VeryHigh2 THEN cVeryHigh;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

2.3 V-Controller FCL

```
FUNCTION_BLOCK

VAR_INPUT
      Curvature   REAL; (* RANGE(0 .. 5.5) *)
      dR    REAL; (* RANGE(0 .. 4000) *)
      dPhi  REAL; (* RANGE(-180 .. 180) *)
      CurrentVelocity   REAL; (* RANGE(0 .. 1000) *)
END_VAR

VAR_OUTPUT
      Velocity    REAL; (* RANGE(0 .. 1000) *)
END_VAR

FUZZIFY Curvature
      TERM Low :=(-0.1,0) (0, 1) (1.5, 0);
      TERM High := (1, 0) (2, 1) (5.5, 0) ;
      TERM VeryHigh := (3, 0) (4, 1) (5.5, 1) (5.5, 0) ;
      TERM NoMatterWhat := (-0.1,0) (0,1) (5.5,1) (5.5,0) ;
END_FUZZIFY

FUZZIFY dR
      TERM Near  :=(0, 0) (0, 1) (100, 0);
      TERM Close := (20, 0) (200, 1) (1400, 0) ;
      TERM Far := (150, 0) (500, 1) (7000, 1) (7000, 0) ;
      TERM NoMatterWhat := (0,0) (0,1) (7000,1) (7000,0) ;
```

```
END_FUZZIFY

FUZZIFY dPhi
      TERM nVeryHigh :=(-181,0) (-181,1) (-100, 1) (-50, 0);
      TERM nHigh := (-100, 0) (-10, 1) (-1, 0) ;
      TERM Small := (-10, 0) (0, 1) (10, 0) ;
      TERM VeryHigh :=(50,0) (100, 1) (180, 1) (180, 0) ;
      TERM High := (1, 0) (10, 1) (100, 0) ;
      TERM NoMatterWhat := (-181,0) (-181,1) (181,1) (181,0) ;
END_FUZZIFY

FUZZIFY CurrentVelocity
      TERM Low := (0, 0) (0, 1) (200, 0) ;
      TERM VeryHigh :=(400,0) (500, 1) (1000, 0);
      TERM High := (100, 0) (250, 1) (1000, 0) ;
      TERM NoMatterWhat := (0,0) (0,1) (1000,1) (1000,0) ;
END_FUZZIFY

FUZZIFY Velocity
      TERM Stop := 0 ;
      TERM S100 := 100 ;
      TERM S200 := 200 ;
      TERM S300 := 300 ;
      TERM S400 := 400 ;
      TERM S500 := 500 ;
      TERM S600 := 600 ;
      TERM S700 := 700 ;
      TERM S800 := 800 ;
      TERM NoValue := (900,0) (901,0) (902,0);
END_FUZZIFY

DEFUZZIFY Velocity
      AND:PROD;
      METHOD: COGS;
END_DEFUZZIFY

RULEBLOCK first
      AND:PROD;
      ACCUM:MAX;

      RULE 0: IF High AND NoMatterWhat AND NoMatterWhat AND
NoMatterWhat THEN S300;
      RULE 1: IF VeryHigh AND NoMatterWhat AND NoMatterWhat AND
NoMatterWhat THEN S200;
      RULE 2: IF NoMatterWhat AND Far AND Small AND NoMatterWhat THEN
S700;
      RULE 3: IF High AND Far AND Small AND NoMatterWhat THEN S300;
      RULE 4: IF VeryHigh AND Far AND Small AND NoMatterWhat THEN S300;
      RULE 5: IF NoMatterWhat AND NoMatterWhat AND High AND VeryHigh
THEN S300;
      RULE 6: IF NoMatterWhat AND NoMatterWhat AND nHigh AND VeryHigh
THEN S300;
```

```
      RULE 7: IF NoMatterWhat AND NoMatterWhat AND VeryHigh AND
VeryHigh THEN S200;
      RULE 8: IF NoMatterWhat AND NoMatterWhat AND nVeryHigh AND
VeryHigh THEN S200;
      RULE 9: IF NoMatterWhat AND NoMatterWhat AND High AND
NOT(VeryHigh) THEN S300;
      RULE 10: IF NoMatterWhat AND NoMatterWhat AND nHigh AND
NOT(VeryHigh) THEN S300;
      RULE 11: IF NoMatterWhat AND NoMatterWhat AND VeryHigh AND
NOT(VeryHigh) THEN S200;
      RULE 12: IF NoMatterWhat AND NoMatterWhat AND VeryHigh AND
NOT(VeryHigh) THEN S200;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

2.4 ω-Controller FCL

```
FUNCTION_BLOCK

VAR_INPUT
      Curvature   REAL; (* RANGE(0 .. 5.5) *)
      dR    REAL; (* RANGE(0 .. 4000) *)
      dPhi  REAL; (* RANGE(-180 .. 180) *)
      CurrentVelocity   REAL; (* RANGE(0 .. 1000) *)
END_VAR

VAR_OUTPUT
      Omega REAL; (* RANGE(-35 .. 35) *)
END_VAR

FUZZIFY Curvature
      TERM Low :=(-0.1,0) (0, 1) (1.5, 0);
      TERM High := (1, 0) (2, 1) (5.5, 0) ;
      TERM VeryHigh := (3, 0) (4, 1) (5.5, 1) (5.5, 0) ;
      TERM NoMatterWhat := (0,0) (0,1) (5.5,1) (5.5,0) ;
END_FUZZIFY

FUZZIFY dR
      TERM Near  :=(0, 0) (0, 1) (100, 0);
      TERM Close := (20, 0) (700, 1) (1400, 0) ;
      TERM Far := (1000, 0) (1500, 1) (7000, 1) (7000, 0) ;
      TERM NoMatterWhat := (0,0) (0,1) (7000,1) (7000,0) ;
END_FUZZIFY

FUZZIFY dPhi
      TERM nVeryHigh :=(-181,0) (-181,1) (-100, 1) (-50, 0);
      TERM nHigh := (-100, 0) (-10, 1) (-1, 0) ;
      TERM Small := (-10, 0) (0, 1) (10, 0) ;
      TERM VeryHigh :=(50,0) (100, 1) (181, 1) (181, 0);
      TERM High := (1, 0) (10, 1) (100, 0) ;
      TERM NoMatterWhat := (-181,0) (-181,1) (181,1) (181,0) ;
END_FUZZIFY

FUZZIFY CurrentVelocity
      TERM Low := (0, 0) (0, 1) (200, 0) ;
      TERM VeryHigh :=(300,0) (400, 1) (1000, 0);
      TERM High := (100, 0) (250, 1) (1000, 0) ;
      TERM NoMatterWhat := (0,0) (0,1) (1000,1) (1000,0) ;
END_FUZZIFY

FUZZIFY Omega
      TERM n35 := -35 ;
      TERM n30 := -30 ;
      TERM n25 := -25 ;
      TERM n20 := -20 ;
      TERM n15 := -15 ;
      TERM n10 := -10 ;
      TERM n5 := -5 ;
      TERM Stop := 0 ;
      TERM p5 := 5 ;
      TERM p10 := 10 ;
      TERM p15 := 15 ;
      TERM p20 := 20 ;
      TERM p25 := 25 ;
      TERM p30 := 30 ;
      TERM p35 := 35 ;
      TERM NoValue := (-10,0) (0,0) (0,0);
END_FUZZIFY

DEFUZZIFY Omega
      AND:PROD;
      METHOD: COGS;
END_DEFUZZIFY

RULEBLOCK first
      AND:PROD;
      ACCUM:MAX;
      RULE 0: IF NoMatterWhat AND Near AND NoMatterWhat AND
NoMatterWhat THEN Stop;
      RULE 1: IF NoMatterWhat AND NoMatterWhat AND Small AND
NoMatterWhat THEN Stop;
      RULE 2: IF Low AND Far AND Small AND NoMatterWhat THEN Stop;
      RULE 3: IF NoMatterWhat AND NoMatterWhat AND High AND VeryHigh
THEN p10;
      RULE 4: IF NoMatterWhat AND NoMatterWhat AND nHigh AND VeryHigh
THEN n10;
      RULE 5: IF NoMatterWhat AND NoMatterWhat AND VeryHigh AND
VeryHigh THEN p10;
      RULE 6: IF NoMatterWhat AND NoMatterWhat AND nVeryHigh AND
VeryHigh THEN n10;
      RULE 7: IF NoMatterWhat AND NoMatterWhat AND nHigh AND
NoMatterWhat THEN n30;
```

```
      RULE 8: IF NoMatterWhat AND NoMatterWhat AND High AND
NoMatterWhat THEN p30;
      RULE 9: IF NoMatterWhat AND NoMatterWhat AND VeryHigh AND
NoMatterWhat THEN p35;
      RULE 10: IF NoMatterWhat AND NoMatterWhat AND nVeryHigh AND
NoMatterWhat THEN n35;
END_RULEBLOCK

END_FUNCTION_BLOCK
```

APPENDIX 3

SRI Simulator World Files

This appendix contains a sample world file that comes with the ARIA library. The width and height of the environment as well as all measures are defined in millimeters.

```
width 12640
height 5759
OriginPad 12630 5757

Start Goal 11390 4334 "Chris"
End

Start Goal 1035 3002 "Webpion"
End

Start Line 1820 1239 1820 3719
AttachID 2
   1820 1239 1820 3719
End

Start Goal 9110 4394 "Matt"
End

Start Line 6410 5619 6980 5619
AttachID 4
   6410 5619 6980 5619
End

Start Line 7930 4369 7930 4919
AttachID 5
   7930 4369 7930 4919
End

Start Line 11710 1749 11710 859
AttachID 6
   11710 1749 11710 859
End

Start Goal 6650 4154 "Servers"
End

Start Line 0 3489 0 2599
AttachID 8
   0 3489 0 2599
End

Start Line 10 2599 250 2599
AttachID 9
   10 2599 250 2599
End

Start Line 10180 10 10180 360
AttachID 10
```

```
   10180 10 10180 360
End

Start Line 12070 5759 12090 5759
AttachID 11
   12070 5759 12090 5759
End

Start Line 10130 360 10170 360
AttachID 12
   10130 360 10170 360
End

Start Line 560 1869 790 1869
AttachID 13
   560 1869 790 1869
End

Start Line 7940 4369 8100 4369
AttachID 14
   7940 4369 8100 4369
End

Start Line 12090 4969 12090 5759
AttachID 15
   12090 4969 12090 5759
End

position 7760 675 0

Start Chair 8310 1862 400 520 260 200 -0.915101
   8510 2122 8510 1602
   8510 1602 8110 1602
   8110 1602 8110 2122
   8110 2122 8510 2122
End
```

BIBLIOGRAPHY

Beom, H. R. and H. S. Cho (1995). *A Sensor-Based Navigation for a Mobile Robot Using Fuzzy Logic and Reinforcement Learning.* IEEE Transactions on Systems, Man and Cybernetics, **25**(3): 464-477.

Borenstein, J. K., Y. (1990). *Real-Time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments.* International Conference on Robotics and Automation, 1990. Proceedings, 1990 IEEE.

Caracciolo, L., A. de Luca, et al. (1999). *Trajectory Tracking Control of a Four-wheel Differentially Driven Mobile Robot.* 1999 IEEE International Conference on Robotics and Automation. Proceedings, 1999.

Davidson, M. and V. Bahl (2001). *The Scalar /spl epsiv/-Controller: A Spatial Path Tracking Approach for ODV, Ackerman, and Differentially-steered Autonomous Wheeled Mobile Robots.* IEEE International Conference on Robotics and Automation, 2001. Proceedings 2001 ICRA.

DeSantis, R. M., R. Hurteau, et al. (2002). *Experimental Stabilization of Tractor and Tractor-trailer Like Vehicles.* Proceedings of the 2002 IEEE International Symposium on intelligent Control, 2002.

Driankov, D. and A. Saffiotti (2001). *Fuzzy Logic Techniques for Autonomous Vehicle Navigation.*

Fagg, A. H., D. Lotspeich, et al. (1994). *A Reinforcement-Learning Approach to Reactive Control Policy Design for Autonomous Robots.* IEEE International Conference on Robotics and Automation, 1994. Proceedings., 1994.

Farinwata, S. S. F., Dimitar P., Langari R. ed (2000). *Fuzzy Control: Synthesis and Analysis.* New York.

Free Fuzzy Logic Library (2002). Site of SourceForge, [Online].
http://ffll.sourceforge.net/ (Page consulted on June 2, 2005)

Gifford, K. K. and R. R. Murphy (1996). *Incorporating Terrain Uncertainties in Autonomous Vehicle Path Planning.* International Conference on Intelligent Robots and Systems '96, IROS 96, Proceedings of the 1996 IEEE/RSJ.

Hillier, F. S. L., Gerald J. (1990). *Introduction to Stochastic Models in Operations Research.* New York, N.Y. : McGraw-Hill , c1990.

Hart, P., Nilsson, J. (1971). *A Formal Basis for the Heuristic Determination of Minimum Cost Paths.* (Unedited text).

Howard A., Seraji H., *An Intelligent Terrain-Based Navigation System for Planetary Rovers*, IEEE Robotics and Automation Magazine, December 2001.

International Technical Commission (IEC), T. C. n. (1997). *Devices, IEC1131 - Programmable Controllers, Part 7 - Fuzzy Control Programming.* [Online] http://www.fuzzytech.com/binaries/ieccd1.pdf (Consulted on 20 June 2005).

Ju, K.-S. H. M.-Y. (2002). *A Propagating Interface Model Strategy for Global Trajectory Planning Among Moving Obstacles.* IEEE Transactions on Industrial Electronics, (6): 1313-1322.

Kanayama, Y., Y. Kimura, et al. (1990). *A Stable Tracking Control Method for an Autonomous Mobile Robot.* IEEE International Conference on Robotics and Automation.

Khatib, O. (1985). *Real-Time Obstacle Avoidance for Manipulators and Mobile Robots.* IEEE International Conference on Robotics and Automation. Proceedings 1985.

Koh, K. C. and H. S. Cho (1995). *Wheel Servo Control Based on Feedforward Compensation for an Autonomous Mobile Robot.* 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings.

Laumond, J. P. (2001). *La Robotique Mobile.* Hermès Science Publications , c2001.

Lee, M. G. P. J. H. J. M. C. (2001). *Obstacle Avoidance for Mobile Robots Using Artificial Potential Field Approach With Simulated Annealing.* IEEE International Symposium on Industrial Electronics, 2001. Proceedings. ISIE 2001.

Macek, K., I. Petrovic, et al. (2002). *A Reinforcement Learning Approach to Obstacle Avoidance of Mobile Robots.* 7th International Workshop on Advanced Motion Control, 2002.

Mucientes, M., R. Iglesias, et al. (2001). *Fuzzy Temporal Rules for Mobile Robot Guidance in Dynamic Environments.* Transactions on Systems, Man and Cybernetics, Part C, IEEE **31**(3): 391-398.

Ollero, A. and G. Heredia (1995). *Stability Analysis of Mobile Robot Path Tracking.* 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems 95. 'Human Robot Interaction and Cooperative Robots', Proceedings.

Pruski, A. (1996). *Robotique Mobile.* Paris : Hermès , c1996.

Robotics, A. (2001). *Pioneer 2 / People Bot Operations Manual.* ActivMedia.

Robotics, A. (2003). *ARIA Reference Manual 1.3.2.* ActivMedia.

Simmons, R. (1996). *The Curvature-Velocity Method for Local Obstacle Avoidance.* Proceedings. IEEE International Conference on Robotics and Automation, 1996.

Ulrich, I. B., J. (1998). *VFH+: Reliable Obstacle Avoidance for Fast Mobile Robots.* Proceedings. IEEE International Conference on.Robotics and Automation, 1998.

Ulrich, I. B., J. (2000). *VFH: Local Obstacle Avoidance With Look-Ahead Verification.* Proceedings.IEEE International Conference on Robotics and Automation, 2000. ICRA

Weiguo, W., C. Huitang, et al. (1999). *Backstepping Design for Path Tracking of Mobile Robots.* 1999 IEEE/RSJ International Conference on Intelligent Robots and Systems. Proceedings, 1999. IROS '99.

Xin, L., P. Vadakkepat, et al. (2002). *Comparison of Khepera Robot Navigation by Evolutionary Neural Networks and Pain-Based Algorithm.* Proceedings of the 2002 Congress on Evolutionary Computation, 2002. CEC '02.

Xu, H. and S. X. Yang (2001). *Tracking Control of a Mobile Robot with Kinematic and Dynamic Constraints. Proceedings 2001 IEEE International Symposium on Computational Intelligence in Robotics and Automation, 2001.*

Yang, S. X., H. Li, et al. (2003). *Fuzzy Control of a Behavior-Based Mobile Robot.* The 12th IEEE International Conference on Fuzzy Systems, 2003. FUZZ '03.

Yang, X., K. He, et al. (1998). *An Intelligent Predictive Control Approach to Path Tracking Problem of Autonomous Mobile Robot.* IEEE International Conference on Systems, Man, and Cybernetics, 1998.

Yoshizawa, K., H. Hashimoto, et al. (1996). *Path Tracking Control of Mobile Robots Using a Quadratic Curve.* Intelligent Vehicles Symposium, 1996., Proceedings of the 1996 IEEE.

Zhang, J. R., S. J. Xu, et al. (2001). *Sliding Mode Controller for Automatic Steering of Vehicles.* The 27th Annual Conference of the IEEE Industrial Electronics Society, 2001. IECON '01.

Zhang, J. R., S. J. Xu, et al. (2002). *Path Tracking Control of Vehicles Based on Lyapunov Approach.* Proceedings of the 2002 American Control Conference, 2002.