1993-09

Automated cartography by an autonomous mobile robot using ultrasonic range finders

MacPherson, David Leonard, Jr.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

DISSERTATION

AUTOMATED CARTOGRAPHY BY AN
AUTONOMOUS MOBILE ROBOT
USING ULTRASONIC RANGE FINDERS

by

David Leonard MacPherson, Jr.

September 1993

Dissertation Supervisor: Yutaka Kanayama

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
6c. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, California 93943-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification) AUTOMATED CARTOGRAPHY BY AN AUTONOMOUS MOBILE ROBOT USING ULTRASONIC RANGE FINDERS (U)

12. PERSONAL AUTHOR(S)
MacPherson, David Leonard, Jr.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Ph.D. Dissertation	From 9/91 To 9/93	September 1993	386

16. SUPPLEMENTARY NOTATION
The views expressed in this dissertation are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	autonomous vehicles, autonomous mobile robots, software architectures

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The problem solved was for an autonomous mobile robot to generate a precise map of its orthogonal, indoor environment. The maps generated by the robot's sensors must be perfect so they can be used in subsequent navigation tasks using the same sensors.

Our approach performed map-making incrementally with a partial world data structure describing incomplete polygons. A striking feature of the partial world data structure was they consist of "real" and "inferred" edges. Basically, in each learning step, the robot's sensors scan an unexplored region to obtain new "real" and "inferred" edges by eliminating at least one "inferred" edge. The process continues until no "inferred" edges remain in the partial world. In order to make this algorithm possible, linear fitting of sensor input, smooth vehicle motion control, dead reckoning error correction, and a mapping algorithm were developed. This algorithm was implemented on the autonomous mobile robot *Yamabico-11*.

The results of this experiment using *Yamabico-11* were threefold. (1) A smooth path tracking algorithm resulted in motion error of less than 2% in all experiments. (2) Dead reckoning error correction experiments revealed small, consistent vehicle odometry errors. The maximum observed error was 1.93 centimeters and 1.04° over a 9.14 meter course. (3) Precise mapping was demonstrated with a map accuracy in the worst case of 25 centimeters and 2° of hand measured maps. The ability to explore an indoor world space while correcting dead reckoning error is a significant improvement over previous work [Leonard 91] [Crowley 86] [Cox 91].

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL Yutaka Kanayama	22b. TELEPHONE (Include Area Code) (408) 656-2095	22c. OFFICE SYMBOL Code CS/Ka

DD FORM 1473, JUN 86 *Previous editions are obsolete*

Approved for public release; distribution is unlimited

AUTOMATED CARTOGRAPHY BY AN AUTONOMOUS MOBILE ROBOT USING ULTRASONIC RANGE FINDERS

David Leonard MacPherson, Jr.
Lieutenant Commander, United States Navy
B.S., Rensselaer Polytechnic Institute, 1981
M.S., Naval Postgraduate School, 1986

Submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1993

Author: [redacted]
David Leonard MacPherson, Jr.

Approved By:

[redacted]
Yutaka Kanayama
Professor of Computer Science
Dissertation Supervisor

[redacted]
Michael J. Zyda
Professor of Computer Science

[redacted]
Craig W. Rasmussen
Associate Professor of Mathematics

[redacted]
Timothy J. Shimeall
Assistant Professor of Computer Science

[redacted]
Anthony J. Healey
Professor of Mechanical Engineering

[redacted]
Approved by:
Professor Ted Lewis, Chairman, Department of Computer Science

[redacted]
Approved by:
Richard S. Elster, Dean of Instruction

ABSTRACT

The problem solved was for an autonomous mobile robot to generate a precise map of its orthogonal, indoor environment. The maps generated by the robot's sensors must be perfect so they can be used in subsequent navigation tasks using the same sensors.

Our approach performed map-making incrementally with a partial world data structure describing incomplete polygons. A striking feature of the partial world data structure was they consist of "real" and "inferred" edges. Basically, in each learning step, the robot's sensors scan an unexplored region to obtain new "real" and "inferred" edges by eliminating at least one "inferred" edge. The process continues until no "inferred" edges remain in the partial world. In order to make this algorithm possible, linear fitting of sensor input, smooth vehicle motion control, dead reckoning error correction, and a mapping algorithm were developed. This algorithm was implemented on the autonomous mobile robot *Yamabico-11*.

The results of this experiment using *Yamabico-11* were threefold. (1) A smooth path tracking algorithm resulted in motion error of less than 2% in all experiments. (2) Dead reckoning error correction experiments revealed small, consistent vehicle odometry errors. The maximum observed error was 1.93 centimeters and 1.04° over a 9.14 meter course. (3) Precise mapping was demonstrated with a map accuracy in the worst case of 25 centimeters and 2° of hand measured maps. The ability to explore an indoor world space while correcting dead reckoning error is a significant improvement over previous work [Leonard 91] [Crowley 86] [Cox 91].

TABLE OF CONTENTS

ACKNOWLEDGMENTS

First and foremost, I must acknowledge the unfailing love, dedication, and unconditional support I have received through it all from my wife and best friend, Sally, without which this work could never have been completed. Her positive attitude and understanding while handling this incredibly stressful period were remarkable. She proofread every single page of this manuscript at least a dozen times. Her valuable comments helped me to better appreciate her wisdom. This experience has brought us closer together.

I also wish to express my deepest gratitude to Professor Yutaka Kanayama whose support, guidance, and enthusiasm have been a constant inspiration to me. His door was always open for me and for any other student needing help. His patience and positive attitude were invaluable to this research.

I am deeply indebted to my committee for their patience and wisdom. Professor Michael Zyda provided valuable comments on the overall layout of this dissertation. Professor Tony Healey made insightful suggestions regarding the sonar experiments necessary to improve my understanding of Yamabico's sonar system. Professor Tim Shimeall's comments regarding my writing and scientific approach were exceptionally helpful. Professor Craig Rasmussen generously provided me with badly needed mathematical guidance as well as proofreading the manuscript numerous times. I would also like to thank Professor Robert McGhee for his support and encourage in the early stages of my program.

I appreciate the continuous encouragement and support of the members of the Yamabico research group. Their enthusiasm for my work helped me to apply constant, steady pressure during my research phase.

I. INTRODUCTION

Recent advances in computer processing speed have encouraged the development of increasingly capable mobile robot platforms. The popular trend in current military applications is to accomplish the required mission with a minimum loss of life. Consequently, many government-sponsored efforts are underway for building systems for fighting fires, handling ammunition, transporting material, conducting underwater search and inspection operations, and other dangerous tasks now performed by humans [Everett 92]. One useful naval application would be a robot for inspecting tanks, voids and other dangerous spaces on board a military ship. This kind of robotic vehicle must first be physically robust to cope with the harsh ship-board environment. Additionally, this robot must have the proper sensors, mobility, and intelligence to perform a variety of tasks. This interdisciplinary set of problems is part of the robotics field. The capability to explore an unknown environment and record data about this environment is a critical capability of this robotic vehicle. This capability is part of a larger problem called cognition. Cognition is composed of "all processes by which sensory input is transformed, reduced, elaborated, stored, recovered, and used" [Nachtigall 86]. Automated cartography is one step towards robot cognition.

In civilian robotics applications, robots must be able to adapt to changing circumstances in their environment. Ideally, a useful robot should be sufficiently adaptable to function in a totally unfamiliar environment. A highly desirable robotic domestic application is a robot vacuum cleaner that plugs itself into a wall socket and cleans the floors of a residence. This is an annoying and time consuming task for humans, and could instead be accomplished by a robot. A robotic vacuum cleaner would be capable of entering a totally unfamiliar indoor environment to clean an entire floor space with no prior knowledge of the floor plan. Additionally, the robot must be capable of adapting to unexpected changes in the house. For instance, the resident may want to rearrange the furniture for a party or add a new piece of furniture. The vacuum robot must be adaptable to changes in its environment

and clean the house with no human intervention. In order to be marketed successfully, this device needs to be able to distinguish between a diamond earring and a crumb of food on all floor surfaces in the household. Why is it that such a product does not yet exist on the market? Obviously, the cost of such a device is prohibitively high. Additionally, the robotic technology required to perform the navigation and cartography tasks does not yet exist. Indoor robot navigation and cartography are the central issues of this dissertation.

A robotic vehicle's capability to explore and map an unknown work space is called automated cartography. There are two other tasks related to automated cartography: navigation and map representation. The automated cartography problem has a circular interdependence among the robot tasks,

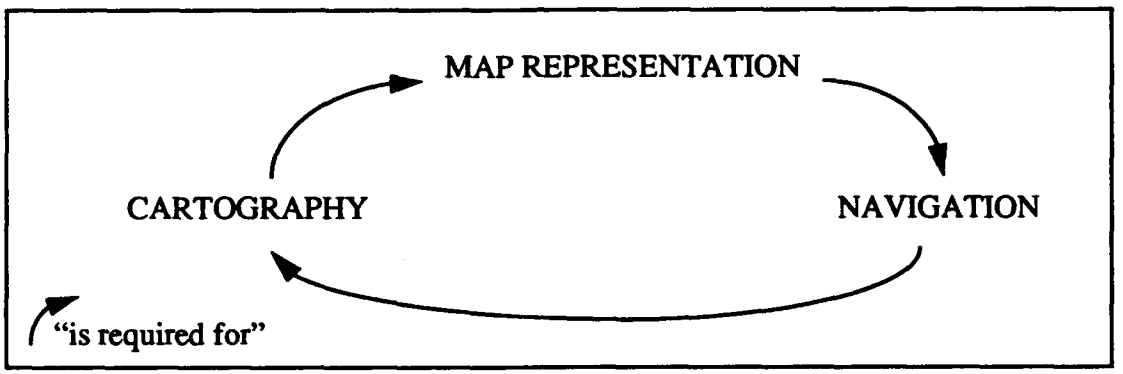


Figure 1.1 - Cartography-Map-Navigation Dependency

since navigation is required to perform cartography, cartography is the method of building maps, and the maps produced by cartography are used for navigation. Figure 1.1 illustrates this concept; each of the three arrows means "is required for" in this diagram. Obviously, some sort of bootstrapping process is required when starting without a map. In the research conducted for this dissertation, an autonomous mobile robot performs automated cartography starting with no map.

The robot environment is called the world space [Hwang 92]. The robot's internal view of the world space consists of some map representation. Guidance is defined as the process of controlling the motion of a robotic vehicle. Navigation is the process of directing the safe movement of a vehicle from one point to another [Dutton 78]. A robot navigation system provides commands to the guidance system for vehicle control. The guidance system uses a set of guidance rules to issue the proper

vehicle control instructions. The three basic problems of navigation are: 1) how to determine position (x, y) 2) how to determine direction (θ), and 3) how to determine distance traveled (s), [Bowditch 84] [Dutton 78]. The dead reckoning processing is a common means of keeping track of these four parameters. Dead reckoning refers to the projection of a present position or anticipated future position from a previous position using known directions and distances [Dutton 78]. The term localization refers to the process of determining a robot's position using information from external sensors [Leonard 91]. Dead reckoning errors are normally corrected based upon localization information. Odometry is defined as the process of integrating the robot's wheel motion in order to maintain an estimate of a robot's current configuration.

A configuration is a four element data structure used to describe a robot position, the error in a robot's position, the position of some object and some types of path elements. The four elements are x, y, θ, and κ. When used to describe a robot's position, (x,y) is the robot's location in the Cartesian plane, θ is the robot's orientation with respect to the global x axis, and κ is the robot's instantaneous path curvature. Odometry estimate error is defined as the algebraic difference between a vehicle's estimated configuration and its actual configuration. Odometry is a purely internal means of estimating a vehicle's position. Normally, wheel rotations are integrated by reading some type of wheel encoder. Odometry error tends to increase linearly as a function of the total distance traveled and is primarily associated with slip between the robot wheels and the ground. Automated odometry error correction is defined as the process whereby an autonomous vehicle reduces its odometry estimate error by determining its position with respect to some external feature in the world space. Features suitable for odometry correction by a given sensor are called landmarks. In this dissertation, only naturally occurring landmarks are used for navigation. This means that no artificially placed landmarks are installed in the robot's world space to facilitate navigation.

Map representation is the other important part of automated cartography. A map is a symbolic representation of some finite space; a detailed map of a small portion of the world

may contain a large volume of information. In order to ensure that the cartography problem remains tractable, only the salient features of the world that apply to the map's intended application should be represented on the map. The maps considered in this dissertation are designed specifically for autonomous mobile robot navigation.

A. SCOPE OF DISSERTATION

This dissertation presents a novel software system for the automated cartography of an unknown world space by an autonomous mobile robot. The automated cartography algorithm developed in this dissertation is an efficient means for an autonomous mobile robot to effectively explore an unknown world space while building a spatially consistent map of the space. The FSM problem domain has three major aspects: automated workspace navigation and exploration, path tracking as the method of vehicle control, and automated landmark recognition with odometry error correction. Previously, no approach has successfully integrated the solutions of these three component problems into a single software system.

How is the automated cartography algorithm different and better than the rest? As already explained, the map representation, the vehicle navigation and the cartography requirements are intimately related. This algorithm allows the robot to use a partially built map to decide which areas of the world space require exploration. The algorithm uses office-building heuristics to take advantage of features found in most office buildings. An example of this heuristic is that an indoor hallway is assumed to have straight walls broken by doorways. Basically, a scan model derived from a single robot motion is merged with the robot's partial world model (*PW*) and the state of the new *PW* is used to guide the search for unexplored portions of the world space. The path tracking vehicle control sub-system provides smooth vehicle control and the necessary vehicle motion for odometry error determination and correction while the vehicle is moving.

B. THE AUTOMATED CARTOGRAPHY PROBLEM

1. Problem Statement

The automated cartography problem involves the incremental modeling of an unknown, indoor world space. Automated cartography involves a robot system R mapping a planar, static,

people-free, indoor world space W. All objects in the environment are represented as rigid, convex polygons in the robot's world space. All areas mapped lie in the same Cartesian plane. This enables R to map, for example, a single floor of an office building. R builds a large scale, spatially-consistent, metrically-accurate map PW after being placed in any arbitrary configuration C_o in the world space W. The robot sensors S are 12 fixed ultrasonic range finders. R starts with no prior knowledge of W and it must navigate in a manner required to explore the entire accessible portions of W using its partially built map PW and sensors S. Spatial consistency of PW is maintained by the robot's capability to update its dead reckoning configuration C using naturally occurring landmarks L_i in the work space.

2. Example of a Typical Automated Cartography Experiment

The enclosed-space experiment illustrates the problem best: an uncluttered, enclosed, indoor space in a typical office building is selected as the robot's world space. The automated cartography software is loaded onto the robot and the robot is placed in any arbitrary starting configuration with no map of the workspace loaded into its memory. The program is started and the robot is left in the enclosed space. The robot spends about one hour exploring the entire enclosed space, building and refining a precise map of all accessible regions in this enclosed space. Upon map completion, the robot returns to its starting position ready to execute commanded trajectories using its learned map for accurate position determination at arbitrary locations in the environment [Leonard 92].

3. Assumptions

The 2D assumption - The world space is a flat planar world with obstacles. The floor is the x, y plane. All obstacles faces are perpendicular to the x, y plane and have a constant size along the positive z-axis. This assumption is required to assure a good sensor return from all objects.

Orthogonal Wall Assumption - Walls in the robot's world space are always rectilinear, as are found in most office buildings.

Rigid Body Assumption - The vehicle and all objects in the robot's world space are rigid. The surface of any object in the world space may be represented by a single configuration.

Static World Assumption - All objects in the world space are immobile both in a relative and an absolute sense.

C. ORGANIZATION OF DISSERTATION

Chapter II reviews the major challenges and issues currently faced in the field of mobile robot cartography. The issues discussed represent the major hurdles that must be overcome for automated cartography. This chapter also describes research work in the field of mobile robot navigation. The significant contributions in the field are reviewed.

Chapter III gives a detailed description of the Naval Postgraduate School autonomous mobile robot Yamabico-11 vehicle and the simulator used to develop the robot software used in this dissertation. The limitations of ultrasonic sonar sensors for robotic applications are described. This simulator description provides an introduction to the Model-based Mobile robot Language (MML) programming environment. The simulator proves to be an important software development tool since some debugging is tedious or even impossible on the robot. Efforts to improve software efficiency, organization, and functionality are tried in simulation first before testing begins on the robot.

Chapter IV presents the software architecture for the vehicle control for Yamabico-11, the robot test-bed for this dissertation. The software scheduling system, a geometry module, the sonar subsystem, and the input/output subsystem are described in detail.

Chapter V presents the theory of robot odometry correction. An algebraic approach is taken to describe a robot configuration, the error in the robot's configuration as well as the configuration of landmarks in the robot's world space. A detailed explanation of the method of real-time, on-line odometry correction is provided.

Chapter VI provides the basic data structures for world representation. Chapter VII describes a theory for robot automated cartography using an idealized sensor. This provides the theoretical

foundations for robot cartography using real sensors. The limitations of the real sensors impose the modifications required to the ideal algorithm presented in Chapter VII.

Chapter VIII describes the theory of real automated robot cartography using the automated cartography algorithm. Each aspect of this algorithm is explained using examples. The means for on board map representation, robot exploration and map refinement are described.

Chapter IX gives the details of the experiments performed and the results obtained. Experimental results are plotted to help the user better review the results. Chapter X of the dissertation finishes with a summary of the conclusions drawn from the theory and the experimental results and recommendations for further research.

Appendix A is a comprehensive robot user's manual. Appendix B provides MML locomotion source code. Appendix C gives the sonar functions used for feature extraction. Appendix D lists the odometry correction source code and appendix E gives the real robot cartography source code.

II. AUTOMATED CARTOGRAPHY: MAJOR CHALLENGES AND ISSUES

This chapter is designed to provide the reader with background on the major challenges and issues in robot cartography. Automated cartography represents a significant research undertaking in the development of an intelligent autonomous robot capable of exploring its environment. This chapter addresses the major challenges and issues that were addressed in the development of the Yamabico-11 automated cartography system. They are presented briefly in this chapter to lay a firm foundation for understanding and to provide an overview of the design decisions that went into Yamabico's software. This chapter cites significant robotics projects from the literature to illustrate each issue.

Map representation is a critical issue because the world is rich with features and a robot's memory is typically of limited size. Therefore, only the important features should be stored since the map must contain the necessary information for robot navigation but cannot be too large. Robot motion planning is an important challenge since a robot must plan a purposeful route through the world space in order to map the space. Software architecture of the entire control system is important since this factor determines the software's efficiency and modifiability to a large extent. Some software architectures lend themselves to cartography more readily than others. Robot sensing is a critical issue with regard to cartography because a robot must sense its world in order to build a map of it. Robot localization and navigation are important challenges because the robot must navigate effectively to explore its environment. Dead reckoning errors must be corrected by means of localization in order for a robot to build a spatially consistent map. Robot exploration is necessary in order for the robot to move its sensors to all reachable portions of the world space. These issues are not limited to Yamabico; they span the fields of robotics, navigation, computer science and mathematics.

A. METHODS FOR MAP REPRESENTATIONS

A robot must have a model of objects in its environment before it can plan a collision-free path through its world space. The robot may have an *a priori* map of its environment or it may use sensors to acquire knowledge about its surroundings. Sensors are typically used to build a depth map of the surrounding environment. A depth map is a statical representation of many range finder returns. This information is normally converted into a some compact representation to save memory space and to speed up computations. Once information about shapes and configurations of objects is acquired, it can be represented in several ways. The trade-offs between simplicity, resolution, and computational efficiency must be carefully considered when choosing the best means of representation for a specific application. The remainder of this section reviews the commonly used map representation techniques.

1. Grid Representation

The grid representation method divides the robot's environment into an array of identical cells. These cells are typically rectilinear. The robot's environment is represented by marking the individual cells as either one if it is occupied by an object or as a zero for unoccupied. The simplicity of this method has many computational advantages, especially on a massively parallel computer. The cell size governs the overall resolution in the robot's environment; smaller cells give higher resolution but incur a penalty in terms of on board storage requirements and computational efficiency. Elfes called this factor the resolution axis [Elfes 87]. Moravec used certainty grids for mobile robot map representation [Moravec 87]. Borenstein and Koren used a grid-type representation called a vector field histogram on the robot Carmel at the University of Michigan [Borenstein 92]. Beckerman and Oblow [Beckerman 90], Everett [Everett 89], Noborio et. al. [Noborio 90] and Zelinsky [Zelinsky 88] have also used various grid-based representations to build maps from sonar data.

Elfes implemented an autonomous mobile robot navigation system called Dolphin on the Neptune (indoor) and Terregator (outdoor) mobile robots. This system used sonar range data to build a multilevel description of the robot's surroundings using a grid-based map representation called occupancy grids [Elfes 87]. Elfes used a multilevel description of the robot's operating environment. Several dimensions of the representation were defined: the abstraction axis, the geographical axis, and the resolution axis. The system was completely autonomous in that it had no *a priori* model or knowledge of its surroundings [Elfes 87]. Range measurements from multiple points of view were combined into a sonar map while accounting for uncertainties and errors in the data. By combining the evidence from many readings as the robot moved in its environment, the area known to be empty was expanded [Elfes 87]. Elfes used 24 ultrasonic range finding transducers arranged in a circular array to build dense two dimensional maps based upon empty and occupied volumes in a cone in front of the sensor.

Elfes's research involved grid-based mapping, whereas this dissertation focuses on feature-based mapping. The reason the feature-based approach was chosen is that the computational complexity of this approach is lower than the $O(n^4)$ complexity of Elfes's work where n is the number of grid squares in the map. In the Dolphin system, all map computation was done off-line on a VAX-11/780. Yamabico's mapping system does all mapping computation using the on board processor. On board processing eliminates communication delays between the processor and the robot, and allows Yamabico full autonomy. The Dolphin system used Polaroid laboratory grade ultrasonic range transducers with a 30 degree beam width. The 3 dB beam width was approximately 15 degrees. Yamabico uses a collimated beam sonar sensor with separate emitter and receiver.

Grid-based approaches to mapping make weaker assumptions about the environment than the polyhedral approach since grid type representations do not explicitly represent surface boundaries in the robot's world space [Leonard 91]. Thus arbitrary inaccuracies and uncertainties are always present in grid-based approaches.

2. Cell Tree Representation

The cell tree representation is also called quadtree for two dimensional (2D) representations or octree for three dimensional (3D) representations. This method was developed to improve the overall efficiency of the grid method when representing a large object or a large open space. The cell tree representation divides the robot's world space into a small number of large cells. The cells are not necessarily all the same size. Cells completely inside or outside of the objects are marked either occupied or empty. Cells partially occupied are further divided. This process is repeated until the cell size reaches an arbitrary resolution limit. This method represents a significant reduction in storage space requirements at the expense of additional complexity. Several researchers have used cell trees for map representation [Fryxell 88] [Airey 90]. The cell tree approach is a stronger approach to mapping than the grid-based approach, but still does not explicitly represent object surface boundaries as does the polyhedral approach. Cell tree representation straddles the representation spectrum midway between grid-based and polyhedral representations since the division of space is defined to some extent by the objects being represented.

3. Polyhedral Representation

A polyhedron is a solid figure having many faces. Objects in the robot's environment may be approximated by the unions of polyhedra. This is an efficient means for representing a robot's world space since much less storage space is used than for the grid-based method. Only the boundaries between open space and objects are represented, instead of every grid-square in the world space. Curved surfaces must be approximated as planar surfaces to maintain the polyhedral representation. Efficient ($O(nlogn)$) algorithms exist for computing the intersection of and the distance between two polyhedra [Hwang 92]. A robot's world space was first represented by polygonal objects by Lozano-Perez in his influential Configuration Space (C-Space) [Lozano-Perez 79].

4. Constructive Solid Geometry Representation

The Constructive Solid Geometry (CGS) method of free space representation is used in solid modelers. The CGS represents objects as unions, intersections, and set differences of primitive shapes including spheres. This method has the advantage that curved surfaces can be represented with a small number of parameters specifying the curve. This method is often used in conjunction with computer aided design (CAD) systems for environment mapping [Hwang 92].

5. Topological Map Representation

The topological map representation approach uses a graph-theory approach to represent robot free space. A graph $G = (V,E)$ is a finite non empty set V of elements called vertices, together with a set E of two-element subsets of V called edges [Gould 88]. Vertices represent places the robot may visit and edges represent pathways used to travel between nodes. The robot Huey at Brown University also used a topological graph to represent the map in the 1992 AAAI robot contest [Davis 93].

Mataric used a topological map representation with the robot Toto [Mataric 92]. Toto navigated using ultrasonic range finders and a flux-gate compass. The experiment included automated building of simple topological maps of the robot's world space. Landmarks were represented as a tuple $<T, C, L, P>$ where T was the landmark type, C was the average compass bearing, L was the landmark's length and $P = (x, y)$ was a course position estimate. Whenever a landmark was detected, it was matched to all known landmarks that were stored in a graph structure. Either a unique match or no match occurred. Localization was a simple process of comparing the stored landmark descriptor $<t, c, l, p>$ with the robot current sensory information $<t', c', l', p'>$. The map structure consisted of a graph with each node representing a robot-detected landmark. Edges defined the connections between the landmarks.

The work described in this dissertation is different in several ways. No flux gate compass is used for determination of Yamabico's orientation. In both cases, landmarks are

automatically recognized and used for odometry correction. In Yamabico's case, the position as well as the orientation of individual landmarks is used to correct the robot's orientation. This was not true for Mataric's work. Mataric's algorithm recognized landmarks based upon their length and position whereas the automated cartography algorithm recognizes landmarks by their precise position and orientation. Finally, Mataric's algorithm was essentially a modified wall follower behavior; it did not adapt well to open spaces. The automated cartography on the other hand greedily acquires and maps the open space available and is not restricted to modified wall following.

Topological representations provide a compact method of storing a map with many features. However, topological maps do not explicitly record the metric distance between vertices. This renders topological maps less useful for robot navigation than maps with distances explicitly represented.

B. APPROACHES TO ROBOT MOTION PLANNING

Robot motion planning is the process whereby a robot's path is planned based upon the robot's current configuration and the representation of the robot's environment. Planners use a world model as an input to plan a safe, efficient path from one configuration to another. Not all robotic systems plan the robot's motion in a deliberative fashion. In fact, there exists a broad spectrum of motion planners, from no plan/no model to a flexible plan to a rigid, unalterable plan. Many different methods have been developed for robot motion planning. Some methods are widely applicable, whereas others solve only a narrow range of motion planning problems.

The motion planning problem is defined as follows. Let R be a robot system having k degrees of freedom, and suppose that R is free to move in a two or three dimensional space V amidst a collection of non-moving obstacles whose geometry is known to the robot system. The motion planning problem for R is: given an initial position Z_1 and a desired final position Z_2, determine whether there exist a continuous obstacle-avoiding motion of R from Z_1 to Z_2 and if so plan such a motion [Schwartz 88]. The general motion planning problem

can be solved in polynomial time in the number n of algebraic constraints defining FP where FP denotes the space of free positions [Schwartz 83].

Motion planning methods fall into four general categories: skeleton, cell decomposition, potential field, and mathematical programming [Hwang 92]. Most motion planning problems can be approached by one of these four methods. Hybrid combinations of these approaches are often used in developing new motion planners.

1. Skeleton

In the skeleton approach to motion planning, the set of all feasible motions is mapped onto a network of one dimensional lines [Hwang 92]. These lines represent safe pathways for robot motion in the free space. This approach has also been called the retraction, roadmap, or highway approach. The advantage of this method is that the search for a solution is limited to the skeleton. Using this approach, motion planning is accomplished by first moving the robot from its starting position to a point on the skeleton. Next, the robot is moved from the goal configuration to a point on the skeleton. Finally, the two points on the skeleton are connected using lines in the skeleton. Two well-known skeletons are the visibility graph and the Voronoi diagram [Canny 88]. One advantage of this method is that skeletons for a large area can be preprocessed using a known world model as input. Brooks represented free space as a union of possibly overlapping generalized cones. A generalized cone has an axis of a certain length and a boundary on each side of the axis. He used generalized cones to represent free space in a 2D world and the robot traveled on spines of the generalized cones [Brooks 83]. An improved quality path was obtained by representing free space as a union of generalized cones and convex polygons [Kuan 85].

2. Cell Decomposition

In the cell decomposition approach, the free configuration space is first decomposed into a set of simple cells and then adjacency relationships between the cells are computed. To find a collision free path, the cells containing the start and goal configurations are connected with a sequence of empty, adjacent cells. In this method cell boundaries can be

object dependent or independent. With an object dependent decomposition, boundaries of obstacles are used to generate cell boundaries. The free space is the union of the free cells. With this method the number of cells is small but the complexity of the decomposition is high.

With object-independent decomposition, the configuration space is partitioned into cells of a simple shape, then each cell is tested for occupancy. Since the cell shape and location are independent of the object shape and location, the cell boundaries do not tightly enclose the object [Hwang 92]. Increasing the number of cells can make the representation error arbitrarily small. Examples of object-independent cell decompositions are grid and quadtree.

3. Potential Field

The potential field approach treats the robot as a particle under the influence of an artificial potential field whose local variations are expected to reflect the "structure" of the free space [Latombe 91].This approach has been compared to a sticky marble rolling downhill on the interior surface of a bowl [Arkin 89]. The goal point for the robot is the lowest point in the bowl and obstacles are represented by inward dents in the bowl. This approach constructs a scalar function called the potential that has a minimum when the robot is at the goal configuration, and a high value on obstacles in the configuration space. At all other locations in the configuration space the function is sloping downward toward the goal configuration. The robot moves toward the goal by following the negative gradient of the potential to the minimum.

To use this approach, an obstacle potential is constructed. This field has a high value on the obstacles and decreases monotonically as the distance from the obstacles increases. Superimposed onto the obstacle potential is a goal potential that has a large negative value at the goal and increases monotonically as the distance from the goal increases.

This approach has the advantage of being simple but there are usually several local minima other than the goal. These minima can trap the robot. Another disadvantage is

the potential field expression becomes very complex when there are many concave obstacles in the configuration space. The potential field approach is best used as a local motion planning algorithm in conjuction with some other global motion planning algorithm [Hwang 92]. Most planning methods based on the potential field approach have empirical connections. They usually do not guarantee that a path will be found even when one exists. They are, however, particularly fast in a wide range of situations. Potential field planners are increasing popular because an efficient and reliable motion planner can be constructed using this paradigm [Latombe 91].

4. Mathematical Programming

In this approach, motion planning is formulated as a mathematical optimization problem that finds a path between the start and the goal configuration by minimizing some scalar quantity. Mathematical programming has difficulties with non-unique solutions, singular matrices, and non-static environments.

C. SOFTWARE ARCHITECTURE

The software architecture is the structure of the control system for a robotic platform. Typically, a robotic vehicle and its associated control software have been developed in tandem [Busnel 79]. The robot's morphology and its software architecture are closely related since software and hardware are both designed to solve some particular problem [Brooks 93]. The various approaches to system architecture for autonomous vehicle control have been grouped into six categories [Arkin 89]. These classes are monolithic control, hierarchical (deliberative) control, behavior-based control, hybrid systems, distributed control, and machine learning systems.

1. Monolithic Control

Monolithic control systems are limited capability systems typically used on a factory floor. These systems tend to be sensor dependent and employ a teaching pendant approach to motion control. The robot's environment is engineered so that the robot uses ar-

tificial landmarks for localization and navigation. These systems are inflexible and not generalizable. However they are advantageous since they are easy to develop.

2. Hierarchical (Deliberative) Control

Hierarchical systems typically have a top-down control structure. The complexity of the system is managed by abstracting complex vehicle behavior into successively less complex functional levels in the same manner as structured computer programming. Typically, the high level planner is at the top of the hierarchy and the low level servo control is at the bottom. These systems normally maintain a symbolic world model to support sensory processing. The world model contains the robot's current state and the current state of the robot's environment.

Commands are passed from the top level symbolic planner down the hierarchy. Sensory information is passed up the hierarchy. The update rate of a given level tends to increase as one moves down the hierarchy. The planning horizon for each level tends to grow longer as one moves up the hierarchy. The state space reasoning tends to occur at a high level and tends to be purely symbolic in nature. These systems are characterized by a slow response time to sensor input. This can be a significant disadvantage in a rapidly changing environment. The symbolic reasoning gives the advantage of a high-level, global intelligence due to deliberative reasoning. These systems are also characterized by a variable latency due to the running time of different deliberative portions of the system. The Hughes control system for the Autonomous Land Vehicle (ALV) is a classic example of a hierarchical control system [Daily 88].

3. Behavior-Based (Reactive) Control

Autonomous vehicle control using the behavior-based control architecture tends to focus on reaction to input stimuli rather than deliberative planning. These systems are typically built in layers of successively more complex behaviors. The lowest level of behavioral competence for the robot is designed, built, and tested first. Progressively higher

level layers of behavioral competence are then added. These layers all run independently [Brooks 86a].

This control method employs no centralized intelligence. Instead the intelligence is distributed among the layers of competencies. Each layer processes sensory data and outputs a specific behavior. These behaviors compete through a network of surpression nodes. The vehicle's overall behavior is said to "emerge" from the interaction of multiple, competing, unintelligent layers.

Reactive control is characterized by no central world model since the world provides its own model [Brooks 91]. Therefore these systems tend to be representation-free. Since the individual layers have short sensor-effector arcs, these systems have the advantage of real-time response to input stimuli. No central intelligence system is operating, therefore these systems tend to have low-level overall intelligence when compared to deliberative systems. The layers are typically simple, consequently, these systems tend to execute only simple computations. Brooks at MIT coined the term subsumption to mean higher level robot behaviors subsume lower level behaviors when appropriate. He rejected traditional Artificial Intelligence (AI) as dogma and ridiculed precise robot navigation research. He emphasized robotics systems with an ongoing physical interaction with the environment [Brooks 90]. He believed the world provided the robot with the best model. Based on evidence from evolution, he believed robot mobility, acute vision and the ability to carry out survival related tasks in a dynamic environment provided the basis for the development of intelligence [Brooks 91]. Further, he argued that issues of representation stalled artificial intelligence research. More surprisingly, he claimed that traditional representation was unnecessary. His robotic systems architecture was decomposed into independent and parallel producers that interfaced directly with the world through perception and action. Brooks adopted a layered architecture approach and built completely autonomous mobile agents that coexisted in the world with humans and called them *Creatures* [Brooks 91]. These *Creatures* were expected to cope with changes in their world. They were robust

and adaptable to changes in the environment, maintained multiple goals, and were able to capitalize on opportunities presented by the environment.

Brooks argued that no central symbolic information processor was necessary to build the *Creatures*. Instead, the robot's software was built incrementally by the use of behavioral layers. Each of these layers added an additional behavioral competence. Brooks called the approach subsumption architecture since the layers acted independently and in parallel with other existing layers. The system had no centralized control, and no centralized repository for sensor information. Further, Brooks claimed that intelligent robot behavior emerged even though the robot stored no internal representation of the physical world. Maintaining no internal representation of the world has some significant limitations. For instance, a robot tasked with repeatedly traversing the same obstacle field would greatly benefit from an internal representation since a path could be planned around a previously encountered obstacle. Instead a robot with no memory of obstacle location is doomed to repeat the same obstacle avoidance behavior each trip. One researcher expressed his skepticism by saying "subsumption architecture is better suited to building thermostats than intelligent agents" [Wallich 91]. This means that researchers who do agree with Brook's approach do not believe that useful, intelligent behavior will ever emerge from a collection of primitive reflexes.

As one of Brook's students, Connell demonstrated a subsumption program for gathering soda cans in an office building environment using the robot Herbert. The robot built no maps of its surroundings, but managed to wander about, find and pick up a soda can and return to its starting position [Brooks 93]. This experiment required careful placement of the robot and the soda can since the robot followed only one path in response to external stimuli. Brook's philosophy represented the opposite end of the representation spectrum with respect to the Yamabico project. Yamabico's software system relies heavily on centralized control, explicit goals, and an internal representation of the world. Brook's software was composed of individual, competing behaviors to produce emergent intelligent

behavior. Brooks has rejected any sort of centralized representation on the external environment whereas Yamabico builds a detailed map of its world.

4. Hybrid Systems

Hybrid vehicle control architectures lie on the continuum between the hierarchical and behaviorist extremes [Byrnes 93]. Hybrid architectures attempt to combine the best characteristics of both hierarchical and behavorist architectures. The explicit global intelligence advantage of hierarchical systems is typically combined with the quick-reacting, reflexive behavior of the behavior-based models. Plan formulation in hybrid systems tends to borrow from hierarchical systems in order to gain deliberative intelligence. Plan execution, however, is similar to reactive control.

The Autonomous Land Vehicle (ALV) was designed and developed by Martin Marietta Aerospace as a test bed for research in autonomous mobility systems [Turk 88]. Its dimensions were 2.7 meters wide, 4.2 meters long, and 3.1 meters high, and provided the capacity to carry all power, sensors, and computer systems necessary to support autonomous operations. The ALV weighed approximately 16,000 pounds fully loaded yet was capable of traveling both on and off road. The vehicle had an eight-wheel drive, was diesel powered, and driven by hydrostatic transmission. A wide range of sensors was employed, and included a video camera, a laser range finder, and wheel-mounted odometers.

A control software architecture was developed for the ALV by Hughes Artificial Intelligence Center [Daily 88]. The hybrid architecture was organized into four levels and each level contained planning and perception functions. At the highest level, the mission planner was used to define mission goals and constraints. These were passed to the next level, which maintained the world model and developed plans based on stored maps. The resulting route plan was then passed to the third level containing the local planner. The local planning module selected and monitored reflexive behaviors at the lowest level. It was at that level that reflexive behaviors were used as real-time operating primitives [Payton 90]. Reflexive behaviors were independent of each other and executed concurrently, however,

it was the responsibility of the local planner to partition the appropriate behaviors depending on the current environment. The ALV architecture was field tested and was the first system to demonstrate obstacle avoidance in natural terrain [Olin 91].

5. Distributed Control (Blackboard Paradigm)

The term blackboard has been applied to any globally-accessible data structure to which multiple processes may communicate by posting messages. The blackboard method of vehicle control is characterized by one or more global data structures called blackboards. These blackboards constitute the working memory (or global database) for the control system. Separate knowledge sources read from and write to the blackboards. These knowledge sources reason about information obtained from the blackboard and write their conclusions back to the blackboard. These systems are typically synchronous and cooperative in operation. The Task Control Architecture deployed on the Ambler walking robot at Carnegie-Mellon University (CMU) is a distributed control system [Simmons 91] [Simmons 92].

6. Machine Learning Systems

Artificial neural nets are computer programs designed to imitate the brain's ability to learn from experience. A common type is the feed-forward neural net. Machine learning is accomplished by encoding information in the net's simulated synaptic connections. Neural nets have been used as an architectural approach for robot control [Nehmezow 92] [Thrun 92].

The CMU robot Odysseus placed fourth in the American Association for Artificial Intelligence (AAAI) robot competition in 1992 [Davis 93]. Odysseus used ultrasonic sonar data to build environmental maps off-line at CMU Artificial Intelligence (AI) Lab. While at CMU, Thrun developed a system to explore and model an office building environment efficiently [Thrun 93]. He used the robot Columbus, a modified Heathkit robot. This was an autonomous, wheeled robot with bumper sensors, a rotating sonar sensor, and a motion sensor for odometry. The exploration system used an instance-based learning technique for developing the map. Two artificial neural networks were used to encode the

characteristics of the robot's sensors and the characteristics of a typical indoor environment [Thrun 93]. One neural network encoded sensor interpretation and the other encoded confidence assessment. Exploration was achieved by navigating the robot to regions with little sonar data [Thrun 92]. The world model was represented using a grid-based mapping technique with four inch grid squares.

Neural network training allowed the network to encode the specific characteristics of the sensors as well as those of typical environments of a mobile robot; it captured knowledge independent of any particular environment the robot might face. An instance-based approximation technique was employed for modeling the environment. Exploration was guided by an anytime planner based upon dynamic programming for planning low-cost paths to poorly explored areas [Thrun 93]. The approach to model building and position control was successfully used as part of the CMU entry Odysseus in the AAAI robot competition in 1992 [Thrun 93][Davis 93].

A neural network approach to automated cartography was demonstrated by Nehmzow [Nehmezow 92]. The robot Alder used a self-organizing network to construct internal representations of the world it experienced as it moved around. The resulting neural network was a map, but a map in motor-sensory space rather than the physical space [Nehmezow 92]. In this way robot behavior and sensing were well coupled to the environment and the task of map building.

Neural networks are limited for robot cartography applications because they are typically not portable among robot platforms. Neural network programs tend to be opaque with respect to human understanding and do not scale well to larger software systems. Additionally, the long training time for a neural network system is a distinct disadvantage in robot navigation applications.

D. ROBOT SENSING

Although sensor interpretation and world modeling are fundamental for robots to operate in the real world, robotic perception is still one of the weakest components of current

robotic systems [Iyengar 91]. The advantages and disadvantages of the most commonly used types of robot sensors are presented to help the reader understand sensor interpretation issues, sensor integration and the recovery of the world model from the robot's spatial perception. The main problems in robotics perception are interpretation of noisy sensor data, computational overhead required to process sensor input, and sensor integration.

1. Laser and Light Range Finders

There are two basic laser range finder designs dependent upon the round trip time of flight to objects in the environment. The first kind measures phase shift in a continuous wave modulated laser beam that leaves the source and returns to the detector coaxially. The second measures the time a laser pulse takes to go from the source, bounce off a target point and return coaxially to a detector. Since light travels at approximately one foot per nanosecond, the supporting instrumentation must be capable of 50 picosecond time resolution for a range accuracy of plus or minus one quarter inch [Jarvis 93].

The ALV used a laser range scanner for navigation during off-road operation. The laser range scanner was an effective sensor in this type of environment. In a structured, indoor environment a smaller, more maneuverable robot is required to perform the mapping task. One important indoor limitation is doorway width. The large size of the laser range finder used on the ALV prohibited its use for indoor applications. Also, since most indoor spaces have flat floors, a 3D terrain mapping system is unnecessary.

Laser range finders provide 3D data directly by active sensing. At CMU, the Autonomous Land Vehicle and Planetary Exploration projects focused on perception of outdoor terrain for path planning and object recognition [Hubert 88]. Perception techniques for mobile robots have been validated by using real robots in real environments. 3D vision techniques have been implemented on three mobile robots developed by the Field Robotics Center at CMU: the Terregator, the NavLab, and the Ambler.

The Terregator was a six-wheeled vehicle designed for rugged terrain. The Navlab, with all computing equipment on board, was a converted van designed for navigation

on roads or on mild terrains. The Ambler was a hexapod walking robot designed for studying robotic exploration of Mars and was capable of traversing steep slopes, rocks, and wide gullies.

The active exploration of other planets by mobile robots demands that they be fully autonomous. A manned mission, even to Mars, is highly unlikely in the foreseeable future. In addition, conventional teleoperation of robots appears to be impractical due to the long time delays in signal transmission (up to 30 minutes) over the extreme distances involved. An alternative solution would involve an autonomous mobile robot capable of safely navigating extremely rugged terrain while intelligently gathering materials and telemetry readings and returning them to earth for analysis. The National Aeronautics and Space Administration Jet Propulsion Laboratory (NASA/JPL) Mars Rover uses four laser sensors for object recognition.

CMU used a time-of-flight laser range finder developed by the Environmental Research Institute of Michigan (ERIM). This was a phase difference type device. A two-mirror scanning system allowed the beam to be directed anywhere within a 30 degree by 80 degree field of view. The ERIM sensor gave 64 by 256 range images coded on eight bits from zero to 64 feet with a range resolution of three inches.

The ALV used a laser range scanner to measure the distance along the line of sight to the nearest object. A phase-shift laser scanner was used with a maximum range of 64 feet and a range resolution of 1% (3 inches). A Cartesian Elevation Map (CEM) was built to represent laser range data. The CEM was a downward-looking terrain representation that was used for autonomous navigation [Olin 91]. Multiple CEMs were fused together to build traversibility maps that were based upon artificially moving a model of the ALV over the a model of the sensed terrain. A map-based planner used digital terrain data to determine the vehicle's route which was represented as a set of subgoal point locations.

The HILARE multi-sensory system included an array of 14 ultrasonic emitter-receivers with a maximum range of two meters. A camera and laser range finder mounted on a pan and tilt platform provided the robot with 3D data about the environment. HILARE

also had an infrared triangulation system used in areas where fixed navigational beacons were installed. Robot localization was performed by reference to fixed infrared beacons when available. Sonar and laser fixed obstacle edge referencing were used when no beacons were available. HILARE explored its environment and performed automated cartography by a space structuring method that divided the known world into simple polygonal shapes. HILARE then moved in a fashion that expanded its known world and built a map using laser range finder data.

Using active sensing such as a laser range finder has the advantage of eliminating the calibration problems and computational cost inherent in passive techniques such as stereo vision [Hubert 88]. A second advantage is the lack of sensitivity to outside illumination conditions, which considerably simplifies the image analysis problem. This is particularly important for outdoor navigation since scene illumination varies widely. Other advantages of laser methods include high data rate, accuracy, and long range.

Current disadvantages include expensive equipment cost for laser sensors. Additionally, in some cases lasers are color dependent, such that shiny surfaces give either no range due to poor reflectance or false range values.

2. Infrared Range Finders

Infrared sensors are active emitters sensors that work on a send/receive format. Infrared range finders use light with a frequency just below the visible spectrum. The sensor emits an infrared light from one source and measures the amount of reflected light with one or more light detectors. Since these devices measure light attenuation, they are highly biased by the environment. Object color, object orientation, and ambient light all contribute to erroneous readings. Since the transmission signal is light instead of sound, these sensors can have a high sampling rate. Due to noise factors, infrared range measurements are only useful for short distances [Crowley 89]. Infrared sensors are frequently used to provide range data inside the minimum sonar range (typically 17 inches) for Polaroid sensors.

Infrared sensors have the advantage of being small and low cost. The Khepera™ miniature robot at the Laboratorie de Microinformatique in Switzerland is the size of a soda can but has 37 infrared range finders as its primary sensor [Modada 93]. Therefore, they are employed on many smaller robotic platforms. Infrared sensors have the disadvantage that their output is not proportional to the target range since they are adversely affected by ambient light conditions and by the color and texture of the target's surface. Infrared sensors have a relatively short maximum range, normally about 20 centimeters. This limits the sensor to certain "close-in" applications.

3. Contact Sensing

Contact sensing includes force and tactile sensing methods. Force sensing measures the resultant mechanical effects of contact, while tactile sensing involves the detection of a wide range of local parameters (physical and chemical) affected by contact [Dario 86]. Contact sensing limits a robot's ability to quickly gather data about its environment since some part of the robot must physically contact the environment. Contact sensing also has limited resolution and is limited by the robot's range of motion.

Tactile sensing has been ignored historically in favor of other types of sensing, particularly vision. Tactile sensing is important for short range recognition tasks, assembly and parts-fitting work and inspection tasks. Robotic tasks that call for close tolerances or low absolute error can benefit from tactile sensor input.

Some examples of contact sensors on mobile robots are bumpers, whiskers, and feelers. These are simple force sensors that employ some type of contact switch that shuts when contact is made. More complex tactile sensors measure feedback force and are often found on robotic arms, hands, or fingers. The Genghis™ robot designed by Brooks at MIT uses force feedback on its leg servos to step over obstacles while it is walking on rough terrain [Brooks 93].

4. Video Camera

Video input has been used as a robot sensor since the first robots were built in the 1970's [Meystel 91]. Video camera technology has the advantage of an extremely high sampling rate. A robot can detect and recognize objects at long range using video images. Modern video cameras now are available in extremely small packages with low power consumption. This supports small robots operating for extended periods in a harsh environment.

One of the primary disadvantages of video sensors is the computational expense required to process the video image. For this reason many robotic systems with video sensors process their video images on a separate ground computer. Video sensors are also susceptible to variations in ambient light. This is a particularly difficult problem outdoors since scene lighting is highly variable. At night, most video systems are useless in the dark. Many modern systems currently use video sensors. The NASA/JPL Mars rover Rocky-V uses five video cameras for perception. The FINALE vision-guided mobile robot system controlled an indoor mobile robot at speeds of approximately 10 meters per minute in the presence of obstacles [Kosaka 93]. This model-based system matched landmarks in the scene with features extracted from the images to perform self-localization. Odometry errors were corrected retroactively once the vision calculation was completed. This system was limited to stop and start motion since the robot had to be motionless to obtain an accurate video image. Additionally, the robot had to be provided with an accurate feature map of the world space. This system was computationally slow, since approximately 27 seconds were required for one cycle of the localization process using a 16 million instructions per second (MIPS) computer [Kosaka 93].

From 1973 to 1981, Hans Moravec at the Stanford University Artificial Intelligence Laboratory developed a remote-controlled, TV-camera-equipped robot called Cart [Moravec 81]. The camera was remotely linked to a DEC KL-10 computer and the computer functioned as both the vehicle controller and as an image processor. The robot used stereo imaging to locate objects and to deduce its own motion. Distinctive features extract-

ed from the video images were used to perform a 3D analysis of the scene in front of the robot. The Cart robot represented obstacles with a map that was an ordered list of line segments and empty areas. The empty areas were represented as convex polygon cells which included obstacle line segments. The system used this map to plan optimum paths that minimized costs in terms of distance and energy requirements [Meystel 91].

The Cart robot was different from the Yamabico project in several important ways. The Cart's principle sensor was a video camera, while Yamabico sensor system uses 12 sonars. Off-board computer processing was required for the Cart, whereas Yamabico is fully self-sufficient with regard to computing resources. Yamabico has the capability to recognize and utilize naturally occurring landmarks for odometry correction.

5. Ultrasonic Range Finders

Ultrasonic sensors have the advantage that they are low cost and readily available. Additionally, many examples exist in nature to demonstrate the effective use of high frequency sound waves for navigation. Bats, for instance, hunt their insect prey using ultrasonics. Their sonar processing is far more sophisticated than any robot's. Their main sonar energies used are in the 30 to 60 KHZ range. They can vary the time between sonar ranging pings to vary their range gate [Nachtigall 86]. Long ping intervals are used primarily for search and shorter intervals are utilitized for the terminal homing phase. Dolphins and other marine mammals use self-generated sound for echolocation of their prey underwater. There is evidence that the returned sounds are used for some kind of pattern recognition [Nachtigall 86].

For mobile robot operating in air, ultrasonic range-finders do, however, have limitations. The speed of sound in air varies with ambient temperature, humidity, and barometric pressure. Since ultrasonic range finders rely on time-of-flight, the variations in ambient conditions can affect range values. An ultrasonic pulse is transmitted by an acoustical transducer and reflected back to the ultrasonic receiver by the nearest obstacle. Any factor that

affects the speed of sound in air, will affect the time of flight of the sound pulse. The speed of sound in fluids can be calculated by the Equation 2.1 [Kinsler 82],

$$c = \sqrt{\frac{\beta}{\rho_o}} \qquad 2.1$$

where β is the bulk modulus of the fluid, c is the speed of sound in air, and ρ_o is the fluid density.

The ideal robot sensor is a pencil-thin, collimated beam that returns an accurate range to target regardless of the sensor beam's angle of incidence. A collimated beam is a focused, parallel beam of transmitted energy. The wavelength of sound is long relative to light, consequently most target surfaces appear to be acoustic mirrors. Accordingly, surfaces not nearly orthogonal to the direction of propagation reflect the signal energy away from the source and the surface is not detectable [Brown 85]. This is the biggest limitation of ultrasonic sensors in mobile robotic.

The piston source emitter commonly used for robotic range sensors tends to emit a wide sound beam. The sound pressure and sound intensity as a function of radial source distance R and angle θ with the acoustic axis can be calculated using the farfield expressions [Dario 86]. The directivity of the piston source may be expressed numerically by their 3, 6 and 10 dB beam widths which are the angles θ at which the intensity has dropped 3, 6, and 10 dB relative to the intensity on the acoustic axis. The sonar beam directivity is dependent upon the geometrical shape of the sound source and the frequency of the sound used. The range value returned by the sensor is the distance to the closest target anywhere within the emitter's sonar beam that returns an echo with sufficient intensity to exceed the threshold of the receiver. A close, strong sound reflector may provide reasonable returns 10 to 15 degrees off of the acoustic axis of the sonar beam. This results in poor overall directionality since the sensor returns only a range value and no precise measure of the direction of the target is available. The position of the target reflector is calculated assuming that it

is centered on the acoustic axis. This results in position errors that, in some cases, exceed 10 centimeters at a range of two meters [Leonard 92].

The target's ability to reflect the incident sound wave is of crucial importance. Soft, sound-absorbing surfaces are poor reflectors. The target's ability to reflect incident sound energy per unit area is called the target strength. A good example of a low target strength object is drapery on a window. Conversely, cardboard boxes have a high target strength. The experimentally determined target strength of various materials is given in Chapter III.

Due to the finite amount of time required to transmit the sonar pulse, ultrasonic range finders have a minimum detectable range. This is due to the fact that the receiver cannot make a range measurement while the transmitter is transmitting. Sonar pulses are typically one millisecond, which causes the minimum sonar range to be greater than 10 centimeters [Dario 86]. Minimum range performance and cross-talk problems can be improved considerably when a separate emitter and receiver are used. Polaroid range finders have a single element that is both the emitter and the receiver. They are the most commonly used sonar system in robotics.

Difficulties during the manufacturing stage may result in a large variation in pulse echo response of two commercial sensors purchased at the same time and said to be nominally identical [Dario 86]. Slight differences in the sound pulse amplitude, pulse length, and pulse shape all contribute to differences between individual emitters. Differences in electrical connections, housing and material defects can also cause detectable difference in receiver performance.

Ultrasonic range finders are large with respect to some of the smaller robots in existence today. For example, the Khepera^{TM} miniature mobile robot at the Laboratorie de Microinformatique in Switzerland is 6.0 cm in diameter and 3.0 cm high [Modada 93]. In contrast, the emitter and receiver package for an ultrasonic sensor typically occupy a volume of about 100 cm^3. Size is not an issue on large military-style robots such as the ALV.

Many smaller research robots use infrared range-finders which are significantly smaller, but have a shorter range. The IS Robotics™ R2 [Brooks 93] robot has five small infrared proximity detectors instead of sonars.

The University of Michigan robot Carmel placed first among ten contestants in the 1992 AAAI Robot Competition. Carmel was a Cybermotion KA2 mobile platform with a ring of 24 sonar sensors. Object detection was performed using a color camera. Carmel used an error-eliminating rapid ultrasonic firing to accomplish fast obstacle avoidance while it navigated [AAAI 92a][Borenstein 92]. For mapping during the contest, Carmel used a global Cartesian system that stored only the location of the poles and the current position of the robot.

While at Oxford University, Leonard performed extensive research on model-based localization and map building using only ultrasonic range finders [Leonard 91]. The research involved the development of a specular sonar model for a rotating sonar sensor. The model predicted that clusters of strong sonar returns formed regions of constant depth (RCD). A RCD was a connected set of sensor returns with range differences less than some predefined range difference threshold. The RCD data was gathered by performing stationary 360° scans at fixed intervals as the robot moved about the world space. No robot exploration was involved since the vehicle moved in a preprogrammed "seed spreader" pattern to gather the sensor data. Leonard used the Robuter robot and the SKIDS robot for his experiments. Ultrasonic range data was then processed off-line. In Leonard's experiments, the aim was for a robot to maintain continuous map contact, "grabbing hold" of corners, planes and cylinders in the environment and then use them as handrails to guide the navigation process [Leonard 91]. Robot motion planning was accomplished off-line using a Voronoi diagram trajectory planner.

Leonard's robots used Polaroid sensors that had a significantly different beam pattern than Yamabico's sensors. Yamabico uses collimated ultrasonic sonar detected with a separate emitter and receiver. Yamabico's sonars tend to have a far narrower beam width and are therefore less prone to directionality problems. Also, Yamabico's sensors have a

collimated receiver that tends to pick up less secondary specular reflections than the Polaroid sensors. Leonard's robot used RCD data to build its map while Yamabico analyzes line segment data derived from sensor data. This results in a lower computational complexity for Yamabico's cartography algorithms. Typically, Leonard's algorithm analyzed over 1000 individual sonar returns [Leonard 91] to build a small map, whereas Yamabico processes about 15 extracted line segments (automatically derived from 1000 individual sonar returns) to perform the equivalent task.

E. ROBOT LOCALIZATION AND NAVIGATION METHODS

For a mobile robot moving in an unstructured environment, maintaining exact position informatic poses a major problem. Over long distances, dead reckoning estimates are not sufficiently reliable, consequently, motion solving methods that use landmark tracking or map matching are usually applied to reduce registration imprecision due to motion [Elfes 87]. There are three basic types of localization dependent upon the map representation: model-based, grid-based, and local composite model [Leonard 91]. In the model-based localization, correspondence is achieved directly between individual observations and the geometric model. For grid-based localization, an intermediate representation is built up from sensor range input and then correlated with the global grid-based map. The local composite model is similar since an intermediate representation is also built from sensor data and matched to the geometric world model.

Elfes performed robot localization using an Approximate Transformation (AT) framework for robot localization with sonar data [Elfes 87][Elfes 90]. A robot motion M is represented as $M = \langle M', E \rangle$, where M is the estimated (nominal) configuration and E is the associated covariance matrix that captures the position uncertainty. E is applied periodically to correct odometry errors.

The Stanford Research Institute's (SRI) robot Flakey placed second among ten contestants in the 1992 AAAI Robot Competition. Flakey was a custom built, octagonal robot with a circular array of 12 Polaroid ultrasonic sensors, an infrared laser and a charged cou-

pled device camera (CCD) [AAAI 92a]. Flakey used a tolerant global map that contained local Cartesian patches related by approximate metric information [Davis 93]. Each of these patches contained a landmark or feature the robot used for localization. During the contest, the walls of the arena were used as landmarks and the approximate length and relative orientation of the walls were given to Flakey as prior knowledge [Davis 93].

Flakey's system was different from the one used in Yamabico since Flakey used tolerant maps. With Flakey, large dead reckoning errors accumulated over just four or five meters of motion, especially when turning was involved. Yamabico has more accurate dead reckoning so its mapping is less error prone. Flakey stored patches of detailed grid-based landmark information linked together by tolerant metric data. These stored patches could be called features. In a way, Flakey used a hybrid map representation method that was grid-based near landmarks and feature-based with respect to landmark locations in the world space. Yamabico maintains a precise global map of the entire known world. Flakey used Polaroid sonars, while Yamabico uses Nippon Ceramic sensors. Finally, Flakey required some knowledge of the world space landmarks, whereas Yamabico does not require *a priori* landmark knowledge to navigate in an unknown environment.

Crowley performed localization by extracting straight line segments from sonar data and matching them to previously stored global line segment data [Crowley 86]. Cox implemented a continuous localization system using the robot Blanche [Cox 91]. Blanche used was an optical range finder sensor. Odometry updates were provided at eight second intervals with this system. Hinkel also implemented a localization system using a laser range finder sensor [Hinkel 88]. Sugihara [Sugihara 87] and Krotkov [Krotkov 89] performed visual position estimation using vertical line segments as features. Using sonar data and grid-based map representations, localization has been performed by matching local occupancy grids with a globally referenced occupancy grid [Elfes 87] [Moravec 87]. Everett used special side-scanning ultrasonic range finders to detect walls and other obstacles in the robot's environment for localization [Everett 93]. Curran used sonar and infrared range finder data to match expected and actual range values for localization [Crowley 89].

F. ROBOT EXPLORATION AND CARTOGRAPHY

If the geometry of the environment is not fully known to the robot system, one must employ an "exploratory" approach in which plan generation is tightly updated to gather data on the environment and to dynamically update the world model [Schwartz 88]. Robot exploration is the process in which a robot incrementally acquires and stores knowledge about the world space through intelligent motion and sensor input. In this section it is necessary to distinguish between two families of exploration schemes: undirected and directed exploration. Undirected exploration techniques explore the environment through randomness. Directed exploration differs from undirected exploration in that the former utilizes some exploration specific knowledge for guiding the exploration decisions [Thrun 92], and actions are chosen based upon the maximum expected knowledge gain.

1. Undirected Exploration

Undirected exploration is an uninformed, random exploration technique. When selecting the next exploration action, no attempt is made to pick the action with the best expected outcome. The cost of the search or the reward associated with finding new, unexplored space is not considered. Actions are selected stochastically based upon a uniform probability distribution for pure undirected exploration resulting in enhanced probability distribution for action selection such that the better the action, the higher the probability. Actions are still selected randomly. Undirected exploration is usually inefficient and the anticipated exploration time normally scales exponentially with the size of the space to be explored. However, undirected exploration has been demonstrated as effective for some applications. The robot Scarecrow used a random walk exploration technique to place third in the AAAI robot contest in 1992 [Dean 93].

2. Directed Exploration

Directed exploration involves robot guidance based upon some specific knowledge or rules for searching. An exploration rule is used to determine the next action that best explores the environment. Directed exploration rules are heuristics since the robot is

exploring an unknown environment. Directed exploration techniques are usually more efficient than undirected techniques in terms of time and energy to explore a given space.

There are three general types of directed exploration; counter-based exploration, recency-based exploration, and error-based exploration. In counter-based exploration, the robot is driven to explore the least visited neighboring state next. Recency-based exploration favors the state which occurred least recently. Error-based exploration schemes make the assumption that states or regions in the state space with large error are little explored and demand further explanation [Thrun 92].

While at CMU, Thrun developed a system to explore and model an office building environment efficiently [Thrun 93]. He used the robot Columbus, a modified Heathkit robot. This was an autonomous, wheeled robot with bumper sensors, a rotating sonar sensor, and wheel encoders for odometry. The exploration system used an instance-based learning technique for developing the map. Two artificial neural networks were used to encode the characteristics of the robot's sensors and the characteristics of a typical indoor environment [Thrun 93]. One neural network encoded sensor interpretation and the other encoded confidence assessment. Exploration was achieved by navigating the robot to regions with little sonar data [Thrun 92]. The world model was represented using a grid-based mapping technique with four inch grid squares.

G. SUMMARY

This chapter has provided an overview of the major challenges and issues with regard to mobile robot cartography. Map representation methods are primarily methods for a robot to store the world model information gained through sensor perception or prior knowledge. The size of the world model is a major factor since robot memory is a limited resource and restricts the size or resolution of the world the robot can understand. Additional world model considerations are complexity and resolution. Robot motion planning issues are addressed since a robot must move intelligently about in its environment to acquire sensor data to build a map. Motion planning is essentially a process that takes the current world

model as an input and gives a robot motion plan as an output. Software architectures for robotic vehicle control are tied to the robot's morphology, mission, and map representation techniques. Hierarchical control gives a robot intelligence through centralized deliberation. These systems, however, are characterized by slow response to environment stimuli. Behavior-based control gives a robot rapid, reflexive response to environmental stimuli, but limited intelligence. Hybrid control architectures represent an effort to combine the most desirable features of hierarchical and behavior-based control.

The issues regarding robotic perception are reviewed by sensor type. Laser and light-type active emitter range finders have high data rate and give data relatively independent of ambient light levels. Contact sensing is limited by a robot's ability to physically reach out and touch the environment. Video camera techniques have a high sampling rate, but are limited by the computational overhead required to process video images. Ultrasonic range finders are extremely popular with mobile robotic projects due to their low cost and availability. The primary limitations are low sampling rate due to the speed of sound in air and the limited target incidence angle problem.

Robot localization periodically corrects robot dead reckoning errors. The primary method of localization in mobile robotics is triangulation. Basically, sensor input is matched against some internal world model. The difference between expected sensor input and actual sensor input is used to derive the dead reckoning error. Robot exploration is necessary to transport the robot to all reachable portions of its world space for cartography. This is essentially a special purpose motion planning problem. The two basic methods are undirected exploration and directed exploration. This chapter reviewed the important issues and challenges of robot cartography to set the stage for the research that follows in the following chapters.

III. THE YAMABICO-11 MOBILE ROBOT

Yamabico-11 is an experimental, wheeled, autonomous mobile robot located at the Department of Computer Science, Naval Postgraduate School. Yamabico provides a test bed for robotic experiments and control theory development. This chapter describes the robot hardware, and programming environment. A graphic-based mobile robot simulator developed as a part of this research is also described. The simulator has been used extensively for prototyping new control algorithms and as a teaching tool for the Advanced Robotics course at the Naval Postgraduate School.

Future robots are expected to possess advanced capabilities of sensing, planning, and control enabling them to gather knowledge about their environment. This knowledge will be stored as a model for planning and carrying out tasks sent to them in high level style by an applications programmer [Schwartz 88]. Yamabico represents this spirit of robotic system development.

A. THE NAVAL POSTGRADUATE SCHOOL YAMABICO-11

1. Hardware Description

Yamabico-11 is an autonomous mobile robot powered by two 12-volt batteries and is driven on two main wheels by separate 35 watt DC motors. Yamabico is pictured in Figure 3.1. These motors drive and steer the main wheels while four shock absorbing caster wheels balance the robot. A VME card cage holds up to eight 6U-type Euroboard VME cards for on board computing hardware. The VME cage has a fan to dissipate heat from the computing boards. An Apple Macintosh Powerbook 145 notebook computer with an Articulate Systems Voice Navigator voice interface is provided for user communications with the robot.

An Ironics Sun3 single-board computer is the main processing unit. The master processor is an MC68020 32-bit microprocessor accompanied by an MC68881 floating

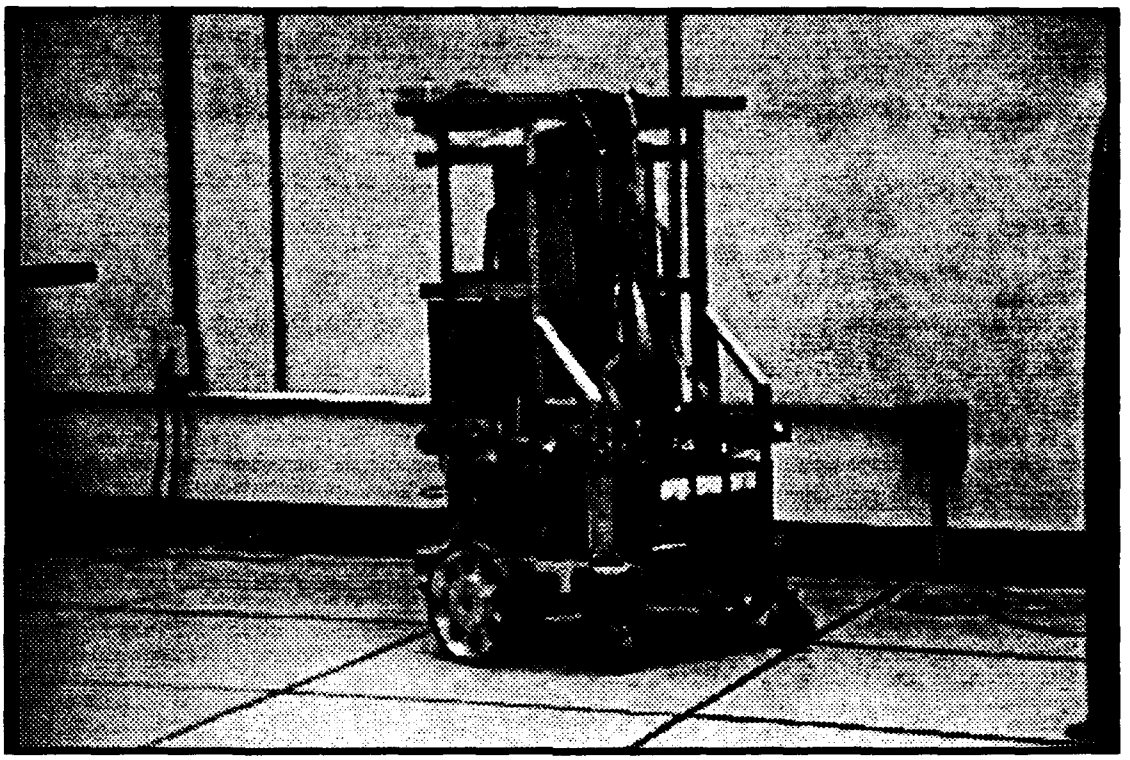


Figure 3.1 Yamabico 11

point unit on a Mizar VME7120 board. This processor has one megabyte of main memory and runs with a clock speed of 16MHz. The processor has a bug monitor in ROM supplied by the manufacturer for basic programming and debugging.

2. Sensor Characteristics

The Yamabico sonar system consists of a sonar ranging board and a sonar array consisting of twelve Nippon Ceramic T40-16/R40-16 ultrasonic range finder emitter/receiver pairs arranged around the robot's perimeter. The ranging board is an Omnibyte OB68K VME I/O board that is controlled by an 8748 microcontroller. The sonar board is a separate input/output controller that makes the overall sensor process more efficient since the main central processing unit (CPU) does no sonar processing. This device has software programmable interrupts. This board takes the distance measurements from twelve sonar

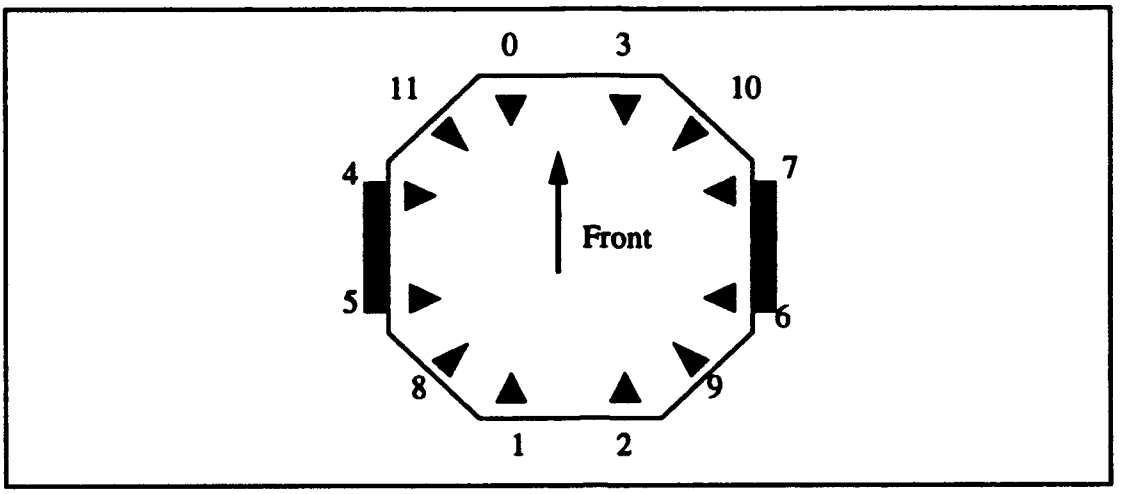


Figure 3.2 Yamabico-11 Sonar Array

sensors and presents the data to the CPU on the Yamabico robot. The sonar hardware design gives a range gate of 409 centimeters and a range resolution of 1 millimeter.

The Yamabico sonar array is illustrated by a top view of the sensor's positions in Figure 3.2. The sonar sensors are arranged in three logical groups, with four sensors in each group. Group 0 consists of sonars 0, 2, 5, and 7; group 1 consists of sonars 1, 3, 4, and 6; group 2 consists of sonars 8, 9, 10, and 11; and group 3 is a virtual group which consists of four fixed test values [Sherfey 91]. Ranging is done on a group basis to prevent mutual interference. All four sensors in a given group range at the same time. Ranging takes place independent of the VME bus CPU. The sonar system completes its measurement of a given group then generates a VME bus interrupt. The VME bus CPU reads the data from the four sensors in the group from registers on the sonar board. After the CPU reads the sonar data, the sonar system begins ranging measurements on the next group. The VME CPU selects which sensor group is active by writing to a command register on the sonar board. The sonar board individually controls the sonar ranging among the three sonar groups in the sonar array.

The sonar transducers operate at a constant frequency of 40 kilohertz. Assuming that sound travels in air at 340 meters per second, the time for sound to travel one millimeter

is 1 millimeter /(340000 millimeter/second) = 2.94 microseconds. Since the time for round trip travel is measured, each millimeter of range gives a 5.88 microsecond delay. The counter used to generate a one millimeter count must have a frequency of 1/5.88 microseconds = 170 kHz.

A sonar ping is a 40 kHz burst lasting 500 microseconds that is emitted by all four sonar emitters in an active group. Starting at the leading edge of the sonar ping, the ranging counter starts counting using a 166.7 kHz clock signal. Each clock cycle, the counter value is written to four 16 bit registers. Each register corresponds to one of the four sonars in the active group. After each counter is incremented, each sonar receiver is checked to see if a signal has returned. When a signal is detected, the memory address for that sonar is locked out to prevent any further writes to the counter. This effectively records the range in millimeters to the first return exceeding the receiver threshold. After 4096 counts, the counter halts. Each memory location contains an integer representing the sonar return in millimeters. If no return is detected, the most significant bit is set in the memory on the 4096 count signaling an overrange condition. The end of the counting generates an interrupt on the VME bus CPU. The VME bus CPU performs a serial I/O transfer by reading the four memory addresses containing the four range values. After the last address is read, the sonar system begins ranging on the next selected group. Each reading cycle takes approximately 24 milliseconds.

Ultrasonic range-finders have limitations. Since ultrasonic range finders rely on sound time-of-flight, the variations in ambient conditions of air can affect the reliability of range values. The speed of sound in air varies with ambient temperature, humidity, and barometric pressure. Therefore, the range value returned by the range finder can vary considerably based upon ambient conditions.

Acoustic sound waves are limited by the speed of sound in air. The speed of sound in air is only 343 meters per second at 20° C at sea level [Kinsler 82]. Air temperature, pressure, and humidity affect this value. The time required for sound to travel a round trip from the emitter to the target and back determines how often the sensor can obtain range data.

Since Yamabico's programmed maximum range gate is 409 cm, the frequency of the emitter's ping can be determined by using 8.18 meters as the maximum round trip distance and the speed of sound in air at 20 degrees C. The robot's ping interval for a single sonar can be computed using this round trip distance as follows:

8.18 *meters* * 343 *meters/seconds* = 0.0238 *seconds* ≡ 24 *milliseconds*.

Given this range gate, a single sensor can receive 1/0.0238 seconds = 41.67 range readings per second. Obviously, the choice of a shorter range gate allows for a higher data rate. Consequently, sound based sensors have a data rate limited by the speed of sound in air when a fixed sensor range is desired.

The ideal robot sensor is a pencil-thin, collimated beam that returns an accurate range to any object it is pointed at regardless of the angle of incidence. A collimated beam is a focused, parallel beam of sound energy. The acoustic wavelength is long relative to light, consequently, most target surfaces act as acoustic mirrors. Since most surfaces act as acoustic mirrors, some of the incident sonar energy is reflected away from the sonar receiv-

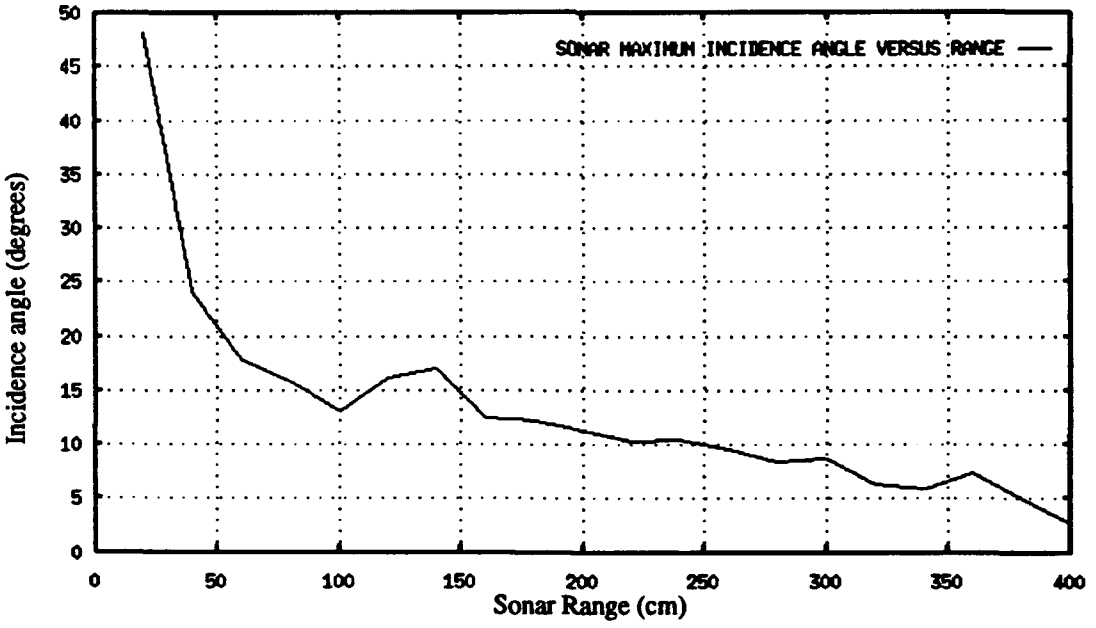


Figure 3.3 Sonar Incidence Angle versus Range

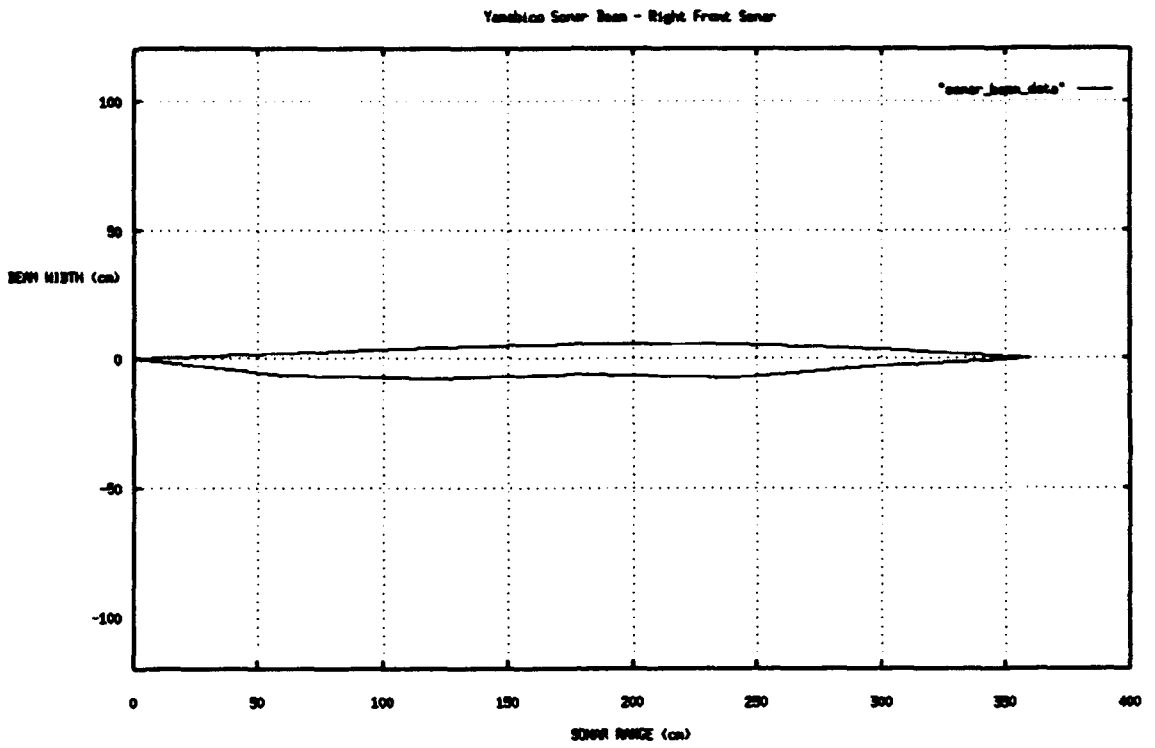


Figure 3.4 Yamabico Sonar Beam Width

er by the target. This effect become worse with increasing range and increasing angle from the normal to the target surface. In Figure 3.3 the sonar beam maximum incidence angle for a valid reflection versus target range is plotted. Sonar is a non-ideal sensor since the sonar's beam must be nearly normal to the target's surface in order to obtain a valid range return. Accordingly, surfaces not orthogonal to the direction of propagation reflect the signal energy away from the source and the surface is not detectable [Brown 85]. This is the biggest limitation of ultrasonic sensors on mobile robots. On Yamabico-11, target surface must be within approximately 15 degrees of normal to the incident sonar beam in order for sonar to return a range to the target.

Yamabico's sonar transmitters have beam collimators to focus the sound beam. This results in good directionality but imposes limitations on specular returns. The Yamabico sonar beam is much narrower than the Polaroid sensors. Figure 3.4 shows experimen-

tally derived sonar beam width data from one of Yamabico's sonar sensors. Notice that the sonar beam is widest between 100 and 250 centimeters due to spreading losses.

The target's ability to reflect the incident sound wave is of crucial importance. The target's ability to reflect incident sound energy per unit area is called the target strength. Soft, sound-absorbing materials have poor reflectance. A good example of a low target strength target is drapery on a window. In contrast, cardboard boxes have a high target strength.

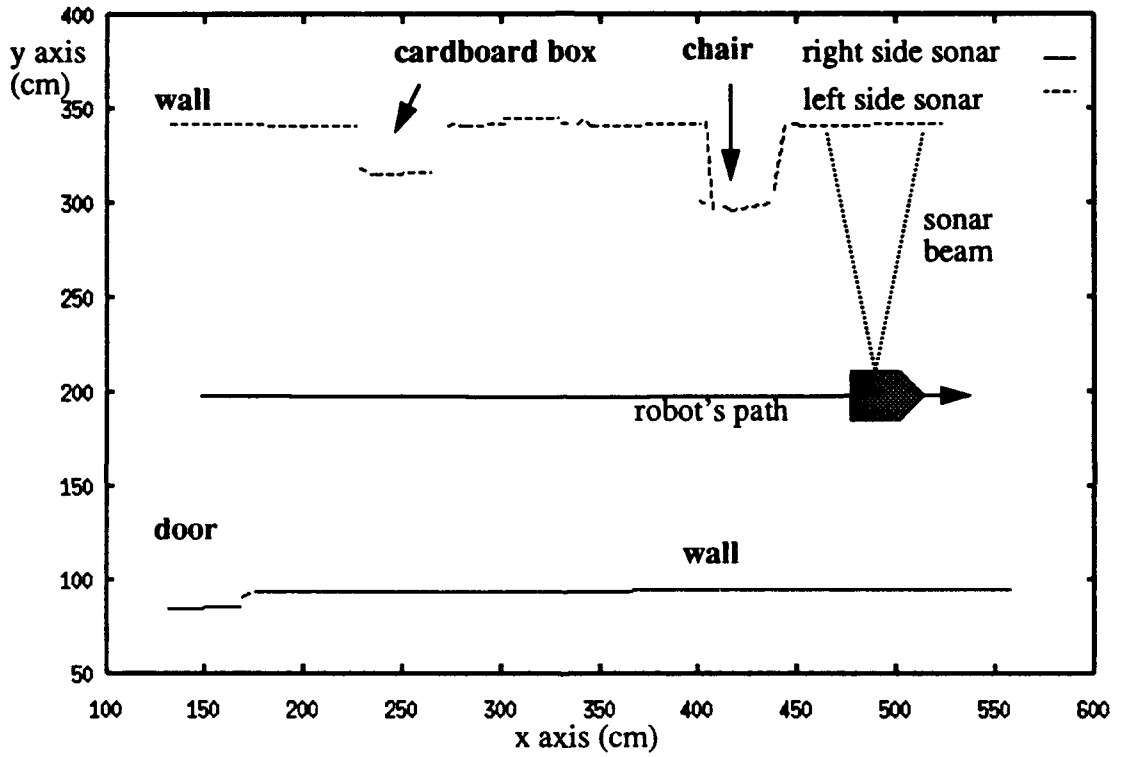


Figure 3.5 Yamabico Sonar Scan Data

Figure 3.5 illustrates Yamabico's sensor data extracted from four meters of robot motion. The robot moved down the center of a hallway while scanning objects on both sides with its range finders. Objects found in a typical office environment were placed along the left hand wall. The targets included a cardboard box and a wooden chair. Notice that only

line segment data is extracted from the sonar scan. Yamabico does, however, have the ability to return global position data from the individual sonar returns [Sherfey 91].

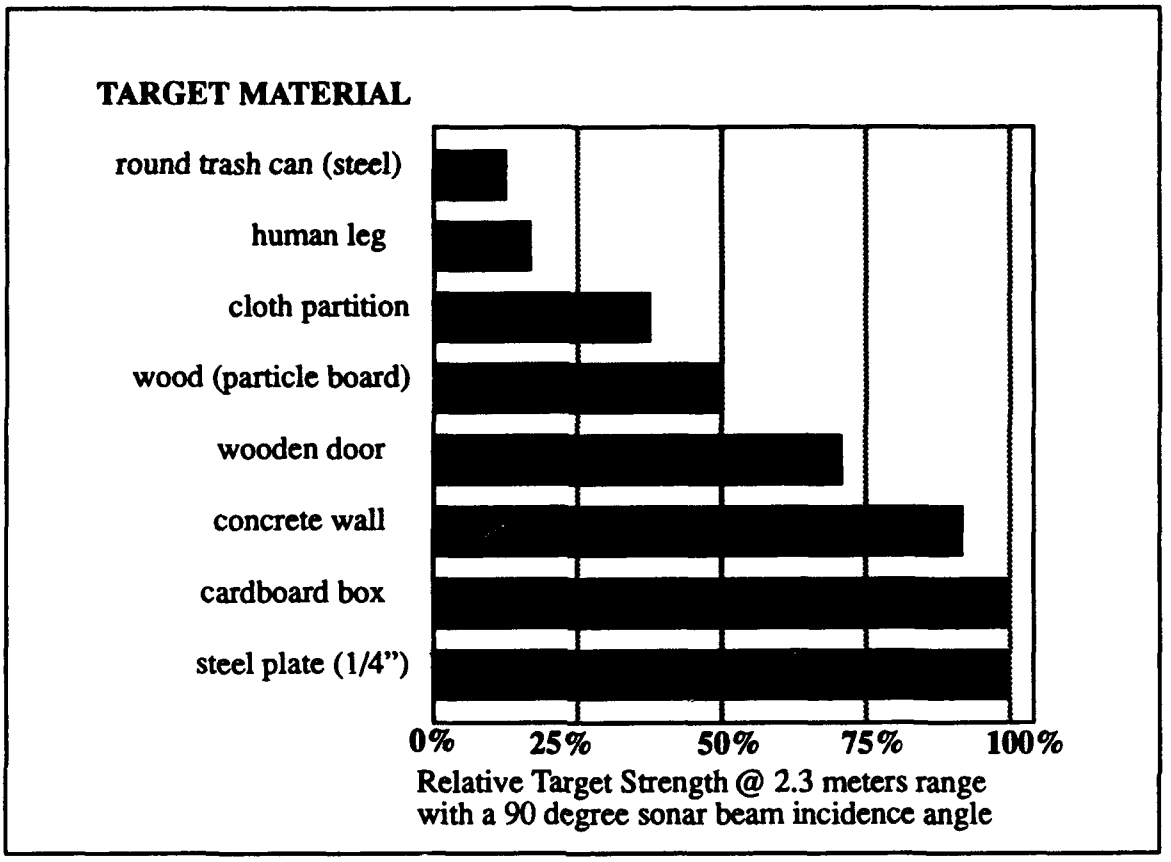


Figure 3.6 - Relative Material Target Strength

Figure 3.6 shows a bar chart of experimentally determined target strength data for steel, wood, concrete, cardboard, and other objects using Yamabico's ultrasonic range finders. The amplitude of the return from each object was measured using an oscilloscope connected to one of Yamabico's sonar receivers. The maximum return at the receiver was three volts. The target strength was expressed as a percent of the maximum return at the receiver.

Polaroid sensors do not have separate emitter and receiver units. Yamabico-11 uses a separate emitter and receiver. The pulse length is 500 milliseconds and the minimum range is 9.3 centimeters. This feature is important for precise navigation through cluttered indoor areas.

B. YAMABICO-11 ROBOT SIMULATOR

The purpose of the Yamabico simulator is to allow for robot software development and testing without programming the robot. At the AAAI 1992 Fall Symposium Series the consensus of the group focusing on AI and mobile robotics bears out the importance of this approach.

> For robot navigation in an office environment... simulation to perform a large number of experiments economically and physical robots to verify that the simulated results hold up in reality is the best approach [AAAI 92].

The simulator provides an X Windows graphics display to the computer screen to allow the software developer to determine if robot motion is correct. The simulator is written in a portable language to facilitate transfer to other host computers. The system runs on a variety of computers with MIT X-Windows [Johnson 92] and a 'C' compiler. The time expended in building this simulator was well compensated by the time saved in software development. The simulator runs in faster than real time. This allows for simulating long robot experiments more quickly than the actual robot run time. Simulations run on an independent workstation, so the software developer is not subjected to a number of limitations including battery life (currently Yamabico-11 lasts about six hour on a battery charge), maneuvering space (lab space is tight), availability (Yamabico is a one-of-a-kind robot with a dozen software developers involved in programming), and convenience (the developer can test new algorithms anywhere an appropriately configured workstation is located). Experiments can run overnight or over a weekend. The results can be quickly evaluated without a human operator having to physically watch the robot. A 2D graphics display is sufficient to allow the software developer to evaluate the robot's behavior. This simulator was developed on a Sun workstation since these are less expensive and are generally more available than special purpose graphic workstations.

The simulator has also served as a teaching tool in the Advanced Robotics course offered at the Naval Postgraduate School. Students learn more efficiently when they can first practice their programs on the workstation before trying them on the actual robot. The de-

mand for experimental time on the robot is greatly reduced since most application development time is spent testing robot code on the simulator.

1. Design Goals of the Yamabico Simulator System

The Yamabico simulator's primary goal is to faithfully execute all commands that the robot executes. This is to include all sonar returns that would be received by the real robot and specifically it should accurately model the ultrasonic sensors. Also the robot's multi-tasking system must be faithfully modeled in simulation; this alone is a challenging goal in a hard real-time system.

The simulator uses the same "user.c" file that could be compiled and run on the robot hardware with no modifications. Additionally, the same code should be used for the robot and the simulator where possible. Compiler flags are used to switch between simulator and robot code when necessary.

Robot motion should be shown on the workstation screen frequently during the test run so the developer can see the path the robot is taking. This allows developers to quickly test new software and watch a five sided robot symbol move about in the simulated world space. The above goals were adhered to as closely as possible during the simulator's development.

2. Simulator Top Level

The top level of the Yamabico simulator is the main menu display that is shown in Figure 3.7. It is a graphic screen device that allows the simulator user to select the next simulator function. This graphic main menu device was developed using NASA's graphical user interface toolkit TAE 5.1 [TAE 90]. The CMPL button is the compile button. When any portion of the software is changed, this button can be used to compile and link the software. This button invokes an UNIX makefile which recompiles all modified code files. The EDIT button invokes the "vi" editor in the current directory for the file user.c. This allows the user to edit the "user.c" command file and quickly recompile the code.

Figure 3.7 Yamabico Simulator Menu

The RUN button is used to run the simulator. This starts the program which displays a graphic of the robot's world space. The robot's configuration is plotted symbolically five times per second. The real-time plot of robot motion is more fully described in the next section and an example appears in Figure 3.8. The elapsed time and the robot's configuration are displayed numerically on the screen.

The PLOT button allows the user to see a complete plot of the most recent robot mission. This button invokes the "gnuplot" program [Williams 92] which plots the robot's entire trajectory and any sonar data obtained during the robot's last mission.

The INFO button displays a help file that gives instructions for new users. The EXIT button allows the user to quit from the simulator. An additional screen display provides the contents of the instruction buffer for the user. This is particularly useful for debugging Yamabico system code.

3. Utility of a Robot Simulator

The simulator motion plot is show in Figure 3.8. This is an X Windows application program design for rapid prototyping and analysis of robot control algorithms. This plot displays the robot's trajectory as it executes the user's command file. The robot's current configuration is displayed in the upper left hand corner of the display. An outline of the robot's world space is provided to allow the user to determine the robot's current location in the world space. The world space can be easily reconfigured by changing a world input specification file. The robot is plotted as a five sided icon every 20 vehicle control cycles. This is equivalent to five times per second. Depending on the speed of the host com-

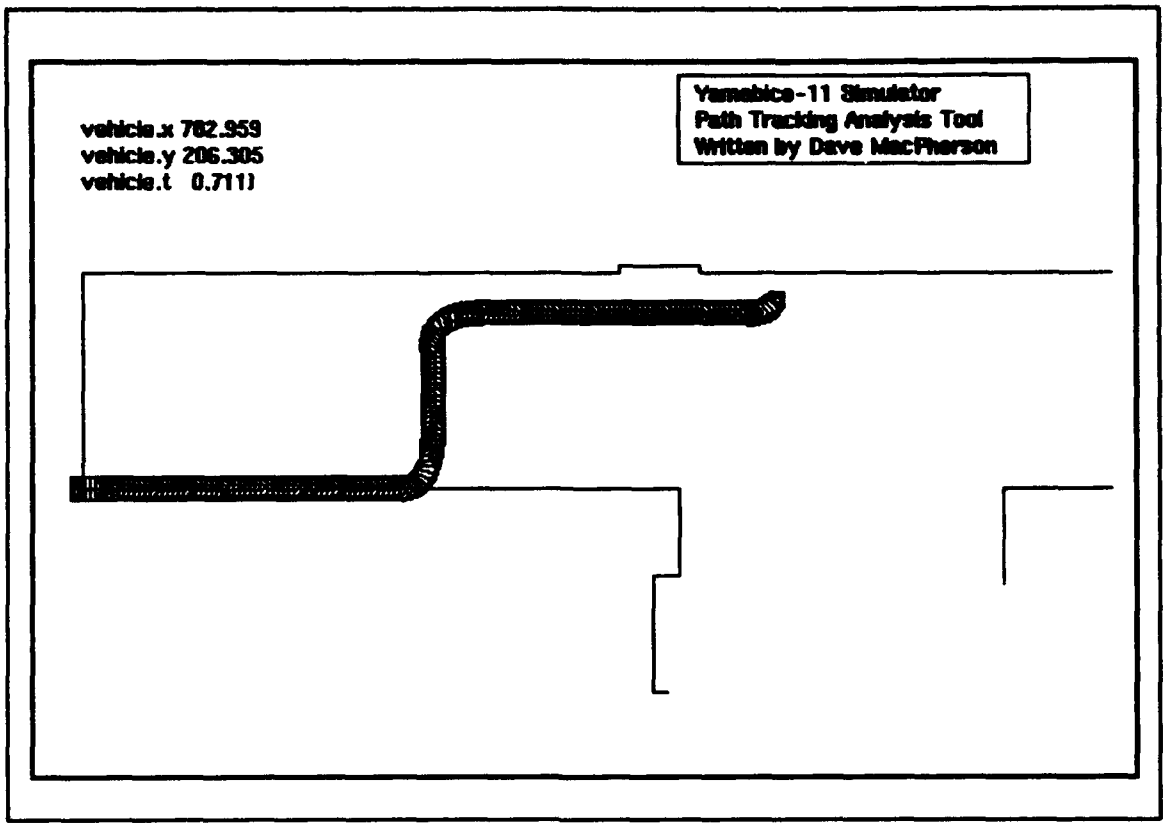


Figure 3.8 - Simulator Motion Plot

puter for the simulator, the robot executes the motion commands much faster than the actual robot. Many MML programs have been developed for Yamabico using the simulator to test the robot trajectory first.

The instruction stack for a typically "user.c" command file is shown in Figure 3.9. The robot's commands appear in the first column. Then the following columns show the commanded geometry for the command. For path elements this geometry specifies the configuration of each path element. The path element tracking is more fully explained in Chapter IV and Appendix A.

Practically all robotic projects have some kind of simulator to provide an environment for software development, thus allowing software developers to develop and test new software modules without physically testing them on the robot. Students in the advanced robotics course have conducted simple simulation experiments in order to learn the MML system. The simulator also allowed researchers in the MML design group to develop ad-

vanced robot control systems in simulation. One example is the CLIPS Yamabico simulator designed by Fish for obstacle avoidance [Fish 93]. Another examples involves the simple automated cartography experiments used to build a symbolic map of the robot's world space in simulation.

```
Instruction Stack :
Class     pc.x     pc.y     pc.t     pc.k     tp.x0    tp.y0   mode1   mode2
SET_ROB   0.00    -30.00    0.00     0.00     0.00     0.00     0       0
SIZE     15.00     0.00     0.00     0.00     0.24     0.24     0       0
SPEED    60.00     0.00     0.00     0.00     0.00     0.00     0       0
LINE      0.00     0.00     0.00     0.00     0.00     0.00     0       0
PARABOLA  0.00    80.00     0.00     0.00   298.17     0.00     0       0
LINE      0.00     0.00     0.00     0.00   460.16    19.84     0       0
LINE    700.00     0.00     1.57     0.00   671.88     0.00     0       0
LINE    300.00   200.00     3.14     0.00   700.00   171.87     0       0
SLINE   300.00   100.00     0.00     0.01   433.75   200.00     0       0

Total Number of Instruction : 9

SLINE
Elapsed Time:   6.63 seconds
SPARABOLA
Elapsed Time:  10.52 seconds
SLINE
Elapsed Time:  14.39 seconds
SLINE
Elapsed Time:  17.69 seconds
SLINE
Elapsed Time:  22.57 seconds
SBLINE
Elapsed Time:  60.00 seconds
```

Figure 3.9 Robot Instruction Stack From Instruction Buffer

4. Simulator Sonar Model

The simulator's sonar model is shown in Figure 3.10. A simple ray tracing type algorithm that simulates the expected range finder returns from Yamabico's ultrasonic sonars is used. The sonar model consists of 12 virtual sonar beams represented as line segments. Each beam starts at the global position of the corresponding sensor and ends 4.1 meters from the sensor on the sonar beam's main axis. This distance corresponds to the maximum range of the real sonar sensors. The robot's world is modeled as a doubly linked

list of line segments that represent the boundaries of the robot's world. This world is a single polygon in which the robot operates. No collision modeling is included since it is unnecessary to support automated cartography experiments.

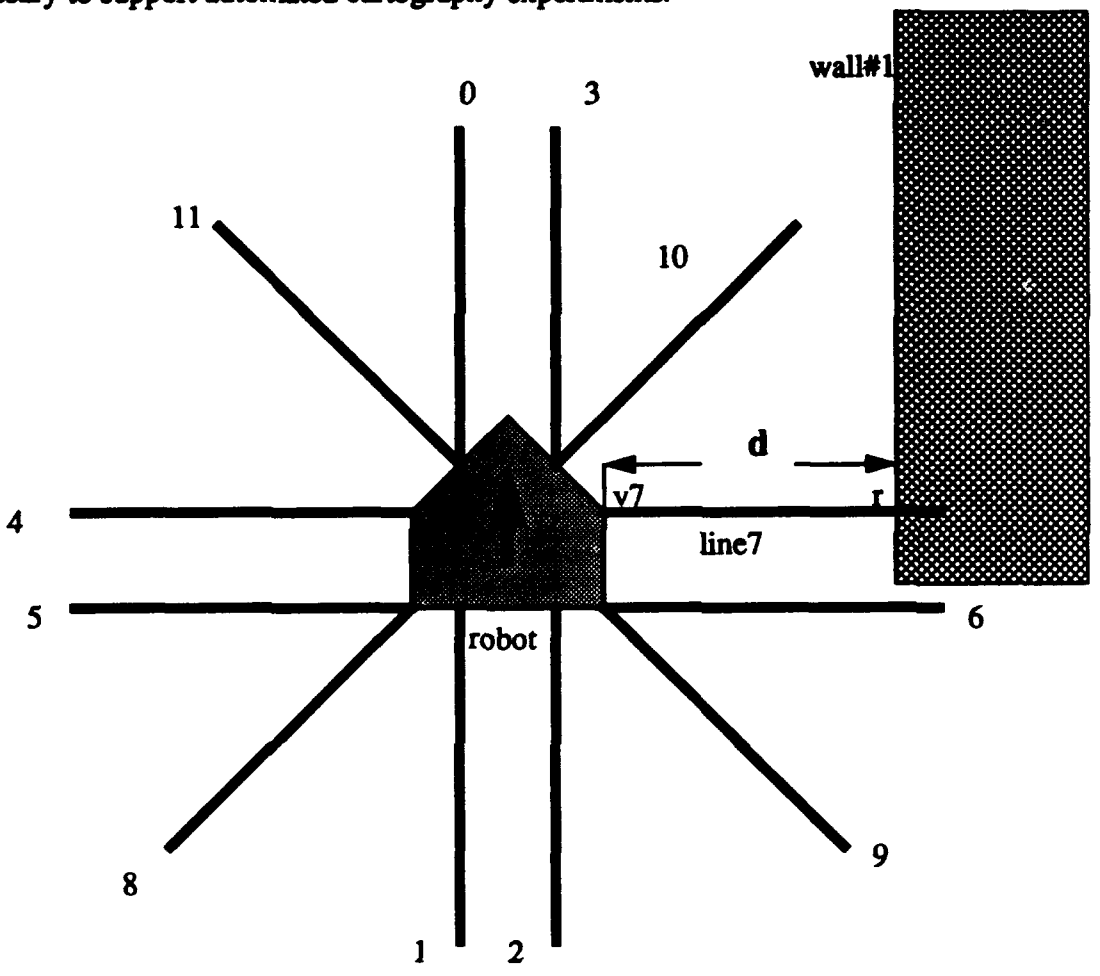


Figure 3.10 Yamabico Sonar Model

The ultrasonic sonar beam is a simple line segment. The algorithm does a simple segment crossing test and returns the sonar range to the intersection of the sonar beam and world space surface. An incidence angle of ±15° from a normal to reflecting surface is required for a valid range return. As the simulator moves throughout the simulated world space, the sonar beam line segments from the enabled sonars are tested to check if they cross any portion of the world model. If a beam segment and a world segment cross, then

the distance from the robot's sensor to the crossing point is determined. The sonar beam incidence angle is computed then a range value is returned only if the sonar beam's incidence angle is between 75 and 105 degrees. Sonars ping in groups just as the real robot sonar. When one or more sonars are enabled, the simulator pings a group of sonars every three vehicle control cycles. This correctly simulates the actual vehicle sonar cycle of 24 milliseconds.

5. Simulator Fidelity

Vehicle motion is modeled accurately by the vehicle simulator. There is, however, no vehicle dynamic model so a simple kinematic model is used. Ultrasonic sonar is modeled in a simple fashion and accurately produces the same messages to the laptop computer interface as the robot software.

The multi-tasking system that is interrupt driven on the real robot is not faithfully modeled in the simulator. As a result, some temporal ordering is improper. Specular reflections are also not modeled, and only the range to the primary reflection from the closest virtual surface is returned. The true shape of the sonar beam is approximated by a straight line ray. This is a relatively good approximation based upon Figure 3.4. Researchers have done extensive study on the physical nature of the ultrasonic sonar beam [Kuc 87] but no good specular model exists in the literature.

C. SUMMARY

The Naval Postgraduate School robot, Yamabico-11, is introduced in this chapter to provide a basis for comparison. The hardware characteristics are introduced to provide the reader with sufficient background for the next chapter. This robot is the test bed odometry correction theory in Chapter V and the automated cartography studies developed in Chapters VII and VIII. All experimental results are reported in Chapter IX.

The Yamabico simulator is an important tool for robot software development in the MML project. A simple, but valid, sonar model is included to develop model-based mobile robot navigation algorithms and automated cartography. It has also be used as a teaching

tool in the Advanced Robotics course at the Naval Postgraduate School. This simulator was first used to refine the path tracking algorithms for Yamabico. The simulator allows the software developer to quickly test modifications to the MML language without operating the robot. A plot of the robot's current location is displayed on the workstation screen at regular time intervals. A final full trajectory plot is displayed at the end of the simulation run.

IV. YAMABICO SOFTWARE ARCHITECTURE

This chapter describes Yamabico's software architecture that was developed in part to facilitate automated cartography. Specifically, this architecture provides facilities for task scheduling, resource allocation, spatial reasoning, vehicle motion control, sonar control and input/output functions. The geometric module provides spatial reasoning utility functions that support higher level robot behavior such as path tracking and dead reckoning error correction. The intended path of Yamabico is specified by a series of path elements. The motion control subsystem provides a user interface for controlling vehicle locomotion by tracking geometric path elements. The sonar subsystem controls Yamabico's sensor hardware through a library of 'C' functions. Sonar data collection and processing are accomplished in real time using these functions. The input/output subsystem is crucial for mobile robot troubleshooting and analysis. This subsystem provides functions for robot two-way data transfer between Yamabico and either a host computer or an on board laptop computer.

A. TASK SCHEDULING

As Yamabico's control system, MML is a multitasking operating system that provides robot motion and sensor functions, allocates processing resources, and performs odometry functions. The various required tasks are assigned an appropriate priority depending upon their relative importance. Higher priority tasks will interrupt one or more running lower priority tasks when required. This system is an effective implementation of a "round-robin priority queue." An explanation of the operating system task scheduling is necessary. The Motorola 68020 CPU has eight interrupt levels [Motorola 85]. Some of these interrupts are used to run vehicle tasks at various priority levels in the single CPU, multi-tasking system. Table 4.1 illustrates these vehicle tasks. The higher the interrupt level, the higher the priority of the associated task. At the highest level is Yamabico's reset button. This tasks over-

rides all other tasks, stops the robot and resets the CPU. Interrupt levels five and six are currently not used.

Interrupt level four is the highest-priority task that runs during robot operations. This important task is responsible for steering the vehicle. Every 10 milliseconds, the locomotion task interrupts all other lower priority running tasks and runs for approximately 2500 microseconds. This task first reads the shaft encoders and computes Yamabico's odometry configuration estimate. This is a dead reckoning technique since only internal devices are read. All path tracking computations are performed at this level. Next, the most recent odometry configuration is used to calculate the proper curvature (κ) and velocity (v) for the vehicle. These parameters are used to determine the desired vehicle rotational velocity (ω). A kinematic function calculates the desired left (v_L) and right (v_R) wheel velocities. This information is used to determine the necessary pulse width modulation commands for controlling the left and right wheel drive motors.

Table 4.1 MML SYSTEM TASK PRIORITY

Interrupt Level	Interrupt Source	Function	Interrupt Type	Vector	Duration (μs)
7	stop button	reset	asynchronous	-	-
6	-	not used	-	-	-
5	-	not used	-	-	-
4	Serial Board 1	locomotion	synchronous	64	2500
3	Serial Board 0	teletype	asynchronous	65	variable
2	Sonar Board	sonar	synchronous	66	240
1	Serial Board 0	debugger	synchronous	67	-
0	-	user's instruc	none	-	-

The vehicle's notebook computer interface input/output task runs at interrupt level three. This task is responsible for printing information to the vehicle's on board monitor

and reading input from the user entered on the laptop computer's keyboard. Also, file transfer from the robot back to the host computer is controlled by this task.

The vehicle sensor functions run at interrupt level two. This interrupt is triggered by range information that is placed in the sonar board register. When one or more vehicle sonars are enabled, this transfers data from the sonar board back to the main CPU at 24 millisecond intervals. When none of the robot's twelve sonars are enabled, this task is disabled. Interrupt level 1 provides a debugger task which may be enabled to print status information to the on board computer when a change occurs.

Interrupt level 0 is the lowest priority task. All other tasks can interrupt this task. This task reads the user's functions from an input file *user.c* and fills the command buffer based on the user's sequential commands and modifies system parameters based upon immediate commands. These commands are explained in greater detail in Appendix A. The sonar sensors are enabled and disabled at this level. Additionally, all of Yamabico's navigation functions run at this level.

B. GEOMETRIC MODULE

Yamabico's geometric module provides mathematical support for many required spatial reasoning tasks. There are three important components in this subsystem; assignment functions for specifying geometric variables, math utility functions for manipulating the geometric variables and path tracking geometric support functions for reasoning about path elements.

1. Definition Functions

The definition functions are a collection of 'C' functions used to specify geometric variables. These variables are essentially records containing several floating point parameters. The definition functions specify vehicle configuration variables as well as path element variables. A configuration variable represents an object's configuration in the global coordinate system using a four element record. Path elements are represented using ei-

ther a four parameter configuration variable or a five parameter parabola variable. Appendix A provides additional details and examples of each definition function.

2. Functions

The path utility functions provide a library of routines for the algebraic manipulation of geometric variables. For example, the *composition* function is used to perform 2D transformations and the *inverse* function determines the algebraic inverse of a given configuration. These functions support algebraic manipulations for automatic dead reckoning error correction as described in Chapter V. Also provided are an assortment of utility functions for spatial reasoning math on board Yamabico. Examples include three types of *normalize* functions and a *ceiling* function. All of these utilities support path tracking vehicle control.

The path tracking geometric support functions serve to connect individual path elements for smooth vehicle motion. This subsystem is composed of two types of functions which are related. The intersection point functions determine the crossing point of two sequential path elements. These functions have also been adapted to handle the transitions between non-intersecting path elements. The leaving point function calculates a proper departure point for Yamabico from one sequential path element to the next [Alexander 93]. These functions are explained in more detail in section C of this chapter and in Appendix A.

C. MOTION CONTROL SUBSYSTEM

Precise motion control using the path tracking method of vehicle guidance is essential for accomplishing automated robot cartography. Yamabico maintains a record of it current location using distance information provided by its optical wheel encoders. A current odometry configuration is crucial for path tracking and automated cartography. Precise robot motion control is accomplished. The path tracking method of robot vehicle control is a part of the Model-Based Mobile Robot Language (MML) developed principally by Kanayama [Kanayama 91a]. Basically, this method allows Yamabico to move by tracking straight lines, circular arcs, parabolas, and cubic spirals. This control method smoothly

guides Yamabico during real-time, dead reckoning error corrections. Corrections result in smoother motion when Yamabico tracks a reference path element instead of a reference configuration [Kanayama 93].

1. Odometry Capability

Yamabico's software system maintains an estimate of its current configuration in a configuration variable q_o which is called its odometry estimate. The odometry estimate is updated each vehicle control cycle using information obtained from Yamabico's optical wheel encoders. A small set of user functions are provided for three purposes; (1) set Yamabico's initial configuration at the start of a user.c program, (2) read the current value of Yamabico's odometry estimate and, (3) update the current odometry estimate. These functions provide automatic vehicle odometry correction capability for Yamabico. The theoretical details appear in Chapter V. The functions that accomplish these tasks are more fully explained in Appendix A.

2. Path Tracking

Path tracking means that Yamabico's intended path is specified by a series of geometric path elements. Yamabico software control system is extremely convenient for the user since MML automatically calculates the appropriate transitions between sequential path elements. This frees the robot programmer to focus on higher level robot tasks such as path planning and strategic motion control. To reduce the overall complexity of the system, only certain geometries and path sequences are allowed.

Previously, MML used a reference configuration model to steer the vehicle. Early experimental work for this dissertation on robot odometry correction revealed problems with this control model. Odometry resets that resulted in large changes in the current configuration caused non-smooth, jerky motion. These corrections sometimes resulted in a temporary direction reversal by Yamabico. This problem was particularly severe when the new odometry position fell behind the robot. A dead reckoning reset to a position behind the vehicle caused the vehicle to back up to regain the correct configuration on the Carte-

sian plane [Kanayama 93]. Yamabico also was programmed to accelerate to a higher speed than the current operating speed in cases where the reset configuration was ahead of the current configuration. These types of corrections required Yamabico to "catch up" to the correction configuration. This acceleration together with poor control of Yamabico's instantaneous path curvature caused unacceptable wheel slippage that resulted in increased odometry error. This non-smooth motion control was unacceptable for automated cartography.

A better way to specify robot motion is through a series of planar path elements that serve to define Yamabico's intended path. Automatic transitions between path elements provides smooth vehicle motion along the intended path. The available path elements include straight lines, arcs (constant curvature portions of a circle), cubic spirals, or parabolic line segments. One advantage of path tracking is the vehicle odometry reset are performed with respect to a reference path element instead of a reference configuration. This method smoothly guides Yamabico along the specified path when the odometry estimate is reset. No change in speed is required to catch up to a reference configuration. Yamabico corrects its tracking with respect to a linear reference path. This allows it to maintain constant velocity as it follows the intended path. The overall wheel slippage is reduced since the vehicle can maintain a constant velocity after an odometry reset.

Smooth path tracking is accomplished using the steering function $\frac{d\kappa}{ds}$ which controls Yamabico's instantaneous path curvature. Since Yamabico's configuration is represented in terms of x, y, θ, and κ. The steering function is given by Equation 4.1.

$$\frac{d\kappa}{ds} = f(x, y, \theta, \kappa) \qquad 4.1$$

A signed distance value y^* is used to represent the shortest distance between Yamabico's current configuration and the reference path. The sign of y^* depends on Yamabico's position relative to the reference path. When $y^* > 0$, Yamabico is to the left of the reference path and $y^* < 0$ means it is to the right. Yamabico's configuration projected onto

the reference path is called the image. Yamabico's steering function can now be represented as

$$\frac{d\kappa}{ds} = f(y^*, \theta, \kappa) \tag{4.2}$$

The differences in the current curvature and orientation is given as

$$\Delta\kappa = \kappa_{odo} - \kappa_{image} \tag{4.3}$$

$$\Delta\theta = \theta_{odo} - \theta_{image} \tag{4.4}$$

where θ_{odo} is Yamabico's current odometry orientation and κ_{odo} is the current odometry instantaneous path curvature. The proposed steering function is

$$\frac{d\kappa}{ds} = -(a\Delta\kappa + b\Delta\theta + cy^*) \tag{4.5}$$

where a, b and c are positive constants. Equation 4.5 is equivalent to

$$\frac{d\kappa}{ds} + a\Delta\kappa + b\Delta\theta + cy^* = 0 \tag{4.6}$$

In order to find the critical damping conditions required for non-oscillatory vehicle motion, a special path is considered. Assume that p_{ref} is equivalent to the positively oriented x-axis of the global Cartesian coordinate system, i.e. $y = 0$. In this case, the image orientation and curvature are always equal to zero. Thus

$$\kappa_{image} = \theta_{image} = 0 \tag{4.7}$$

and

$$y^* = y \tag{4.8}$$

Assume that $-\frac{\pi}{2} < \theta < \frac{\pi}{2}$, then Yamabico's path can be represented as

$$y = y(x) \tag{4.9}$$

then solving for the steering function in terms of y.

$$\theta = \operatorname{atan}(y') = y' - \frac{y'^3}{3} + \frac{y'^5}{5} - \ldots \tag{4.10}$$

$$\kappa = \frac{y''}{(1+y'^2)^{\frac{2}{3}}} \tag{4.11}$$

$$\frac{d\kappa}{ds} = \frac{\frac{d\kappa}{dx}}{\frac{ds}{dx}} = \frac{\frac{d}{dx}\left(\frac{y''}{(1+y'^2)^{\frac{2}{3}}}\right)}{\sqrt{1+y'^2}} = y'''(1+y'^2)^{-2} - 3y'y''^2(1+y'^2)^{-3} \tag{4.12}$$

then assume

$$y'^2 \ll 1 \tag{4.13}$$

and

$$y'y''^2 \ll y''' \tag{4.14}$$

Equation 4.12 becomes the ordinary differential equation

$$y''' + ay'' + by' + cy = 0 \tag{4.15}$$

which when put into differential form is

$$(D^3 + aD^2 + bD + c)\,y = 0 \tag{4.16}$$

Since Equation 4.15 is a third order linear differential equation with constant coefficients, it must have at least one real root. A non-oscillatory decaying solution to Equation 4.15 is desired, therefore there must be three negative roots of D. A critical damping solution of Equation 4.15 must have a triple root, call this root $-k$ where $k > 0$. Thus

$$(D^3 + aD^2 + bD + c)\,y = (D + k)^3 = D^3 + 3kD^2 + 3k^2D + k^3 \tag{4.17}$$

where

$$a = 3k \tag{4.18}$$

$$b = 3k^2 \tag{4.19}$$

$$c = k^3 \tag{4.20}$$

Under these conditions, there is only one degree of freedom in choosing the coefficients a, b and c. Equation 4.15 becomes

$$(D + k)^3 y = 0 \tag{4.21}$$

and its general solution is

$$y = (\frac{A}{2}x^2 + Bx + C)\,e^{-kx} \tag{4.22}$$

where A, B and C are integral constants. A size constant, s_o may be defined as

$$s_o = \frac{1}{k} \qquad 4.23$$

then s_o has the dimension of distance. The size constant determines the distance Yamabico moves along the reference path before it reaches the path. A smaller size constant makes the transition distance smaller, therefore s_o controls the sharpness of Yamabico's trajectory.

Figure 4.1 illustrates the method of path tracking control. Each vehicle control cycle Yamabico reads its wheel encoders and calculates its current odometry configuration q_o. This configuration is geometrically projected onto a current reference path element. The configuration projected onto the reference path is called the image. The signed distance y^* of q_o from the path elements is also determined. Each vehicle control cycle the image and y^* are used to determine Yamabico's instantaneous path curvature κ.

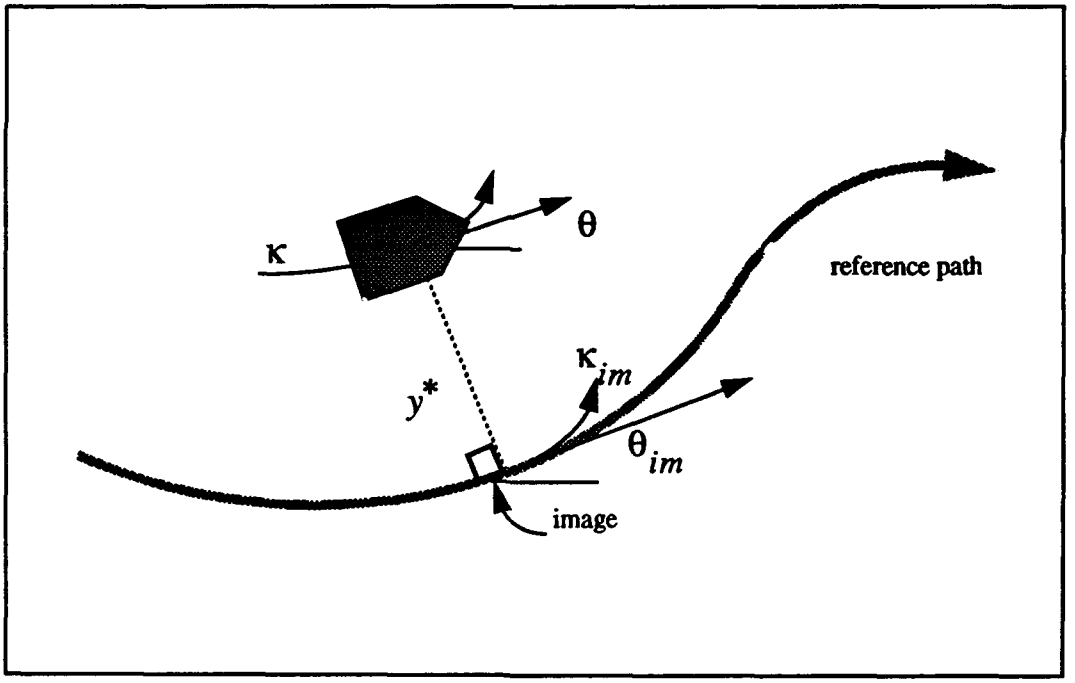


Figure 4.1 Yamabico Path Tracking Control

Figure 4.2 shows Yamabico following a path specified by a single straight line. The path element $p_1 = (x_1, y_1, \theta_1, \kappa_1)$ defines the intended path. The path element p_1 represents a directed half-line. The (x_1, y_1) components of p_1 define the origin of the line, θ_1 gives the orientation with respect to the x-axis and κ_1 represents the path element's curvature. Each vehicle control cycle, Yamabico's control program performs the odometry function by reading its optical wheel encoders. The vehicle's odometry configuration is calculated using the distance traveled by the left and right wheels. This configuration is projected onto p_1 to give the vehicle's image. The tracking algorithm then determines the necessary path curvature and wheel velocity to move the vehicle onto the path element p_1. The size constant s_o determines how rapidly the vehicle converges onto the current path.

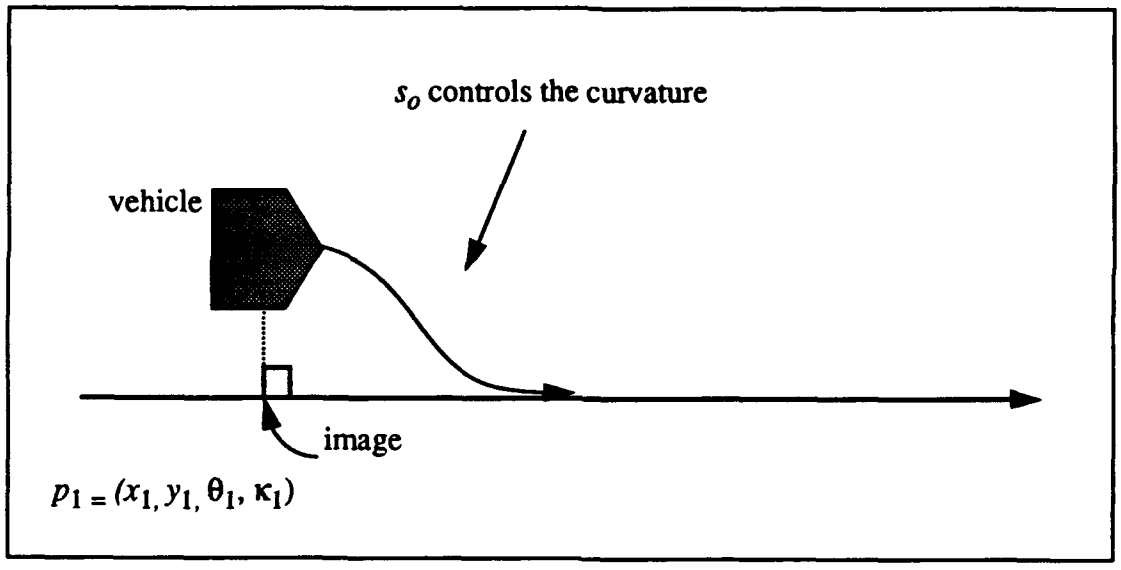


Figure 4.2 Yamabico Tracking a Straight Line

Figure 4.3 shows Yamabico following a path specified by two straight line path elements. These path elements are specified by the configurations p_1 and p_2. Yamabico tracks along initially using p_1 as the reference path. The geometric module computes the point of path intersection while Yamabico is in motion and tracking path p_1. Next the leav-

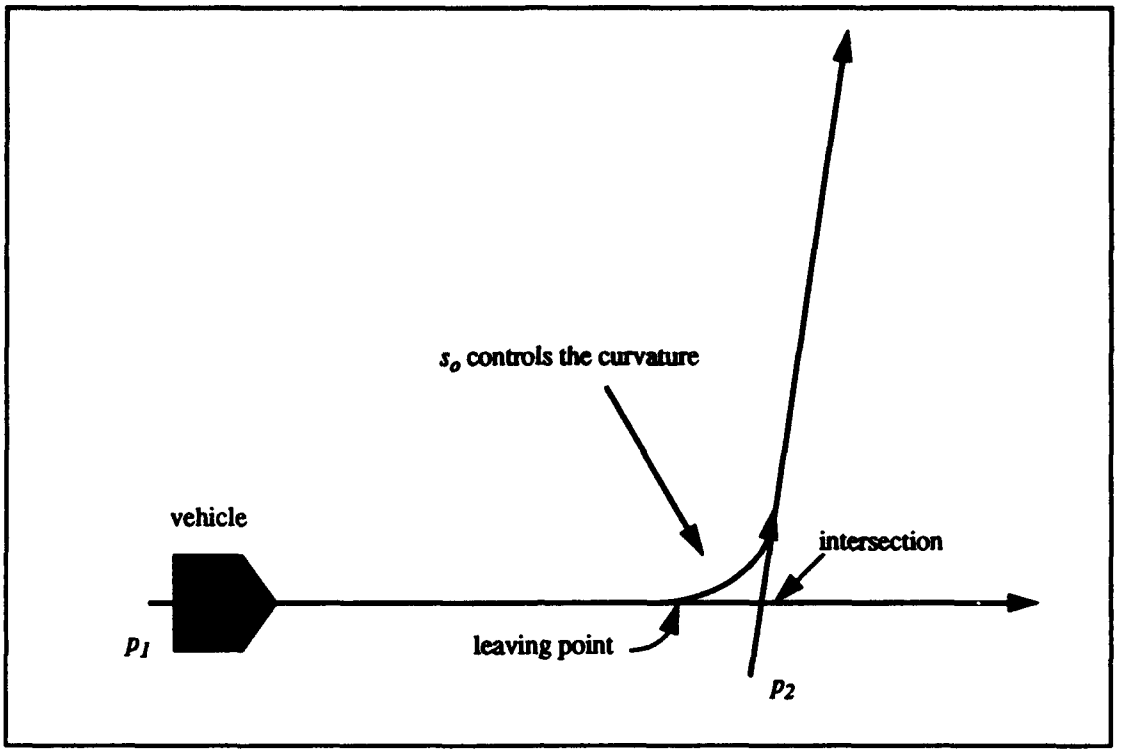


Figure 4.3 Path Tracking Line to Line

ing point on p_1 is determined based upon the intersection and the size constant s_o. This s_o parameter determines the maximum robot path curvature during the transition between p_1 and p_2. Yamabico follows the line p_1 until the image reaches the leaving point. Then it switches to tracking path element p_2. In this manner the vehicle automatically determines the optimum transition point between any two path elements. This technique results in smooth line-to-line path tracking with no overshoot. The motion control subsystem supports all vehicle motion required to explore an orthogonal world space for cartography. Cubic spirals and parabolic path elements have been implemented but are not used for automated cartography algorithm. Vehicle kinematic theory, all MML locomotion functions, and the rules for path element transitions appear in the Yamabico User's Manual provided in Appendix A. Appendix B provides all of the motion control source code.

D. SONAR SUBSYSTEM

The sonar subsystem controls Yamabico's sonar hardware, sonar data processing, and data storage. The sonar subsystem was developed principally by Sherfy and Kanayama [Sherfey 91]. Several improvements were made to support the research described in this dissertation.

1. Hardware Control

Yamabico's sonar hardware is extremely efficient because three dedicated sonar boards control the sonar sensors [Sherfey 91]. Yamabico's main central processing unit is interrupted only when data becomes available from the sonar array. The sonar system provides user interface functions that control Yamabico's array of sonar range finders. At any point within a user's program, any of the 12 sonars may be enabled or disabled. This allows the user to operate a given sonar only when necessary for a particular application. The latest range value for a given sonar may also be provided by a range function. A user's program can also be forced to "busy wait" until some sonar-based condition is satisfied. This feature is particularly valuable for obstacles avoidance. For example, a user's program could be written to wait until the forward looking sonar's range is less than distance d, then stop.

2. Calculation of Global Sonar Return

The global position of the sonar target providing a sonar return to a given sonar may be automatically calculated in real time. This calculation is necessary for linear fitting which provides the input for the edge extraction portion of automated cartography. This portion of the system uses Yamabico's current odometry configuration, the range value returned from a given sonar and the configuration of the sonar sensor with respect to Yamabico's configuration to perform this calculation. A sonar target's location (x_g, y_g) in the global reference frame is calculated by this portion of the module. This feature is illustrated in Figure 4.4 and described in greater detail by Sherfey [Sherfey 91]. Using Yamabico's current odometry configuration estimate q_o, the position of the sonar receiver (x_s, y_s) mounted on Yamabico is calculated by Equations 4.24 and 4.25.

$$x_s = x_o + offset(\sin(phi + \theta_o)) \quad 4.24$$

$$y_s = y_o + offset(\cos(phi + \theta_o)) \quad 4.25$$

The sonar range value d is used to calculate the global location of the sonar target using Equations 4.26 and 4.27.

$$x_g = x_s + d(\cos(axis + \theta_o)) \quad 4.26$$

$$y_g = y_s + d(\sin(axis + \theta_o)) \quad 4.27$$

3. Least Squares Linear Fitting

Linear fitting of global sonar data for a given sonar is performed in order to extract line segments representing sonar reflecting surfaces in Yamabico's world space. The linear fitting algorithm examines each individual global sonar return and determines if it can be fitted to the current line segment. When ten or more global returns fall onto a straight line (with a user's selected tolerance), the linear fitting algorithm builds a line segment for a particular sonar. Linear fitting continues as long as sonar returns fall onto the line segment under construction. Linear fitting is terminated when one global sonar return fails to fall onto the projected line segment being constructed. Line segment data can also be manipulated during line segment construction as well as after the segment has been completed. The line segment data may be manipulated using pointers to the individual line segment data structures. This is an important feature for automated cartography because sonar line segment data must be efficiently manipulated in order to build a partial world from sonar data.

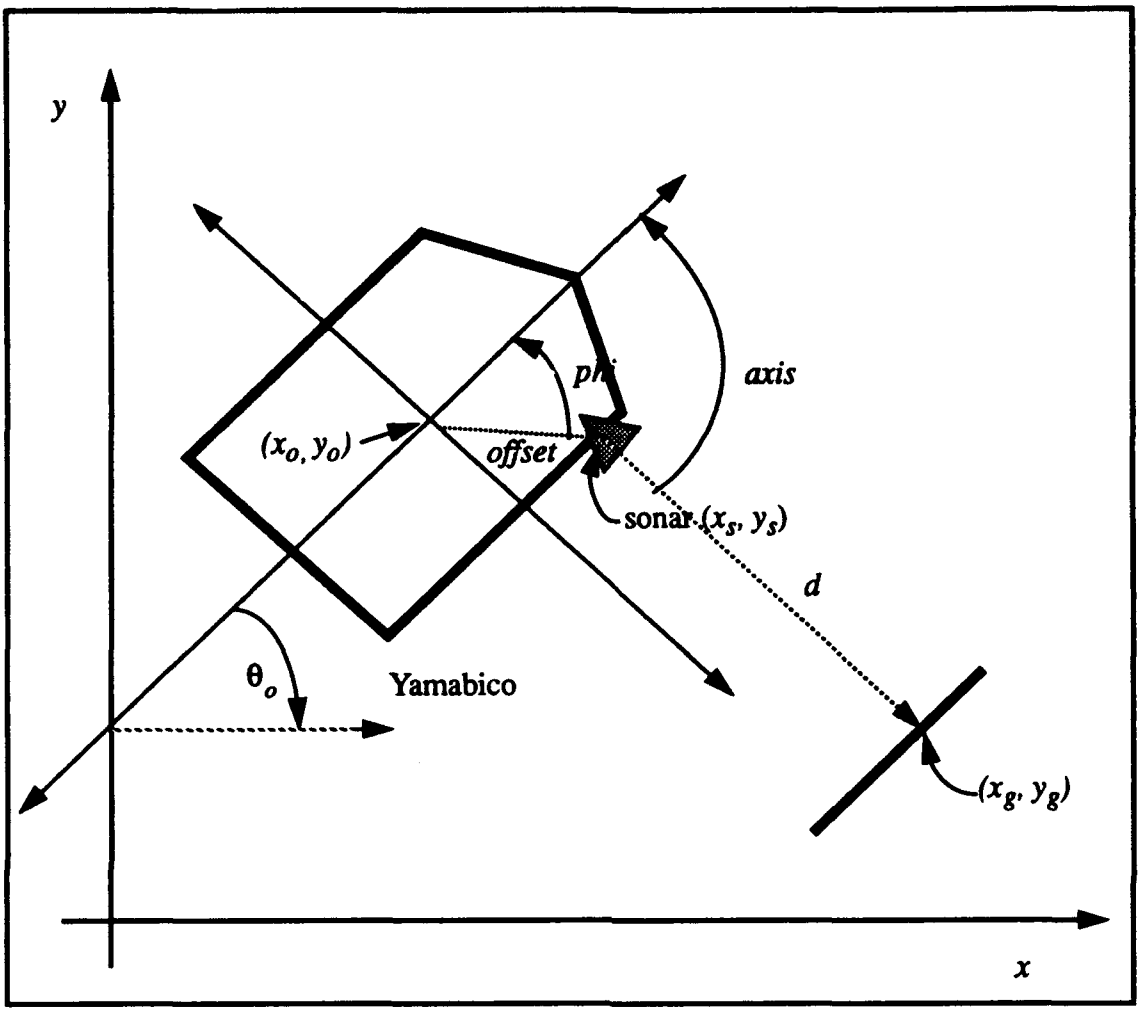


Figure 4.4 Global Sonar Return

Suppose n consecutive valid data points have been collected in a local coordinate system, $(p_1,..., p_n)$, where $p_i = (x_i, y_i)$ for $i = 1,..., n$. The moments m_{jk} of the set of points using are obtained Equation 4.28.

$$m_{jk} = \sum_{i=1}^{n} x^j{}_i y^k{}_i \qquad (0 \le j, k \le 2, \text{ and } j + k \le 2) \qquad 4.28$$

Notice that $m_{00} = n$. The centroid C is given by

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \equiv (\mu_x, \mu_y) \qquad 4.29$$

The secondary moments around the centroid are given by

$$M_{20} \equiv \sum_{i=1}^{n} (x_i - \mu_x)^2 = m_{20} - \left(\frac{m_{10}}{m_{00}}\right)^2 \qquad 4.30$$

$$M_{11} \equiv \sum_{i=1}^{n} (x_i - \mu_x)(y_i - \mu_y) = m_{11} - \left(\frac{m_{10}m_{01}}{m_{00}}\right) \qquad 4.31$$

$$M_{02} \equiv \sum_{i=1}^{n} (y_i - \mu_y)^2 = m_{02} - \left(\frac{m_{01}}{m_{00}}\right)^2 \qquad 4.32$$

The parametric representation (r, α) of a line with constants r and α is adopted. If a point $p = (x, y)$ satisfies Equation 4.33,

$$r = x\cos\alpha + y\sin\alpha \qquad (-\pi/2 < \alpha \leq \pi/2) \qquad 4.33$$

then the point p is on a line L whose normal has an orientation α and whose distance from the origin is r as shown in Figure 4.5. This method has an advantage in expressing lines that are perpendicular to the x-axis. The point-slope method, where $y = mx + b$, is incapable of representing such a case ($m = \infty$, b is undefined).

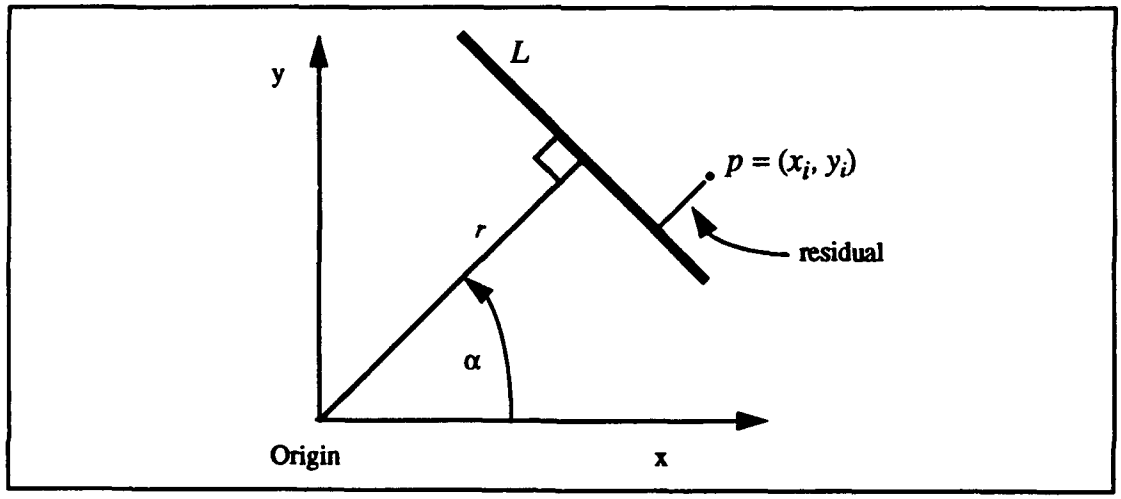


Figure 4.5 Representation of a line L using r and α

The residual of point $p_i = (x_i, y_i)$ and the line $L = (r, \alpha)$ is $x_i\cos\alpha + y_i\sin\alpha - r$. Therefore, the sum of the squares of all residuals is given by Equation 4.34.

$$S = \sum_{i=1}^{n} (r - x_i \cos\alpha - y_i \sin\alpha)^2 \qquad 4.34$$

The line which best fits the set of points is supposed to minimize S. Thus the optimum line (r, α) must satisfy

$$\frac{dS}{dr} = \frac{dS}{d\alpha} = 0 \qquad 4.35$$

Figure 4.6 provides an illustration of Yamabico performing least squares linear fitting of global sonar data. Individual global sonar returns are used to fit a line segment representing the surface providing the sonar returns. At least three global sonar returns are required to start linear fitting. The line segment under construction is geometrically projected forward. Global sonar returns must fall within a certain residual distance of this projected line. If any global return falls outside this zone, linear fitting for the current line segment

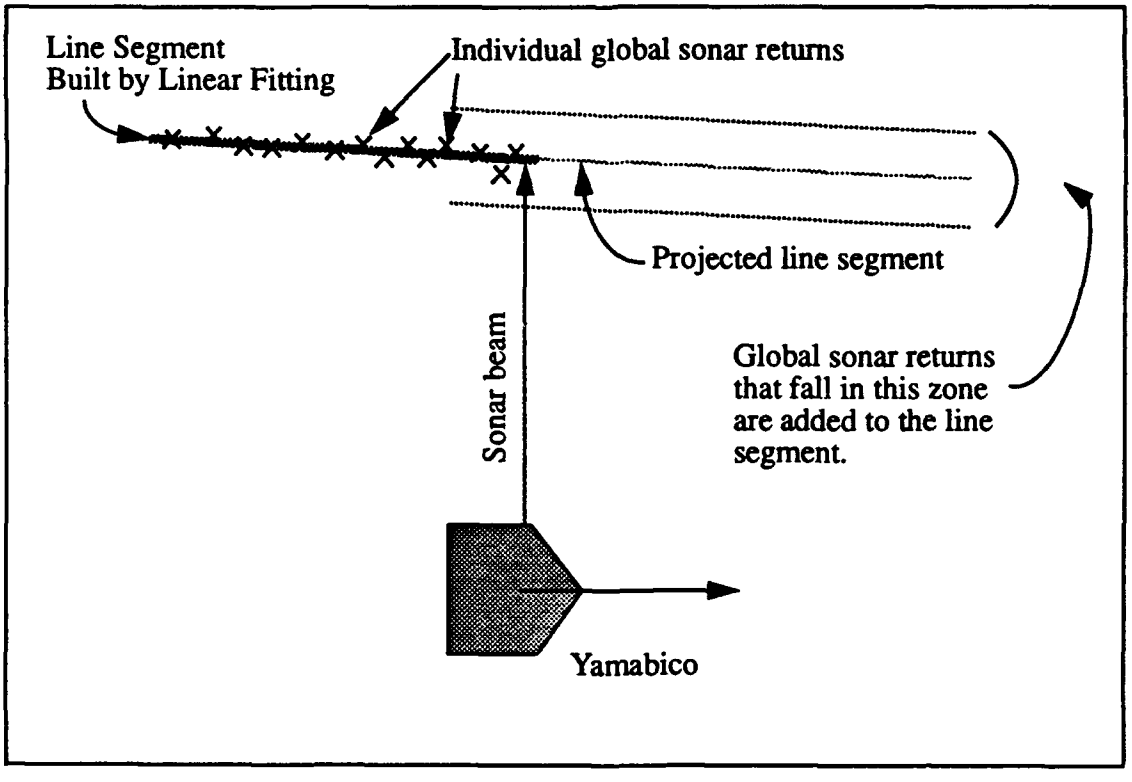


Figure 4.6 Linear Fitting

is terminated and all moments get reset. Figure 4.7 provides an illustration of actual global sonar data fitted to a line segment.

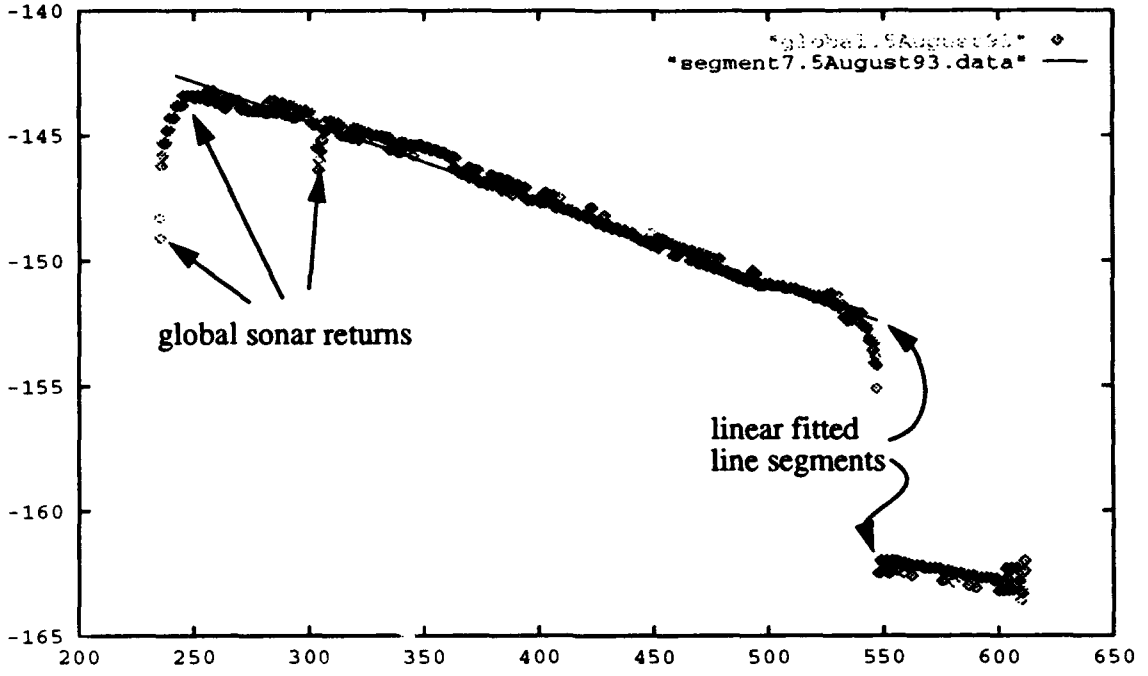


Figure 4.7 Linear Fitting Applied to Global Sonar Data

4. Data Logging

The sonar subsystem also provides facilities for sonar data storage and retrieval. Sonar data may be logged in three forms; raw range data, global target position data, and line segment data. An interval function controls how often sonar data is logged. Raw sonar range data consists of the individual range values for a particular sonar for each sonar ping. Range values are stored as the range from the sonar detector to the target in centimeters. Global sonar data consists of the (x_g, y_g) position of every sonar return received while data logging is enabled. The position of each global return is stored as a pair of floating point numbers representing the x and y locations of the sonar target in Yamabico's global reference frame. The segment data from linear fitting may be logged as a series of line segments extracted by a given sonar. The endpoints, length, and orientation of each line segment are logged. This is a compact method of sonar data storage since an individual line segment can

represent hundreds or thousands of global sonar position values. Logged sonar line segment data is used as input information for some automated cartography functions. All of the 'C' code for the sonar functions is listed in Appendix C. The individual sonar functions for global data position determination are explained in the Yamabico User's Manual in Appendix A.

E. INPUT OUTPUT SUBSYSTEM

Yamabico's input/output subsystem provides three important functions; screen input/output via the on board laptop computer, facilities for downloading executable programs to Yamabico's main memory, and functions for retrieval of sonar data collected by Yamabico. Currently all data transfers are via two 9600 baud RS232 serial ports on Yamabico as described in Chapter III.

1. On Board User Interface

Yamabico is a self-contained autonomous mobile robot. A textual user's interface is provided in the form of a Macintosh™ Powerbook notebook computer installed on board Yamabico. The screen input/output portion of this subsystem provides functions for reading information from and writing information to Yamabico's on board computer. Floating point numbers, text, characters, and integers can be written to the laptop's screen to provide the user with current diagnostic information while Yamabico operating. This feature is an extremely valuable tool for troubleshooting bugs in the robot's code. Similarly, functions are available that read user input from the laptop's keyboard. A "user.c" program can be written that periodically requests keyboard input from a human user. A good example of this is an application that allows a user to choose among several programs loaded in Yamabico's memory.

2. Facilities to Download Executable Programs

Functions that download robot programs from the host computer are also included in this subsystem. Executable robot code on the host computer is downloaded via a 9600

baud RS232 serial line. The entire robot kernel (MML system) or just an user's application program (a compiled "user.c" file) may be transferred to Yamabico's main memory using these functions. Two Unix file transfer programs provide the necessary data transfer protocols. Detailed operating procedures for downloading code onto Yamabico are provided in Appendix A.

3. Retrieval of Data Collected by Yamabico

Several functions provide the user with the ability to transfer data collected by Yamabico back to the host computer for analysis. These functions fall into three categories; location trace data, logged sonar data, and cartography maps. Yamabico's odometry configuration and other guidance parameters may be stored in main memory during robot operations. The input/output subsystem provides a function for transferring this data back to the host computer. In the same fashion, sonar data and cartography data may also be transferred back to the host via a serial link. The input/output subsystem provides the essential link between the mobile robot platform and the researcher. Post mission analysis of the data collected by Yamabico is important for analyzing the success of a mission.

F. SUMMARY

The task scheduling module provides multitasking scheduling for Yamabico software system. System resources are efficiently managed by this scheduler. The geometric module provides geometric spatial reasoning functions for path tracking and low level vehicle motion planning functions. This module supports the motion control subsystem.

Path tracking automatically computes the proper leaving point to facilitate a smooth transition between successive path elements. This frees the user from the tedious task of specifying Yamabico's motion between paths. This also allows Yamabico's software to calculate a path to the goal by a series of abstract path element segments which are easily turned into motion commands. Path tracking reduces odometry reset error due to wheel slippage. This is especially important at higher robot velocities. An odometry reset with re-

spect to a path element is far smoother than an odometry reset with respect to a simple configuration. Yamabico's odometry correction theory appears in Chapter V.

The sonar subsystem provides a user interface for controlling of Yamabico's sensor array. Functions are provided to process and store sonar data in real time. The input/output subsystem provides the essential interface between Yamabico and the user. Functions are provided to send data to and receive data from Yamabico's main memory. This data may be transferred to a Unix host computer or Yamabico's on board laptop computer.

V. THEORETICAL BASIS OF VEHICLE ODOMETRY CORRECTION

The vehicle odometry estimate correction method used to support robot cartography is explained in this chapter. This discussion is preceded by an introduction to several related odometry correction methods and a discussion of fundamental concepts relevant to this work. This algebra provides a useful abstraction since the vehicle's configuration, odometry error, and vehicle landmarks can now be manipulated by simple algebraic equations.

The 3D homogeneous transformation groups and quaternions are widely used in analysis and design of robot manipulators [Paul 84] [Lozano-Perez 83] [Fu 87] [Rolfe 86]. Likewise, a 2D transformation group to represent positioning of rigid body vehicles placed in a plane is needed. However, the formulation given here is not merely the 2D version of existing transformation groups. The expression of a robot's orientation θ is designed to explicitly maintain complete vehicle orientation information beyond the range of $[-\pi,\pi]$. This allows the vehicle's orientation to store the vehicle's motion history. This formulation has the same advantage as a 3D homogeneous transformation, i.e. translation and rotation are described in a single mathematical structure, the configuration. This algebraic system is a variation of the 3D homogeneous transformation group. However, the system does not have a point of singularity, which was one of the drawbacks of the homogeneous transformation.

This chapter introduces a configuration algebra based upon group theory. It is used to calculate Yamabico's position and motion in the 2D plane. This algebra provides the required coordinate transformation calculations for dead reckoning error detection and correction. This computationally efficient method allows odometry error determination and correction in real-time. 2D planar transformations are not a new concept, but this technique makes odometry corrections more amenable to human understanding. Therefore, it is easier to write and debug computer code to perform these transformations. Group theory provides a well known algebraic framework for 2D transformation calculations [Bloch 87].

Odometry correction is performed using only three elementary components.

(1) q^{-1} - configuration inverse (the mathematical inverse of a given configuration),

(2) $e = (0, 0, 0)^T$ - the identity configuration, and

(3) the composition function (a function for combining two configurations).

Group theory provides a simple abstract algebra that makes calculus related to vehicle motion design and control transparent and easy, including the analysis of dead reckoning errors. This algebraic system is implemented in the high level mobile robot language, MML for the autonomous mobile robot Yamabico-11 [Kanayama 88] [Kanayama 91a]. All the definitions and the basic functions (composition, inverse, symmetric property, and so forth) provide a powerful and simple user interface.

A. THE TRANSFORMATION GROUP

This section introduces the 2D coordinate transformation algebra. Let $\Re$ denote the set of all real numbers. A *transformation* q is

$$q \equiv \begin{bmatrix} x \\ y \\ \theta \end{bmatrix} \qquad \text{where } x, y, \theta \in \Re. \qquad 5.1$$

The set of all transformations is denoted by T $(= \Re^3)$. For instance, $q_1 = (2, 1, \pi/6)^T$ and $q_2 = (2, 4, \pi/4)^T$ are examples of transformations (M^T means the transposition of a matrix M). Obviously, a transformation q is interpreted as a 2D coordinate transformation from the global Cartesian coordinate system F_O to another coordinate system F as shown in Figure 5.1.

Let $q_1 = (x_1, y_1, \theta_1)^T$ and $q_2 = (x_2, y_2, \theta_2)^T$. The *composition* $q_1 \circ q_2$ of these two configurations is defined in Equation 5.2.

$$q_1 \circ q_2 \equiv \begin{bmatrix} x_1 \\ y_1 \\ \theta_1 \end{bmatrix} \circ \begin{bmatrix} x_2 \\ y_2 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 \\ \theta_1 + \theta_2 \end{bmatrix} \qquad 5.2$$

The equation $q_1 = q_2$ is true if and only if $x_1 = x_2$, $y_1 = y_2$ and there exists an integer n such that $\theta_1 = \theta_2 + 2n\pi$. The interpretation of $q_1 \circ q_2$ in the domain of 2D coordinate transformation is the composition of the coordinate transformations q_1 and q_2. The following is one of immediate results from the definition above.

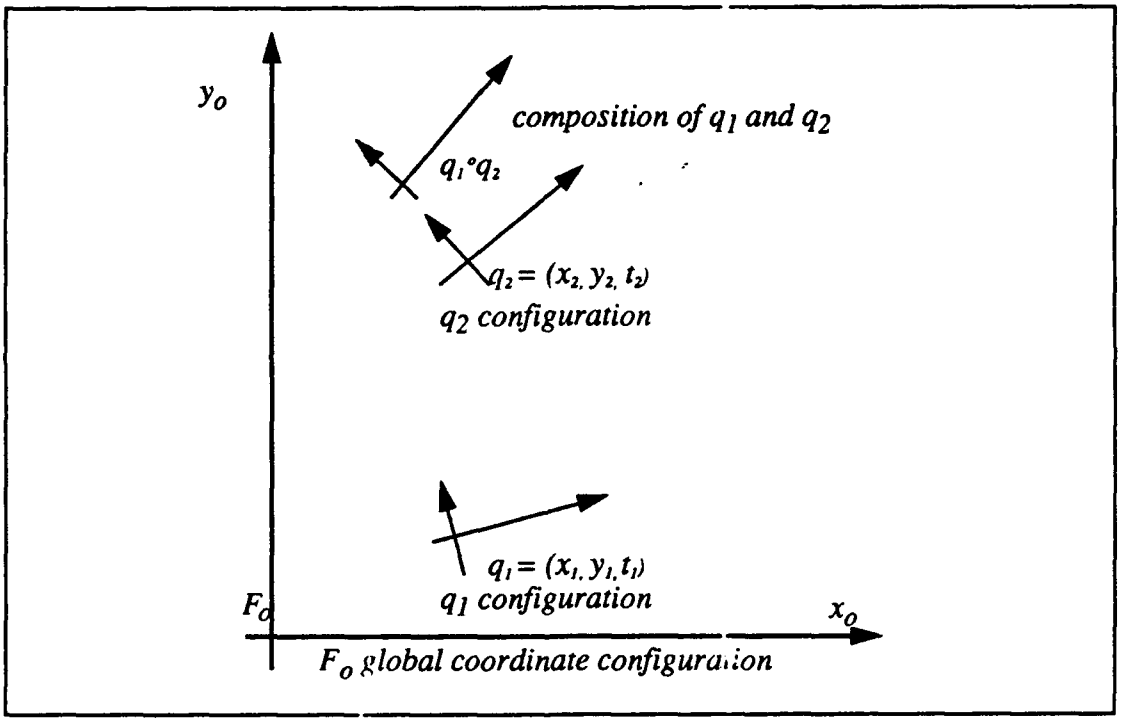


Figure 5.1 Composition

Corollary 5.1 For any $q = (x,y,\theta)^T \in \mathbf{T}$,

$$\begin{bmatrix} x \\ y \\ \theta \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \end{bmatrix} \circ \begin{bmatrix} 0 \\ 0 \\ \theta \end{bmatrix} \tag{5.3}$$

Therefore, a transformation $(x, y, \theta)^T$ can always be decomposed into a *translation* $(x, y, 0)^T$ and a *rotation* $(0, 0, \theta)^T$. Notice, however, that in general $q \neq (0, 0, \theta)^T \circ (x, y, 0)^T$, since the composition function is not commutative. The composition of the example transformations stated above is given by Equation 5.4.

$$\begin{bmatrix} 2 \\ 1 \\ \frac{\pi}{6} \end{bmatrix} \circ \begin{bmatrix} 2 \\ 4 \\ \frac{\pi}{4} \end{bmatrix} = \begin{bmatrix} \sqrt{3} \\ 2+2\sqrt{3} \\ \frac{5\pi}{12} \end{bmatrix} \tag{5.4}$$

In order to use the compose function for transformation algebra, it is necessary to characterize its properties in terms of familiar algebraic properties. In order to use the compose function in the transformation space $\mathbf{T}$, it is important to prove that the set of all transformations form a group with respect to the compose function. The following lemmas are required to prove this fact.

LEMMA 5.1(**Closure Property**) *For any* $q_1, q_2 \in \mathbf{T}$,

$$q_1 \circ q_2 \in \mathbf{T} \tag{5.5}$$

Proof: Each component of $q_1 \circ q_2$ in Equation 5.2 is a real number. □

The closure property means that the composition of any two configurations gives a valid configuration in the transformation space $\mathbf{T}$.

LEMMA 5.2(**Associative Property**) *For any* $q_1, q_2, q_3 \in \mathbf{T}$,

$$(q_1 \circ q_2) \circ q_3 = q_1 \circ (q_2 \circ q_3) \tag{5.6}$$

Proof:

$$(q_1 \circ q_2) \circ q_3 = \begin{bmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 \\ \theta_1 + \theta_2 \end{bmatrix} \begin{bmatrix} x_3 \\ y_3 \\ \theta_3 \end{bmatrix} \tag{5.7}$$

$$= \begin{bmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 + x_3\cos(\theta_1+\theta_2) - y_3\sin(\theta_1+\theta_2) \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 + x_3\cos(\theta_1+\theta_2) + y_3\sin(\theta_1+\theta_2) \\ \theta_1 + \theta_2 + \theta_3 \end{bmatrix} \tag{5.8}$$

$$= \begin{bmatrix} x_1 + (x_2 + x_3\cos\theta_2 - y_3\sin\theta_2)\cos\theta_1 - (y_2 + x_3\sin\theta_2 + y_3\cos\theta_2)\sin\theta_1 \\ y_1 + (x_2 + x_3\cos\theta_2 - y_3\sin\theta_2)\sin\theta_1 + (y_2 + x_3\sin\theta_2 + y_3\cos\theta_2)\cos\theta_1 \\ \theta_1 + (\theta_2 + \theta_3) \end{bmatrix} \tag{5.9}$$

$$\begin{bmatrix} x_1 \\ y_1 \\ \theta_1 \end{bmatrix} \begin{bmatrix} x_2 + x_3\cos\theta_2 - y_3\sin\theta_2 \\ y_2 + x_3\sin\theta_2 + y_3\cos\theta_2 \\ \theta_2 + \theta_3 \end{bmatrix} = q_1 \circ (q_2 \circ q_3) \tag{5.10}$$

□

LEMMA 5.3(**Identity**) *For all* $q \in T$,

$$q^{\circ}\begin{bmatrix}0\\0\\0\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix}^{\circ}q = q \tag{5.11}$$

Proof. If $q = (x, y, \theta)^T$,

$$\begin{bmatrix}x\\y\\\theta\end{bmatrix}\begin{bmatrix}0\\0\\0\end{bmatrix} = \begin{bmatrix}x\\y\\\theta\end{bmatrix} = \begin{bmatrix}0\\0\\0\end{bmatrix}\begin{bmatrix}x\\y\\\theta\end{bmatrix} \tag{5.12}$$

Therefore, $(0, 0, 0)^T \equiv e$ *is the unique identity element in T.* □

The following Lemma demonstrates the presence of the left and right inverse transformations for each $q \in \mathbf{T}$.

LEMMA 5.4(**Right and Left Inverse**) Let $q \equiv (x, y, \theta)^T$ be given. The left and right inverse of q are given by the following equations.

(1) The solution to an equation $q_{left}{}^{\circ}q = e$ is

$$q_{left} \equiv \begin{bmatrix}-x\cos\theta - y\sin\theta\\ x\sin\theta - y\cos\theta\\ -\theta\end{bmatrix} \tag{5.13}$$

(2) The solution to an equation $q^{\circ}q_{right} = e$ is

$$q_{right} \equiv \begin{bmatrix} -x\cos\theta - y\sin\theta \\ x\sin\theta - y\cos\theta \\ -\theta \end{bmatrix} \tag{5.14}$$

Proof. (1) Equation $q_{left} \circ q = e$ becomes,

$$x_{left} + x\cos\theta_{left} - y\sin\theta_{left} = 0 \tag{5.15}$$

$$y_{left} + x\sin\theta_{left} - y\cos\theta_{left} = 0 \tag{5.16}$$

$$\theta_{left} + \theta = 0 \tag{5.17}$$

By Equation 5.17, $\theta_{left} = -\theta$. By substituting in θ_{left} Equations 5.15 and 5.16, Equation 5.13 is obtained.

(2) Equation $q \circ q_{right} = e$ becomes,

$$x + x_{right}\cos\theta - y_{right}\sin\theta = 0 \tag{5.18}$$

$$y + x_{right}\sin\theta - y_{right}\cos\theta = 0 \tag{5.19}$$

$$\theta + \theta_{right} = 0 \tag{5.20}$$

Simultaneous solution of Equations 5.15 through 5.17 yield $x_{left} = x_{right}$, $y_{left} = y_{right}$, and $\theta_{left} = \theta_{right}$. Since the left inverse q_1 and right inverse q_2 are equal, there exists a unique inverse q^{-1} for each $q \in \mathbf{T}$. □

LEMMA 5.5(**Existence of a Unique Inverse**) For any transformation $q \equiv (x, y, \theta)^T$, there exists a unique *inverse*

$$q^{-1} \equiv \begin{bmatrix} -x\cos\theta - y\sin\theta \\ x\sin\theta - y\cos\theta \\ -\theta \end{bmatrix} \tag{5.21}$$

Proof. The configurations q_1 and q_2 in Equations 5.13 and 5.14 each represent the inverse of q and they are equivalent. □

As an example, the inverse of a transformation $(4, 2, \pi/6)^T$ is

$$\begin{bmatrix} 4 \\ 2 \\ \frac{\pi}{6} \end{bmatrix}^{-1} = \begin{bmatrix} -4\cos\frac{\pi}{6} - 2\sin\frac{\pi}{6} \\ 4\sin\frac{\pi}{6} - 2\cos\frac{\pi}{6} \\ -\frac{\pi}{6} \end{bmatrix} = \begin{bmatrix} -1 - 2\sqrt{3} \\ 2 - \sqrt{3} \\ -\frac{\pi}{6} \end{bmatrix} \approx \begin{bmatrix} -4.464 \\ 0.268 \\ -\frac{\pi}{6} \end{bmatrix} \tag{5.22}$$

Theorem 5.1 (Transformation Group) - The set $\mathbf{T}$ of transformation is a group with respect to the composition operation ($\circ$), denoted by $<\mathbf{T}, \circ>$.

Proof. The algebraic structure $< \mathbf{T}, \circ >$ satisfies the closure property by LEMMA 5.1, the associative law by LEMMA 5.2, the existence of the identity by LEMMA 5.3, and the existence of an inverse by LEMMA 5.4. Therefore $< \mathbf{T}, \circ >$ is a group. □

Group theory together with its associated properties provides a well-defined algebraic structure for 2D coordinate transformation calculations. The closure property gives an assurance that the results of the composition operation gives an answer that lies with the 2D transformation space. The inverse property provides the ability to undo any algebraic operation. The inverse property simplifies dead reckoning error analysis.

B. FUNDAMENTAL CONCEPTS

A vehicle is placed in a 2D plane $\Re^2$ with a global Cartesian coordinate system F_0. The vehicle has a body-fixed Cartesian coordinate system F_v. The x-axis of F_v points out of the

front of the vehicle and the y-axis of F_v points out of the left side of the vehicle. Since the vehicle moves, F_v is a function of time. The vehicle's position in this plane is described by a *configuration* $q = (x, y, \theta)^T$, where (x, y) is the position of the origin of F_v and θ the orientation of F_v in the global coordinate system [Lozano-Perez 79]. A vehicle configuration $(x, y, \theta)^T$ can be interpreted as a transformation $(x, y, \theta)^T$ which transforms F_0 into F_v. In other words, the vehicle's configuration is a transformation from the global coordinate system F_0 to the vehicle's local coordinate system F_v. Under this interpretation, group theory can be used for the control and analysis of vehicle motion. There is a method for describing a vehicle's motion by a sequence of configurations [Kanayama 91a]. A set of configurations is selected so that any path segment which is obtained by the smooth path planner passes through the constraints specified [Kanayama 88]. For instance, a path shown in Figure 5.2 is described by a sequence of eight configurations, which are also shown in the Figure 5.2.

$$(q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7) = \begin{bmatrix} \begin{bmatrix}0\\0\\0\end{bmatrix} & \begin{bmatrix}2\\0\\0\end{bmatrix} & \begin{bmatrix}3\\1\\\pi/2\end{bmatrix} & \begin{bmatrix}3\\3\\\pi/2\end{bmatrix} & \begin{bmatrix}4\\5\\\pi/2\end{bmatrix} & \begin{bmatrix}4\\7\\\pi\end{bmatrix} & \begin{bmatrix}1\\7\\\pi\end{bmatrix} & \begin{bmatrix}1\\9\\0\end{bmatrix} \end{bmatrix} \qquad 5.23$$

These configurations are constraints to the path and should be selected so that each path segment obtained by the smooth path planner will not have a conflict with the environment [Kanayama 88]. In order to specify the next configuration relative to the current vehicle configuration, it may be easier calculate a relative configuration rather than a global configuration. For instance, when the current vehicle's configuration is q_o, a relative configuration r is given so that the next configuration q_1 is given by the composition operation:

$$q_1 = q_o \circ r \qquad 5.24$$

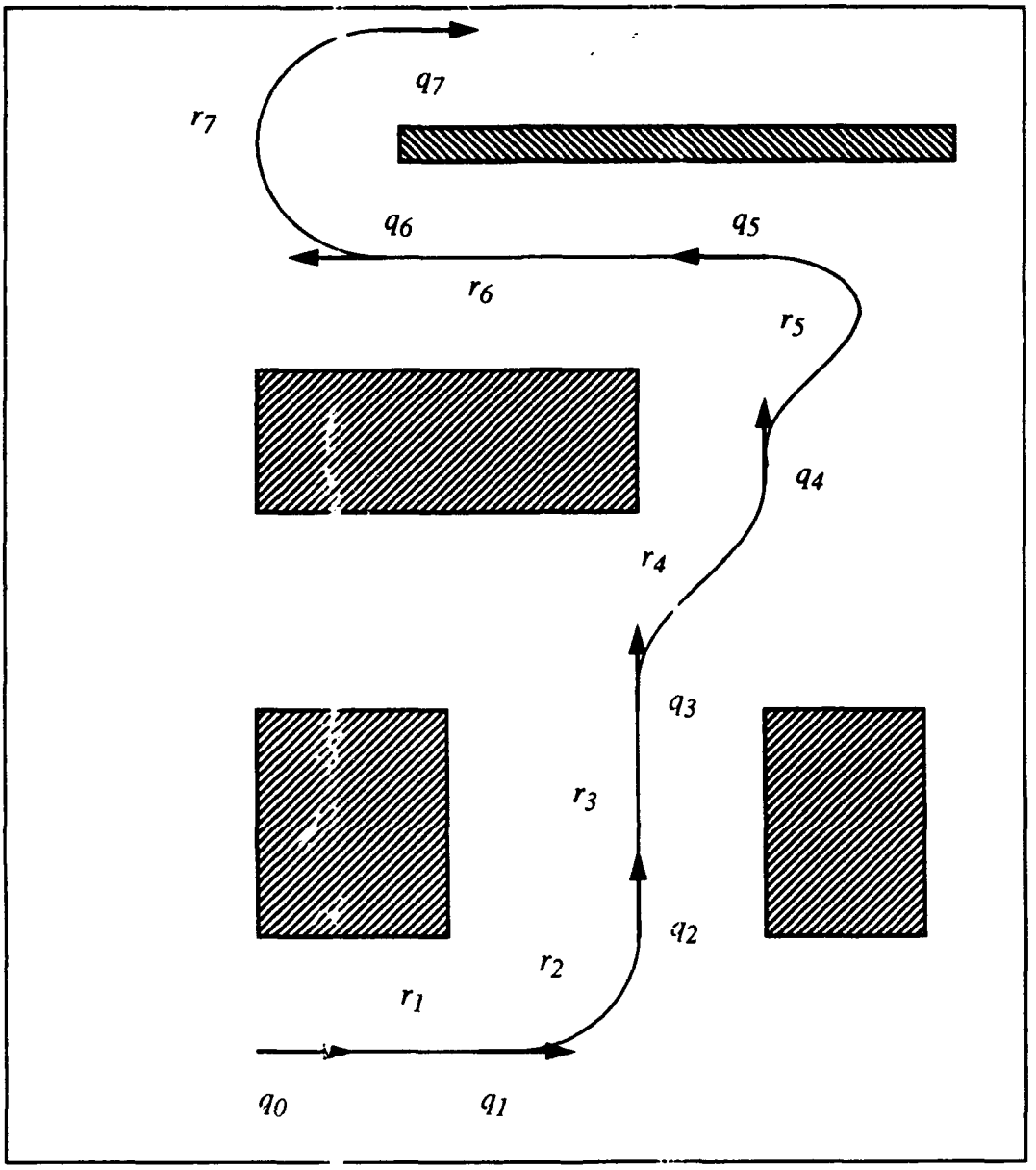


Figure 5.2 Smooth Path Generated by Configurations

In the previous example, each (absolute) configuration q_i is calculated by Equation 5.25,

$$q_i = q_{i-1} \circ r_i \qquad 5.25$$

using a sequence of relative configurations:

$$(r_1, r_2, r_3, r_4, r_5, r_6, r_7) = \left[\begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ \frac{\pi}{2} \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix} \begin{bmatrix} 2 \\ 0 \\ \frac{\pi}{2} \end{bmatrix} \begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ -2 \\ \pi \end{bmatrix} \right] \qquad 5.26$$

These relative configurations are also shown in Figure 5.2.

C. EVALUATION OF ROBOT ODOMETRY ERROR

One unavoidable problem in controlling autonomous mobile robots is the accumulation of dead reckoning errors over time. If dead reckoning error becomes excessive, the vehicle may become lost. The vehicle configuration q_o estimated by the on board odometry function is called the *odometry configuration*. Consider a situation in which the vehicle's odometry configuration is q_o and its actual configuration is q_a. If there is no odometry error, $q_o = q_a$. Otherwise, there is a difference between the vehicle's dead reckoning estimate and its actual configuration as shown in Figure 5.3. An *error configuration* ε is defined such that

$$\varepsilon \circ q_o = q_a \qquad 5.27$$

This relationship is illustrated in Figure 5.3. In this figure the error coordinate system is displaced from the global coordinate system by ε. By the same token, the vehicle's odometry configuration is displaced from its actual configuration by ε^{-1}. That is, this vehicle's best estimate of its configuration q_o is correct only in the "erroneous frame" ε. When there is no dead reckoning error, the error configuration ε is equal to the identity e. If q_o and q_a are known, the error configuration ε can be calculated by Equation 5.28.

$$\varepsilon = q_a \circ q_o^{-1} \tag{5.28}$$

For instance, if $q_o=(2,1,0)^T$ and $q_a=(4,3,\pi/6)^T$,

$$\varepsilon = q_a q_o^{-1} = \begin{bmatrix} 4 \\ 3 \\ \frac{\pi}{6} \end{bmatrix} \circ \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix}^{-1} = (2.768,\ 1.134,\ \frac{\pi}{6})^T \tag{5.29}$$

i.e., the vehicle's odometry configuration $q_o=(2,1,0)^T$ is correct if it is interpreted as a local configuration in ε. The compose operator is omitted from this point until the end of this chapter such that $q_1 \circ q_2 \equiv q_1 q_2$.

An analysis of an error that is caused by a sequence of vehicle motion follows: a vehicle has traveled through a series of two configurations q_{o1} and q_{o2} in this order. These are estimated by odometry. Let q_o be their composition as given in Equation 5.30. Also, let the vehicle's actual movement as measured by an outside observer be q_{a1} and q_{a2}, where q_a is their composition.

$$q_o = q_{o1} q_{o2} \tag{5.30}$$

$$q_a = q_{a1} q_{a2} \tag{5.31}$$

Then, by substituting into Equations 5.27, the equations for the errors ε, ε_1, and ε_2 are determined by the following equations.

$$q_{a1} = \varepsilon_1 q_{o1} \tag{5.32}$$

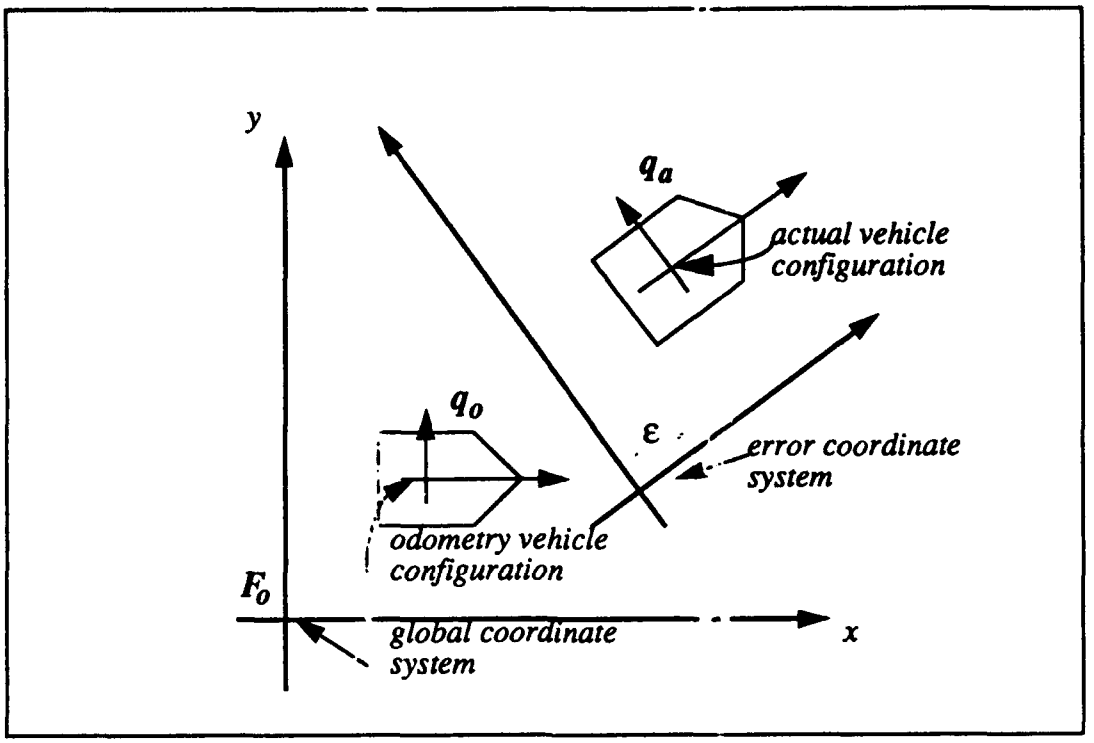


Figure 5.3 Odometry Error Analysis

$$q_{a2} = \varepsilon_2 q_{o2} \tag{5.33}$$

$$q_a = \varepsilon q_o \tag{5.34}$$

Therefore, if each odometry motion estimate and error are known, the total error is

$$\varepsilon = q_a q_o^{-1} \cdots (q_{a1} q_{a2}) (q_{o1} q_{o2})^{-1} \tag{5.35}$$

$$\varepsilon = \varepsilon_1 q_{o1} \varepsilon_2 q_{o2} q_{o2}^{-1} q_1^{-1} = \varepsilon_1 q_{o1} \varepsilon_2 q_{o1}^{-1} \tag{5.36}$$

The involvement of q_1 in this equation makes the error analysis complicated. Similarly, the total error equation for n consecutive motions is as follows:

$$\varepsilon = \varepsilon_1 q_{o1} \cdots \varepsilon_{n-1} q_{on-1} \varepsilon_n q_{on} q_{on}^{-1} q_{o,n-1}^{-1} \cdots q_{o1}^{-1} \qquad 5.37$$

$$\varepsilon = \varepsilon_1 q_{o1} \cdots \varepsilon_{n-1} q_{o,n-1} \varepsilon_n q_{o,n-1}^{-1} \cdots q_{o1}^{-1} \qquad 5.38$$

Assume a special case in which $q_{o1} = \ldots = q_{o,n-1} = e$; i.e., after each component motion q_{oi}, the vehicle is commanded to come back to the initial configuration. In this case, the error configuration becomes simply the composition of all the individual errors as shown in Equation 5.39:

$$\varepsilon = \varepsilon_1 \varepsilon_2 \cdots \varepsilon_n \qquad 5.39$$

This feature is particularly useful for a robot application that requires repeated motion though the same configuration. This allows the robot's dead reckoning error for one circuit of motion to be corrected when the robot returns to the starting point.

D. MODEL-SENSOR-BASED ERROR DETECTION

An algebraic configuration is useful for describing the position of a vehicle. A configuration is also useful to describe the position of any object in the environment. For instance, Object A in Figure 5.4 may be assigned a body-fixed, local coordinate system F_A and its position in this world is described using this local frame. Furthermore, consider a situation in which an ideal sensor mounted on the vehicle senses the configuration of an object in the

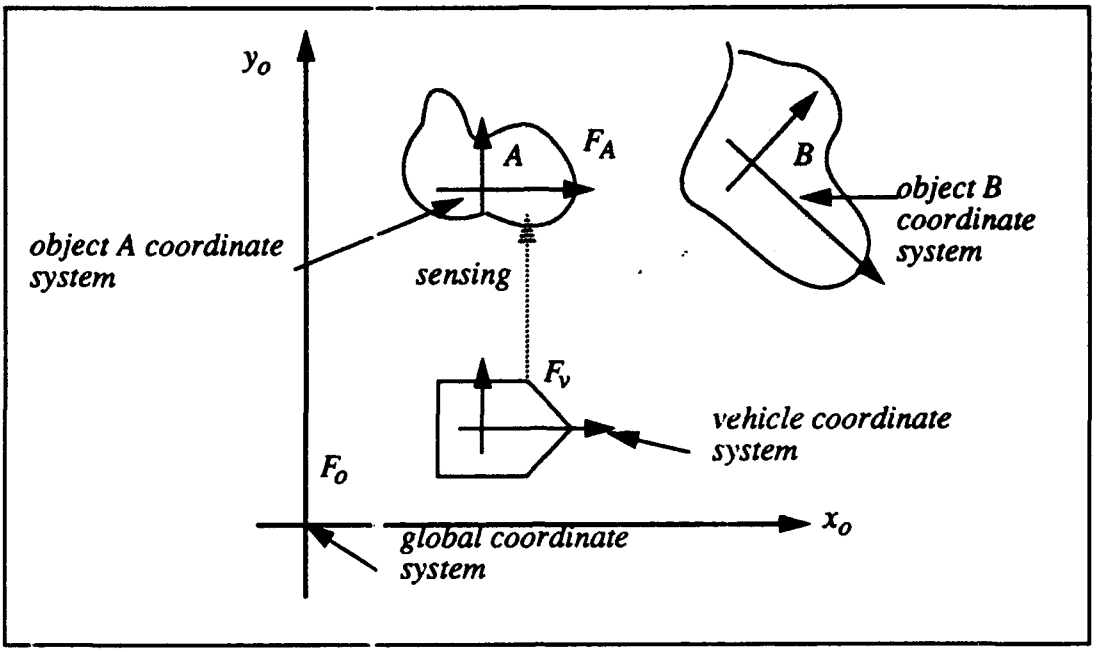


Figure 5.4 Object Configuration

environment. That is, the vehicle is able to sense the relative configuration of an object with respect to its own odometry configuration q_o with complete precision. Therefore, the vehicle's odometry error is effectively super-imposed upon the sensed object configuration.

A method for determining vehicle odometry error by using an external landmark as a point of reference is required for odometry correction. In Figure 5.5, q_a is the vehicle's actual configuration, which is unknown, and p_a is the actual configuration of an object A in the environment, which is obtained from an environmental model. The odometry configuration q_o is known, but contains an error ε. The configuration p_o is the observed configuration of the object A, and may have some error, because this observation is made by the ideal sensor on board using the odometry configuration q_o as a point of reference. As discussed in Section B, a possible difference between p_a and p_o is due to the error ε in odometry.

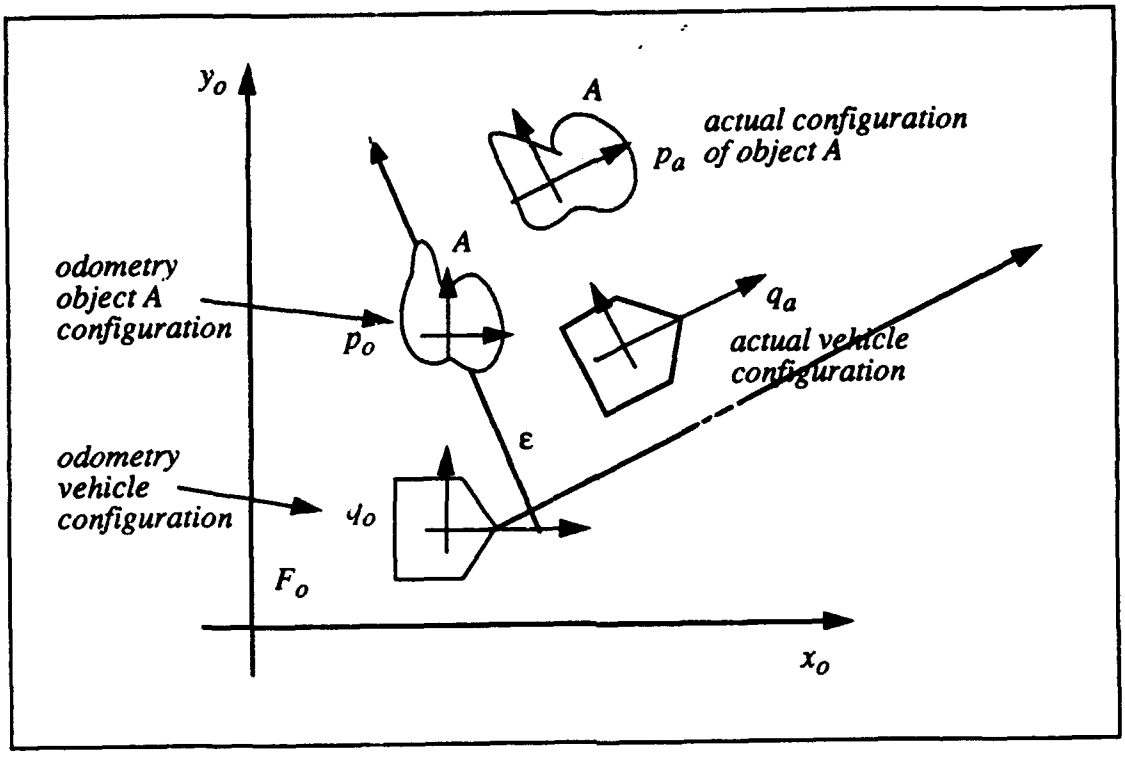


Figure 5.5 Vehicle Odometry Error Detection

The vehicle actually senses the object A at p_a, but since the odometry estimate is at q_o, it believes A is at p_o. The error configuration ε, is introduced as the algebraic difference between both p_o and p_a and between q_o and q_a as shown in Equations 5.40 and 5.41.

$$\varepsilon q_o = q_a \tag{5.40}$$

$$\varepsilon p_o = p_a \tag{5.41}$$

From Equation 5.41, the error configuration is

$$\varepsilon = p_a p_o^{-1}. \qquad 5.42$$

Since both p_a and p_o are known, ε also becomes known. Therefore, by substituting the right hand side of Equation 5.42 for ε in Equation 5.40, the vehicle's actual configuration may be calculated algebraically from known values as shown in Equation 5.43.

$$q_a = \varepsilon q_o = p_a p_o^{-1} q_o \qquad 5.43$$

Since p_a is obtained by the model and p_o is obtained by the sensor, Equation 5.43 formalizes the principle of the model/sensor based odometry error detection. This principle is applied in Chapters VIII for localization during automated cartography. The results obtained from odometry estimate correction experiments appear in Chapter IX.

The use of algebraic constructs for representation of a robot's configuration, a sensed objects configuration, and a robot's dead reckoning error significantly simplifies 2D motion analysis. This algebra is important since three simple constructs provide all of the necessary tools for dead reckoning error detection from sensor data. The expected configuration of an object and the configuration of the object calculated from sensor input are used to allow the vehicle to rapidly calculate its dead reckoning error. This error is applied to the robot's current dead reckoning configuration and the error is corrected.

E. RELATIONSHIP TO OTHER TRANSFORMATION GROUPS

The use of the three-dimensional homogeneous transformation group is common in the robotics field [Paul 84]. The general form of a homogeneous transformation is shown in Equation 5.44.

$$T_3 \equiv \begin{bmatrix} R_{11} & R_{12} & R_{13} & x \\ R_{21} & R_{22} & R_{23} & y \\ R_{31} & R_{32} & R_{33} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.44}$$

where the left-top 3×3 submatrix represents rotation and the right-top 3×1 matrix represents a translation. Its two-dimensional version is

$$T_2 = \begin{bmatrix} R_{11} & R_{12} & x \\ R_{21} & R_{22} & y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & x \\ \sin\theta & \cos\theta & y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.45}$$

This transformation matrix T_2 may be used to represent vehicle's configuration as described in Section C. In order to obtain θ itself from T_2, Equation 5.46 is used.

$$\theta = \text{atan2}\,(R_{21}, R_{11}) \tag{5.46}$$

where the range of the function *atan2* is assumed $[-\pi, \pi]$. However, this method has a drawback. If the vehicle's accumulated rotation is beyond the range of $[-\pi, \pi]$, a part of the vehicle's orientation information is lost. For instance, if the vehicle rotates 2π counterclockwise, θ becomes 0 instead of 2π if Equation 5.46 is used. In order to avoid this information loss, an explicit θ term should be added to T_2 obtaining T_3 as shown in Equation 5.47.

$$T_3 \equiv \begin{bmatrix} \cos\theta & -\sin\theta & x & 0 \\ \sin\theta & \cos\theta & y & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.47}$$

The set T' of all 4×4 matrices of the form in Equation 5.47 under matrix multiplication is a subgroup of the group of all invertible 4×4 matrices. The symbol ($\times$) means matrix multiplication. Therefore $< T', \times >$ is a group.

Groups may be modified but not really but not really changed. When two groups look different but are essentially the same mathematically they are said to be isomorphic. The concept of isomorphism is defined as follows:

Definition: Let <G, #> and <H, $> be groups. G is isomorphic to H provided that there exists a function $\varnothing$: G $\rightarrow$ H such that

1. $\varnothing$ is 1-1.
2. $\varnothing$ is onto H.
3. $\varnothing$ preserves the operation; that is $(a \# b) \varnothing = (a\varnothing) \$ (b\varnothing)$ for all $a, b \in$ G.

Proposition 5.1 $< T, \circ >$ is isomorphic to $< T', \times >$, the subgroup of the multiplicative group of 4×4 invertible matrices consisting of all matrices of the form $\begin{bmatrix} 1 & 0 & 0 & 0 \\ x & \cos\theta & -\sin\theta & 0 \\ y & \sin\theta & \cos\theta & 0 \\ \theta & 0 & 0 & 1 \end{bmatrix}$.

Proof: The transformation group $< T, \circ >$ is isomorphic to a subset of the 4×4 invertible matrices under standard matrix multiplication. Let $f(q)$ be a function that maps each $q=(x, y, \theta)^T \in < T, \circ >$ to the matrix in Equation 5.48.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ x & \cos\theta & -\sin\theta & 0 \\ y & \sin\theta & \cos\theta & 0 \\ \theta & 0 & 0 & 1 \end{bmatrix} \qquad 5.48$$

Clearly, f(q) is a 1-1 function that maps onto every 4×4 matrix of the form above where $x, y, \theta \in$ R. In addition, $f(q_1 \circ q_2) = f(q_1) \times f(q_2)$, where is $\times$ the usual matrix

multiplication. Therefore f preserves the operation and is a homomorphism. Since f is 1-1 and onto, $< T, \circ >$ is isomorphic to the subset of 4×4 invertible matrices of the form

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & \cos\theta & -\sin\theta & 0 \\ b & \sin\theta & \cos\theta & 0 \\ \theta & 0 & 0 & 1 \end{bmatrix}, \text{ where } a, b, \theta \in R. \qquad 5.49$$

To show that f is a homomorphism, let $q_1 = (x_1, y_1, \theta_1)^T$ and $q_2 = (x_2, y_2, \theta_2)^T$ be elements of $< T, \circ >$. It follows that

$$q_1 \circ q_2 = \begin{bmatrix} x_1 \\ y_1 \\ \theta_1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ \theta_2 \end{bmatrix} = \begin{bmatrix} x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 \\ \theta_1 + \theta_2 \end{bmatrix} \qquad 5.50$$

Therefore

$$f(q_1 \circ q_2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 & \cos(\theta_1 + \theta_2) & -\sin(\theta_1 + \theta_2) & 0 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 & \sin(\theta_1 + \theta_2) & \cos(\theta_1 + \theta_2) & 0 \\ \theta_1 + \theta_2 & 0 & 0 & 1 \end{bmatrix} \qquad 5.51$$

It follows that

$$f(q_1) \times f(q_2) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ x_1 & \cos\theta_1 & -\sin\theta_1 & 0 \\ y_1 & \sin\theta_1 & \cos\theta_1 & 0 \\ \theta_1 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ x_2 & \cos\theta_2 & -\sin\theta_2 & 0 \\ y_2 & \sin\theta_2 & \cos\theta_2 & 0 \\ \theta_2 & 0 & 0 & 1 \end{bmatrix} = \qquad 5.52$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ x_1 + x_2\cos\theta_1 - y_2\sin\theta_1 & \cos(\theta_1+\theta_2) & -\sin(\theta_1+\theta_2) & 0 \\ y_1 + x_2\sin\theta_1 + y_2\cos\theta_1 & \sin(\theta_1+\theta_2) & \cos(\theta_1+\theta_2) & 0 \\ \theta_1+\theta_2 & 0 & 0 & 1 \end{bmatrix} \quad 5.53$$

using the trigometric identities for the sum of two angles. □

Although the groups $< T, \circ >$ and $< T', \times >$ are isomorphic, operations in the transformation group $< T, \circ >$ are simpler to represent and are computationally more efficient than the matrix multiplication in $< T', \times >$. This computational efficiency is essential for transformation calculations performed by an autonomous robotic platform. Obviously, this transformation system does not have any singularity.

F. SUMMARY

Dead reckoning error correction is essential to automated cartography since significant robot motion is required to map a world space. All accumulated vehicle error must be reconciled during automated cartography motion or the resulting map will be useless. A 2D configurational algebra based upon group theory provides a solution to the accumulated vehicle error problem. The group theory in this chapter provides an elegant algebraic structure that makes robot motion calculations more transparent. Motion analysis and robot dead reckoning error determination are made possible using this algebra. This system is complete in the sense that for every possible situation, the robot's configuration and its dead reckoning error can be represented. These properties arise from group theory which implies all configurations have a unique inverse and there is no singularity on any configuration.

One of the open problems related to this theory is whether there exists a similar theory in three dimensional transformations. i.e., how can the "composition" and "inverse" for transformations $q = (x, y, z, \phi, \theta, \psi)^T$ be defined. Here, ϕ, θ and ψ are Euler angles [Paul 84]. The periodic detection and reduction of odometry errors allow an autonomous mobile

robot to work for a sustained period with great precision. This theory has been experimentally validated using Yamabico-11 and the results are described in Chapter IX.

VI. REPRESENTATION OF THE WORLD

The automated cartography algorithm employs a polygonal map representation. This has also been called abstract mapping since the computational complexity is related to the number of features in the robot's world space instead of the number of grid squares. Leonard called this type of map a feature-based map [Leonard 91]. The relatively low computational complexity required to build an abstract map enables a robot to perform the automated cartography in real time using on board computing resources. Cartography using only on board computers is more efficient since there is no communication delay associated with communications to another computer and there are no inherent range restrictions. In this chapter, the abstract data structure for the world representation is defined.

A. REPRESENTATION OF A POLYGON

1. Example Polygons

Polygons are used as the means of representing free space for this dissertation. Polygons are the basic building blocks for the world representation. A polygon is a collection of vertices connected by edges that divide the Cartesian plane into two regions. In Figure 6.1, several example polygons are shown. The simplest polygon is a triangle, since a polygon must have three or more edges. All polygons in this dissertation are simple, to distinguish them from polygons that cross themselves. A polygon with n vertices is called an *n-gon.*

2. Definitions

A vertex is defined as a point in the Cartesian plane such that $v = (x, y)$. An edge is a directed line segment designated by $e = \{v, v', type\}$ which represents an ordered pair of vertices (v, v') and a *type* ("real" or "inferred"). An example of an edge is

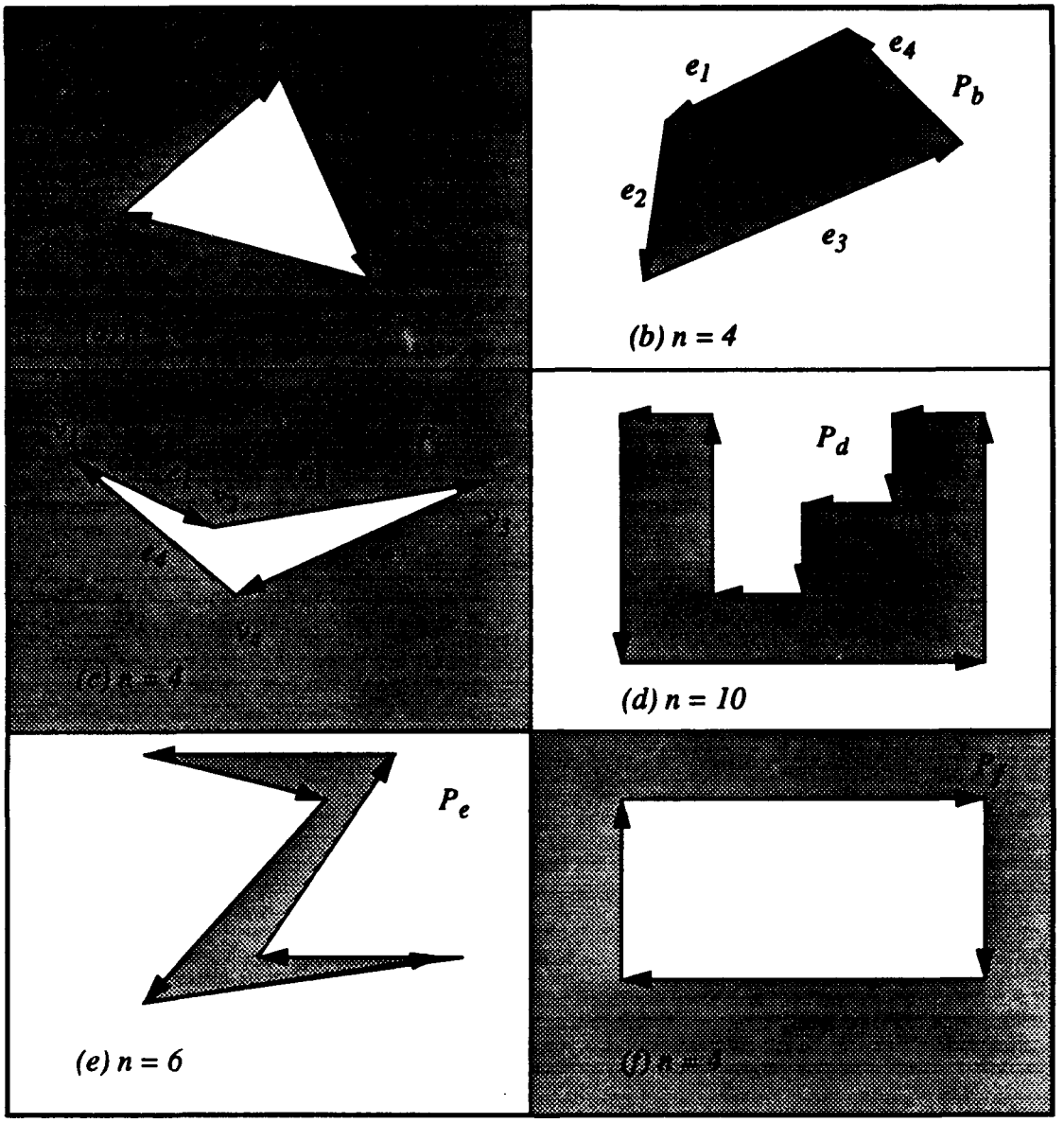


Figure 6.1 - Example Polygons

e_2={v_2,v_3,"*real*" } in Figure 6.1 (c). The edge e_2 = {v_2, v_3, "*real*" } consists of a vertex pair (v_2, v_3) and a *type* = "*real*".

A polygon P = {E, e, *next*} is a tuple containing a set of at least three edges E = (e_1, e_2, ..., e_n) , a first edge e, and a *next* function. The edges and the *next* function are selected such that no pair of nonconsecutive edges share a point [O'Rourke 87]. The *next*

function defines two types of polygons based upon the order of the edges in the polygon; *hole*-type and *boundary*-type. For *hole* polygons the edges are directed in counterclockwise order and for *boundary* polygons the edges are directed in clockwise order. For example, in Figure 6.1 (b), polygon $P_b = \{\{e_1, e_2, e_3, e_4\}, e_1, next\}$ is a *hole* type polygon with four edges that are directed counterclockwise. In Figure 6.1 (c), polygon $P_c = \{\{e_1, e_2, e_3, e_4\}, e_1, next\}$ is a boundary type polygon with four edges directed clockwise. The domain of the *next* function is closed with respect to the polygon's edge list. The vertices of any sequential edges are coincident such that if $next(v_1, v_2) = (v_1', v_2')$ then $v_2 = v_1'$. The *next* function is an one-to-one function such that every edge has a predecessor. A polygon with n edges is a closed, directed cycle such that $(\forall e)\, next^n(e) = e$.

The orientation of an edge is defined as the angle a directed edge makes with the x-axis as illustrated in Figure 6.2. The orientation of a given edge $e = \{v_1, v_2, type\}$ is given by the function $\psi(e)$. The external angle γ of a given edge is the angle formed by the ex-

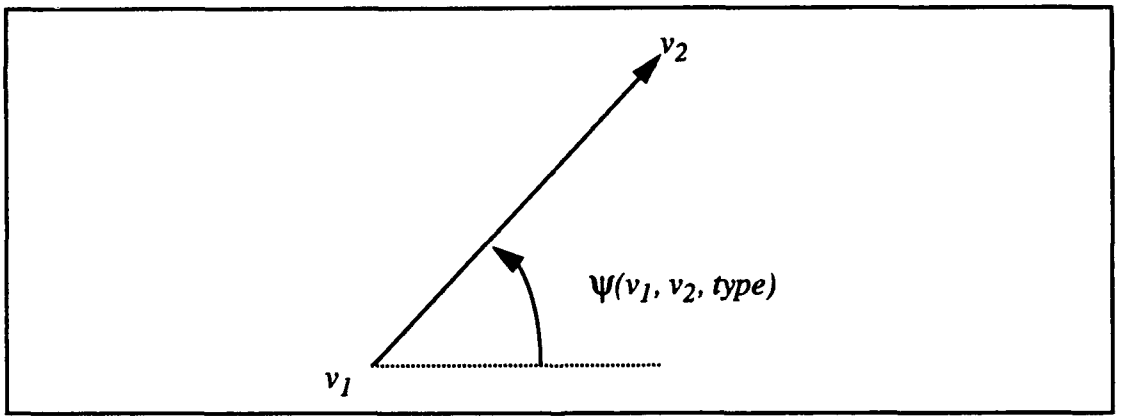


Figure 6.2 - The Orientation of an Edge

tension of a given edge and the next edge of a polygon. Expressed mathematically, the external angle is; $\gamma_i = \psi(next(e_i)) - \psi(e_i)$. The external angle concept is illustrated in Figure 6.3. Kanayama proved that the sum of the external angles of all the edges in a polygon is

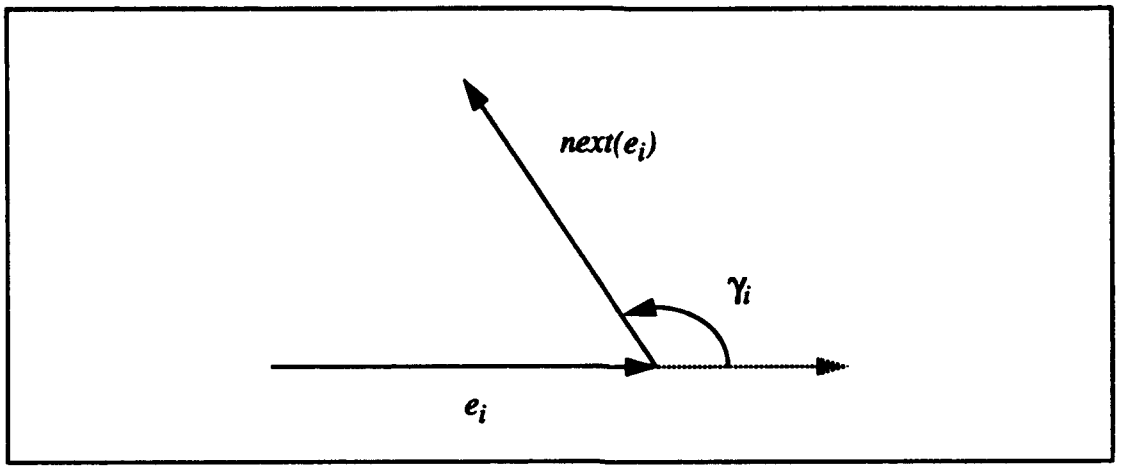


Figure 6.3 - The External Angle

equal to $\pm 2\pi$ in accordance with the equation $\Sigma\gamma_i = \pm 2\pi$ [Kanayama 91]. For *hole*-type polygons $\Sigma\gamma_i = +2\pi$ and for *boundary*-type polygons $\Sigma\gamma_i = -2\pi$.

The collection of edges and vertices is referred to as the boundary of P, denoted by ∂P. The inside of a polygon is defined as the infinite set of points such that a non-osculating, directed half line drawn from any point intersects the boundary of the polygon an odd number of times [Kanayama 91]. This concept is illustrated in Figure 6.4. The point p_1 is inside the polygon P since the directed half-line L_1 intersects the polygon P three times. Another directed half-line (L_2), also drawn from point p_1, intersects the polygon P once. The point p_3 is not inside (it is outside) of P since its directed half-line (L_3) intersects the boundary of P an even number of times.

The *free area* of a given polygon is the planar region defined by the boundary of that polygon. For a *hole*-type polygon the *free area* is the planar region to the right of any given edge and therefore outside of the region enclosed by the polygon. The region inside any hole polygon is defined as filled and the region outside is the free area. For example in Figure 6.1 (b), the filled side of the hole polygon P_b is the shaded region inside of P_b and the free area is the white region outside of P_b. For a *boundary*-type polygon the free area is also the planar region to the right of any given edge and inside the planar region enclosed

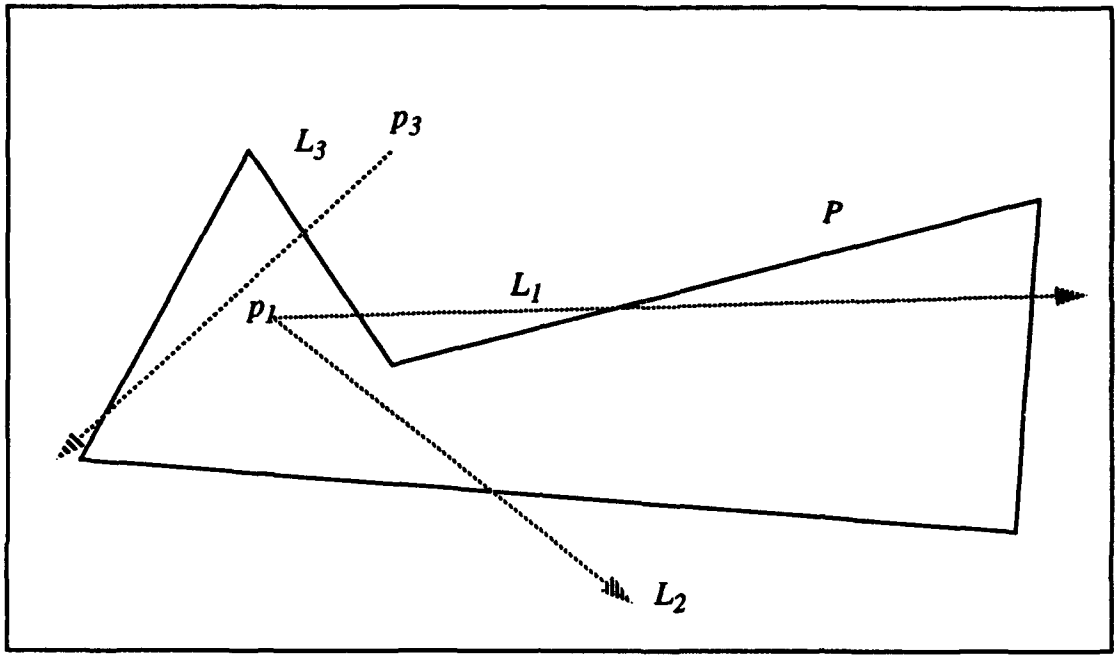


Figure 6.4 - Region Inside a Polygon

by the polygon. The free area of a boundary polygon P is the region inside ∂P and the filled area is the region outside ∂P. This concept is illustrated in Figure 6.1 (a). The free area of boundary polygon P_a is the white region inside ∂P_a and the filled area is the shaded region outside ∂P_a. The function *free(P)* gives the free region for any given polygon P and the function *filled(P)* gives the filled region of any polygon P.

An orthogonal polygon P is defined as a polygon whose edges are all aligned with a pair of orthogonal coordinate axes, which without loss of generality are taken to be horizontal and vertical [O'Rourke 87]. The polygons P_d and P_f in Figure 6.1 are examples of orthogonal polygons since their edges are all orthogonal to a pair of orthogonal coordinate axes. In Figure 6.1 (d) polygon P_d is an orthogonal hole polygon and P_f is an orthogonal boundary polygon.

The edge type designation is used for recording robot sensor input. "Real" edges are derived when the sensor observes some visible edge of its world space W by direct sen-

sor scan. "Inferred" edges are constructed edges that serve to bound unexplored or occluded portions of the world space.

B. REPRESENTATION OF A WORLD

Chapter VII describes the development of a series of three idealized robot cartography algorithms. These algorithm provide a firm theoretical basis for the real automated robot cartography algorithm that follows in Chapter VIII. Polygonal regions are used to represent the free space mapped by the automated cartography algorithm. This section gives some examples of world representations and defines the basic terms used in the algorithms. Worlds are compound polygons with the hole polygons used to represent obstacles in the world and with a single boundary polygon used to represent the exterior boundary of the known world space.

1. Example Worlds

In Figure 6.5, three example polygonal worlds are shown. The shaded area represents the free space bounded by each world. W_a is a world consisting of a boundary polygon with six edges and no holes. W_b is a world with one hole H_1. W_c is a world with two inferred edges and no holes. Notice in W_b that the edges of the boundary polygon P_o are directed in clockwise order. Further notice that W_b's hole polygon H_1 has edges directed in counter-clockwise order.

2. Definitions

A world W is defined by the pair $W = \{P_o, H\}$ such that P_o is an exterior boundary polygon with a sequence of zero or more simple polygonal holes $H = \{H_1, \ldots, H_h\}$. The free area of the world *free(W)* is a multiply connected free space with h holes: it is the region of the plane inside of P_o, but outside of $H_1, \ldots, H_h$. The polygon P_o is a *boundary-type* polygon with edges directed clockwise. The hole polygons $H_1, \ldots, H_h$ edges are ordered such that the edge number increases clockwise around each polygon as shown in Fig-

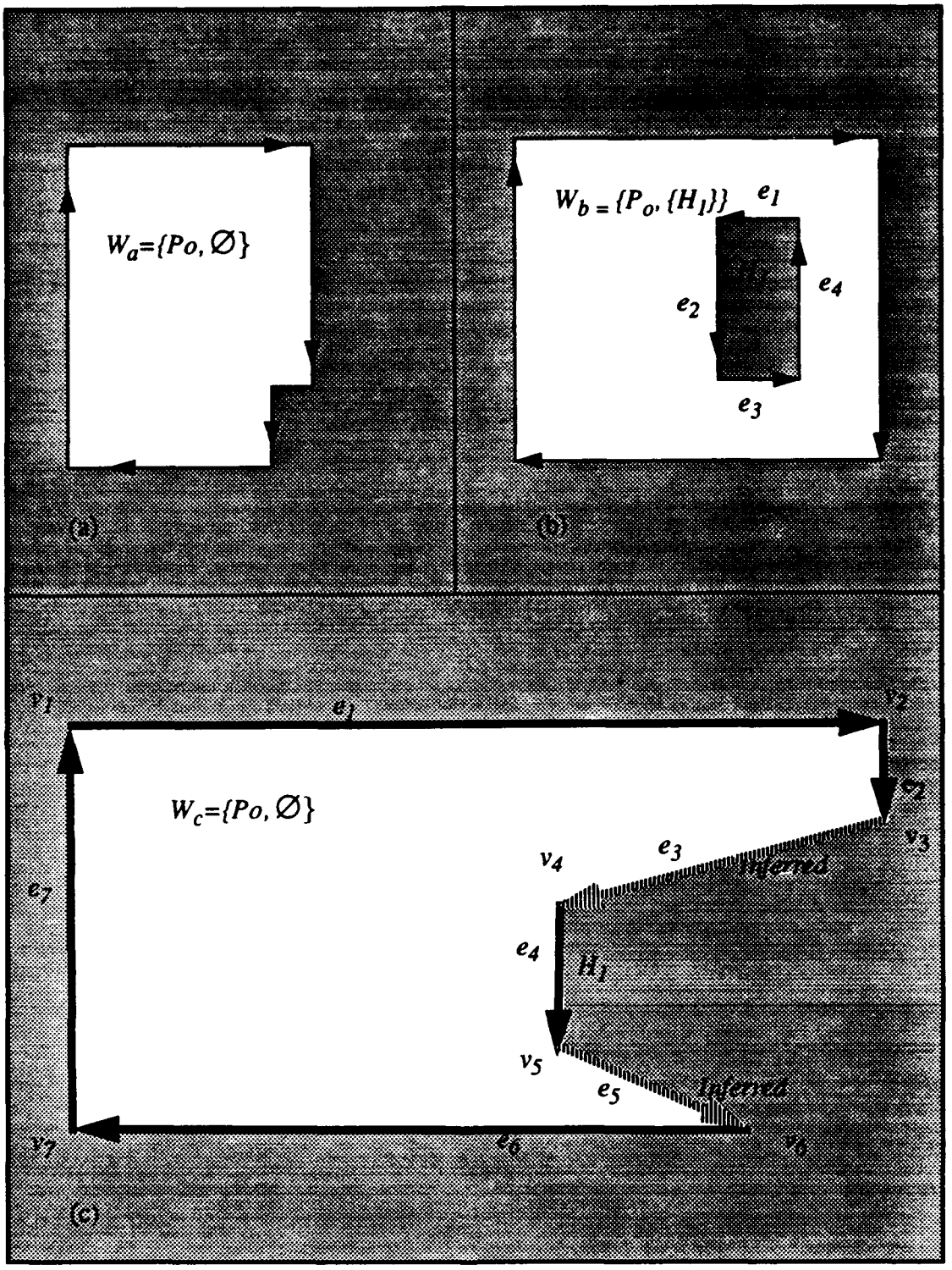


Figure 6.5 - Example Worlds

ure 6.5 (c). In this manner, the edges are directed line segments such that the left side of any given edge is the filled area of W and the right side of any edge is the free area of W as shown by the directed edges in Figure 6.5.

Let the area of the Cartesian plane bounded by a polygon P be represented by $|P|$ and let $|P_1| \cap |P_2|$ mean the intersecting planar area of polygons P_1 and P_2. Let *free(W)* represent the intersection of the free areas of all component polygons in the world W. The area bounded by a world W is the intersection of the area bounded by the world's boundary polygon P_o and all h of the world's hole polygons $H_1, H_2, \ldots H_h$ as given by Equation 6.1.

$$free(W) = free(P_o) \cap free(H_1) \cap \ldots \cap free(H_h) \qquad 6.1$$

Equations 6.2 and 6.3 define the required conditions for a world. Equation 6.2 means that no hole polygon may intersect the boundary polygon. Equation 6.3 means that no two hole polygons may intersect.

$$(\forall i)\ (filled(P_o) \cap free(H_i) = \varnothing) \qquad 6.2$$

$$\forall i, j \left[(i \neq j) \Rightarrow (filled(H_i) \cap filled(H_j)) = \varnothing \right] \qquad 6.3$$

An orthogonal world $W = \{P_o, H\}$ is defined as a world with all edges aligned to the same pair of coordinate axes. In Figure 6.5, $W_b = \{P_o, H\}$ is an orthogonal world, where $H = \{H_1\}$ is the polygon hole list containing one polygon H_1 and P_o is the boundary polygon.

A partial world *PW(W)* of the world W is a polygon with holes such that *PW(W)* represents some region inside of W such that $free(PW) \subseteq free(W)$. For the purpose of these definitions, inside is defined as any point in the planar region within the free space of the

boundary polygon and not within a hole polygon as given in Equation 6.1. The edges of *PW(W)* may be either "real" or "inferred". The "real" edges are the edges shared by *W* and *PW(W)*. The "inferred" edges are edges whose vertices both lie on edges of *W* and separate a region enclosed by both *W* and *PW(W)* from a region enclosed exclusively by *W*. Thus, the boundary of the partial world $\partial PW(W)$ must lie entirely within *W*. The null partial map $PW(W) = \varnothing$ of the world *W* is a partial world such that it has no holes (h=0) and the number of vertices in P_0 is zero. A completed partial world is a partial world such that $PW(W) = W$. *PW* is said to be a correct partial world of *W* if and only if $free(PW) \subseteq free(W)$.

C. SUMMARY

This chapter develops a method for world representation suitable for automated robot cartography. A polygon is the basic structure for representing a free space. A world is a complex polygon with one exterior boundary polygon and zero or more non-intersecting hole polygons all inside of the boundary polygon. The polygonal method of world representation is used for storing the current world model during cartography.

VII. THEORY OF AUTOMATED CARTOGRAPHY

Mobile robots that function autonomously in the real world have been a major objective within the Artificial Intelligence community since the first days of work on intelligent systems. The ultimate goal has been a robot that would navigate through an environment, managing to both learn the layout and to perform assigned tasks [Moravec 81]. To develop a firm basis for robot automated cartography this chapter examines robot cartography using a point robot with an idealized sensor. This theoretical examination is initially unencumbered by robot motion error or the sensor limitations described in Chapter II. This problem abstraction allows a theoretical examination of the proper representation for the robot's partial map and the proper robot motion planning to support automated cartography.

For automated cartography, the issue of how to sense the geometrical relations of the world is the central theme. In this chapter, three progressively less idealized sensors S_1, S_2, and S_3 are introduced and a cartography algorithm for the idealized robot using each idealized sensor is presented. In each algorithm, one additional characteristic of the real ultrasonic range finder's sensor is added. S_1 is an idealized, infinite range sensor that returns information regardless of target incidence angle. The S_2 sensor has infinite range capability but edge detection is limited by sensor beam incident angle. Finally, the S_3 sensor has both finite range and limited sensor beam incident angle capability. Each case study provides an algorithm and a proof of correctness. The progressively less idealized sensor algorithms serve to clarify the logical structure of the complex algorithm for the real robot. The idealized robot R used for this theory has perfect motion control precision so that the algorithms stated above give a map with a precision determined by the range sensors.

A. ALGORITHM FOR IDEALIZED SENSOR S_1

The idealized robot R is initially placed at an arbitrary location inside of W. R may move freely to any point inside of W with no dead reckoning error. This implies that R always has perfect knowledge of its position in the global coordinate system regardless of the distance it has traveled since its initial placement in the world. A robot R is defined as a point robot. The only constraint on R is that it must remain within *free(W)*.

1. Characteristics of the S_1 Sensor

Idealized Sensor S_1 - The sensor S_1 moves about in *free(W)* attached to R. This perfect sensor S_1 is a ray-tracing range finder [Leonard 91] with infinite range. S_1 returns the edges of any portion of W's edges regardless of the sensor beam incidence angle as long as the edge is visible from R. At any point x in W, S_1 is swept through an arc and extracts a list of edges Q from W.

R has a single idealized sensor S_x that is, fundamentally, an edge detector. All edges both visible from R and detectable by S_x are said to be illuminated or detected by S_x. S_x operates either by sweeping a circular arc from R's current position or by scanning left and right perpendicular to R's path while R is in motion. R may remain stationary or move while S_x is operating. S_x has an edge-extracting capability such that any edge or portion of an edge of W that is illuminated by S_x will be returned to R as a "real" edge.

In Figure 7.1, R sweeps the S_x sensor about a 360° arc at point C_1. All edges visible from R are extracted. A boundary polygon for the first partial world PW is formed from the edges extracted from the C_1 sweep. Notice that edges e_a and e_b are not visible from point C_1. An "inferred" edge e_3 is constructed to connect the discontinuity between e_2 and e_4. R moves to point C_2 and performs a second sweep which reveals edges e_a and e_b. The "inferred" edge e_3 may be removed since the region beyond it has now been scanned.

In the special case, an "inferred" edge may turn out to represent a real boundary surface as shown in Figure 7.2. In this figure, edge e_3 happens to be one of W's edges. In

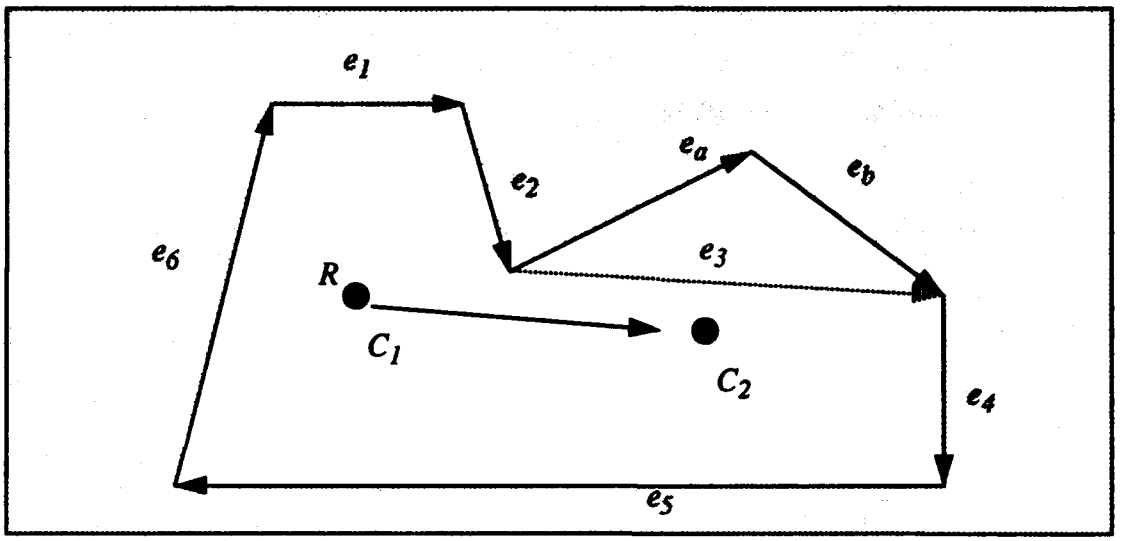


Figure 7.1 Idealized Robot Cartography Sweep

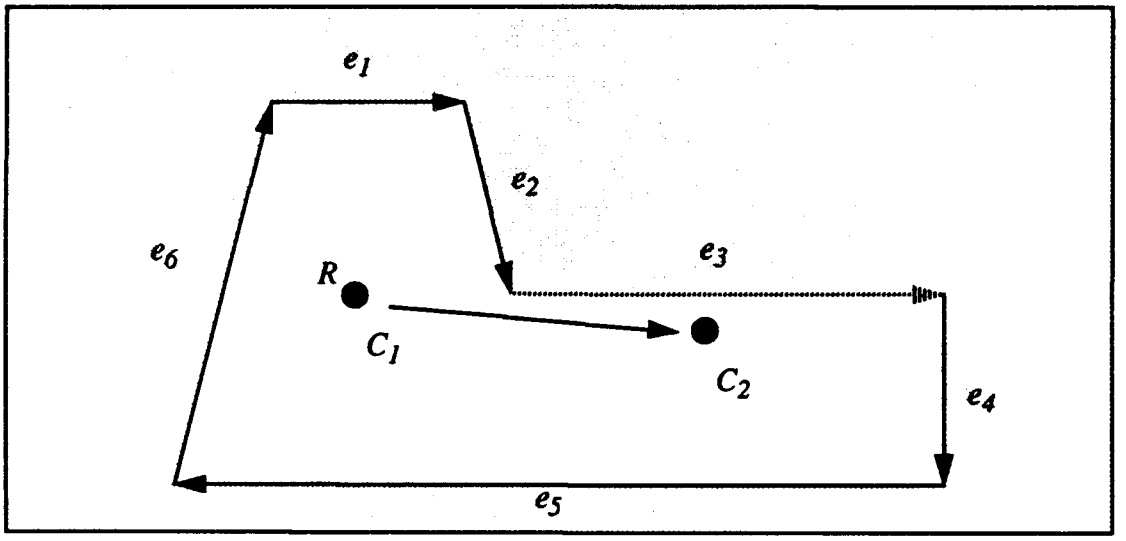


Figure 7.2 Idealized Robot Cartography Sweep Special Case

Figure 7.2, R obtains a partial world PW_1 at position C_1. The partial world boundary polygon $PW_1 \bullet P_0$ has five "real" edges e_1, e_2, e_4, e_5, e_6 and one "inferred" edge e_3 since S_x detects a discontinuity between edges e_2 and e_4. R moves to C_2 and performs a second sensor sweep. This sweep does not reveal any additional free region beyond edge e_3. The sweep does, however, reveal that edge e_3 is in fact a "real" edge and therefore part of W. This fact

was impossible to verify from the position C_1 since C_1 happened to lie on a line extended from e_3 to the left. Therefore, as long as "inferred" edges remain in R's partial world, R needs to move to obtain additional sensor data from another viewpoint. The necessary motion to optimize R's view of an occluded region is a fundamental component of intelligent robot exploration.

The algorithms for the idealized sensors are now described. The S_1 sensor is a perfect robot sensor with infinite range for target detection. S_1 also extracts edges regardless of sensor beam incidence angle. The S_2 sensor also has infinite range capability, but extracts edges within a limited sensor beam incidence angle. The S_3 sensor is further limited by both finite range and limited sensor beam incidence angle. Further abstract sensors are unnecessary since the important non-ideal sensor limitations are addressed by S_1 through S_3.

The *full_sweep* and *arc_sweep* functions control S_1 as described in the S_1 algorithm later in this section. It is assumed that a full 360° sweep of S_1 from any point x inside of W yields a correct partial world PW. It is also assumed, that given that PW_n is a correct partial world, an S_1 sweep of any "inferred" edge from a point x inside of PW_n gives an edge list Q such that when Q is merged with PW_n a correct partial world PW_{n+1} is derived.

World W - The world W is a world as defined in Chapter VI. The world W has a finite number of holes and the boundary polygon P_0 of W has a finite number of edges. All holes $H_1, \ldots, H_h$ have a finite number of edges. All edges of W have finite length. An arbitrary parameter σ exists such that no hole polygons are closer than 2σ to the boundary polygon P_0 or to any other hole polygon.

2. Example of Behavior

To explain how the idealized automated cartography algorithm works, two examples are presented. In the first example, R performs cartography by detecting the boundaries of a planar, closed polygonal world W. R is placed at any arbitrary point C inside of W, with

no knowledge of the configuration space geometry. R first sweeps sensor S_1 about a full circle. This is essentially an idealized visibility sweep that extracts a partial world PW [O'Rourke 87]. From the 360° edge data, R derives an edge list Q. In Figure 7.3(a), the shaded area is the sensor's sweep volume.

The partial world PW_1 has a boundary-type polygon P_o and no hole polygons. P_o has an edge list E_1, a first edge e_1, and a *next* function that determines the edge order as shown by Equations 7.1 and 7.2.

$$PW_1 = (P_o, \varnothing) \tag{7.1}$$

$$P_o = (\{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}, e_1, next) \tag{7.2}$$

In PW_1 in Figure 7.3 (b), each of the consecutive pair of edges in $E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}\}$ share a common vertex and the polygon forms a closed cycle such that the first (e_1) and last edges (e_{10}) share a common vertex. In the polygon $PW_1 \bullet P_o$, the edges are classified as "real" or "inferred" based upon how they are derived from sensor input. "Real" edges represent the visible edges of W from C_1. "Inferred" edges are derived from discontinuity between the "real" edges. In the boundary polygon of the partial world $PW_1 \bullet P_o$ in Figure 7.3 (b), edges e_3 and e_8 are "inferred" and the rest of the edges are "real". Therefore, an "inferred" edge represents S's inability to sense beyond some obstructing portion of its environment. To perform the next step, R must chose a new optimum configuration (or viewpoint) inside of PW_1 to perform the second sensor sweep. A depth first search (DFS) type algorithm selects the next "inferred" edge for investigation. In this case the "inferred" edge e_8 is chosen to be investigated next. R moves to position C_2 near the center of edge e_8 and inside of PW with no dead reckoning error (idealized robot motion). Figure 7.3 (c) illustrates R's choice for the second sweep configuration.

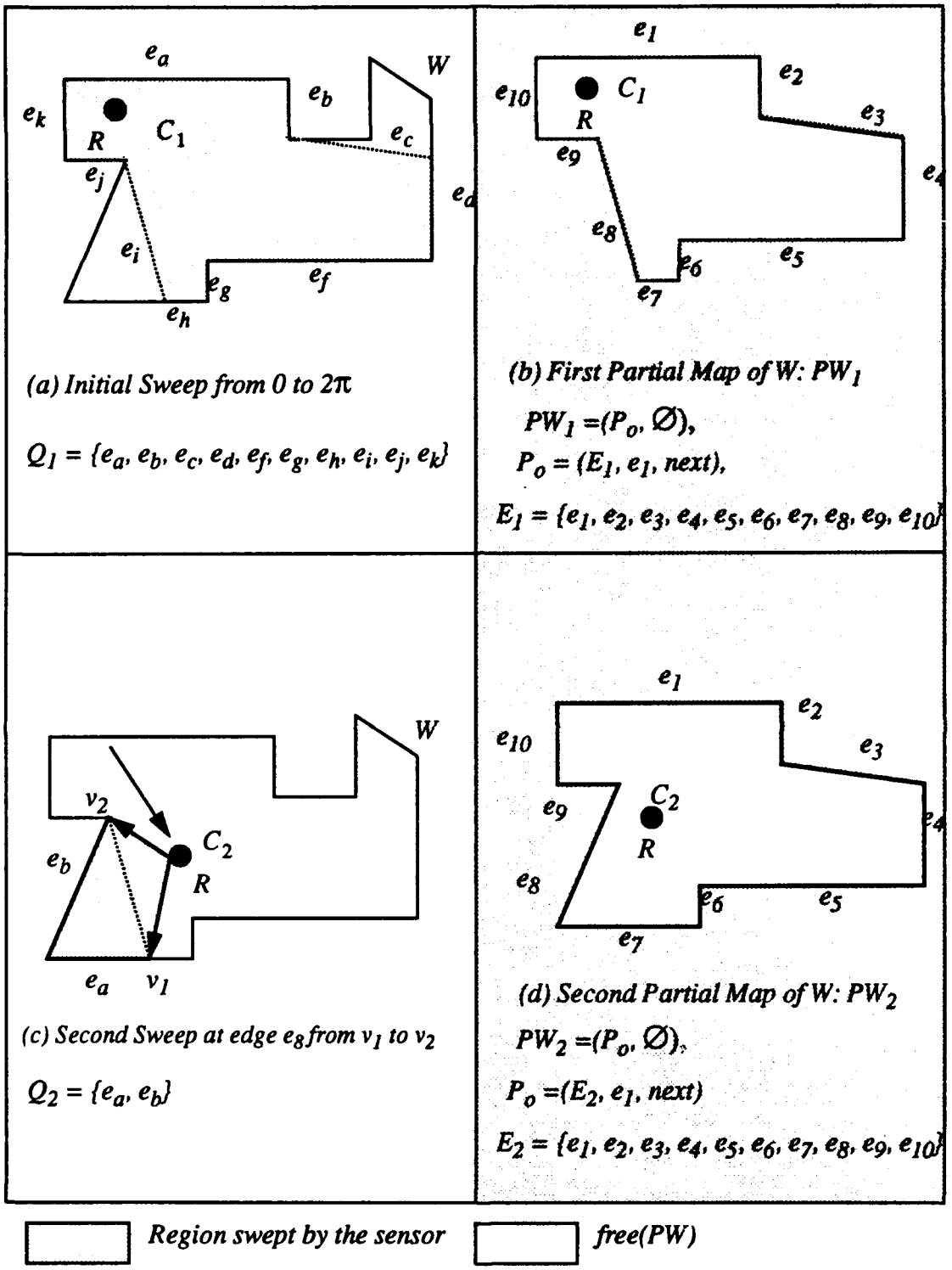


Figure 7.3 - S_1 Sensor Idealized Robot Cartography Example

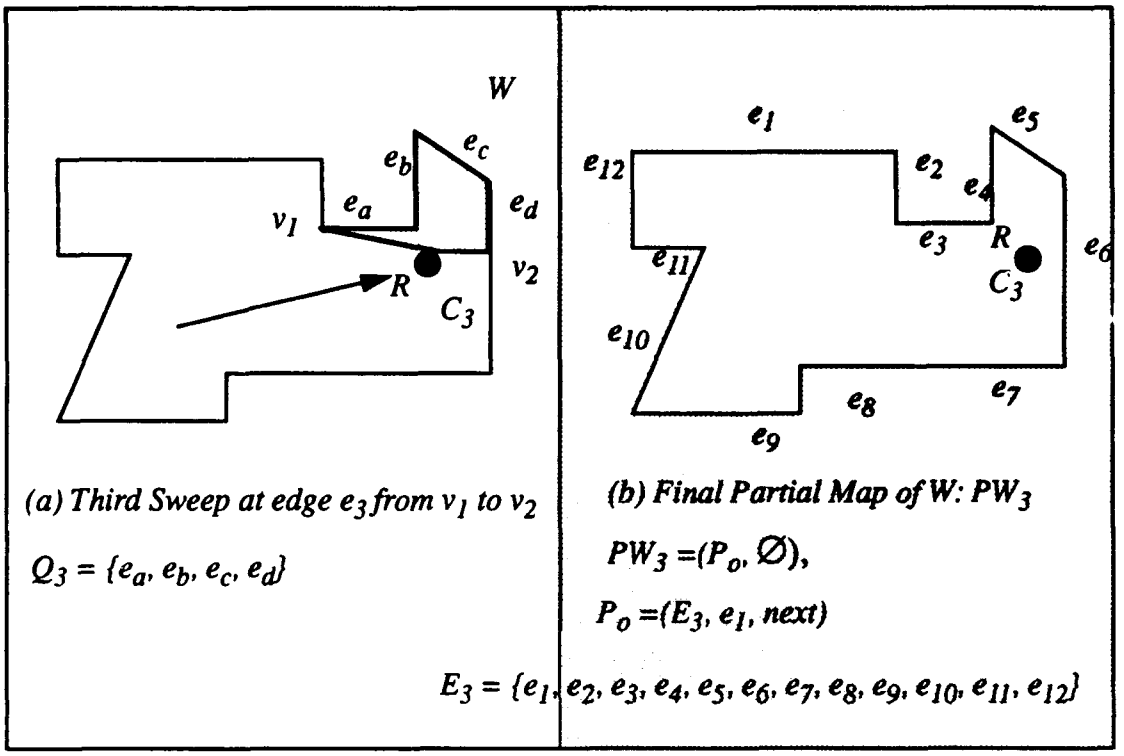


Figure 7.4 - S_1 Sensor Idealized Robot Cartography Example

For the second sensor sweep R moves to a point normal to the center point of the "inferred" edge e_8. This point is called C_2. R sweeps the sensor S_1 clockwise from vertex v_1 to vertex v_2 on edge e_8 in Figure 7.3 (c). The edge list for the partial world PW_1 is represented by $PW_1 \bullet P_o \bullet E_1$. Then the derived "real" edge list $Q_2 = \{e_a, e_b\}$ replaces the "inferred" edge e_8 in accordance with the Equation 7.3. Edges e_7 and e_a are collinear and are therefore combined into one new boundary polygon edge e_7. The new edge e_b gets renumbered as e_8 and all of the edges in the boundary polygon get renumbered in consecutive fashion.

$$PW_2 \bullet P_o \bullet E_2 = PW_1 \bullet P_o \bullet E_1 - \{e_8\} \cup \{e_a, e_b\} \quad 7.3$$

This process is accomplished by a "merge" algorithm.

The second partial world PW_2 is illustrated in Figure 7.3 (d). The only remaining "inferred" edge in PW_2 is e_3. This edge is investigated next. R moves from point C_2 to C_3 in Figure 7.4 (a). Again point C_3 is inside of PW_2, normal to the center of "inferred" edge e_3 and a distance σ inside of PW_2. R performs the next sensor sweep clockwise from vertex v_1 to v_2 on edge e_3 in Figure 7.4 (a). The extracted edge list is $Q_3 = \{e_a, e_b, e_c, e_d\}$. The final partial world PW_3 is derived by a *merge* function as shown by the Equation 7.4.

$$PW_3 \bullet P_o \bullet E_3 = PW_2 \bullet P_o \bullet E_2 - \{e_3\} \cup \{e_a, e_b, e_c, e_d\} \quad 7.4$$

The remaining occluded edges of the world are revealed and the S_1 cartography is completed. Notice that the final world has twelve edges all "real". When all edges in WP_n are "real", the algorithm returns the completed partial world $PW_n = W$ as shown in Figure 7.4 (b).

A second automated cartography example serves to explain S_1 cartography on a world with holes $W = \{P_o, H\}$. In Figure 7.5 (a), R is placed at point C_1. R sweeps sensor S_1 in a full circle and extracts the edge list $Q_1 = \{e_a, e_b, e_c, e_d, e_f, e_g, e_h, e_i\}$. The first partial world $PW_1 = \{P_o, \varnothing\}$ is constructed in accordance with Equation 7.5.

$$PW_1 \bullet P_o \bullet E_1 = Q_1 \quad 7.5$$

$H = \varnothing$ since no hole yet exist in $PW_1 \bullet P_o$ as illustrated in Figure 7.5 (b). The algorithm selects the closest "inferred" edge in PW_1 for DFS. R next moves to point C_2 and performs a second sweep on "inferred" edge e_6 of PW_1, as illustrated in Figure 7.6 (c). The

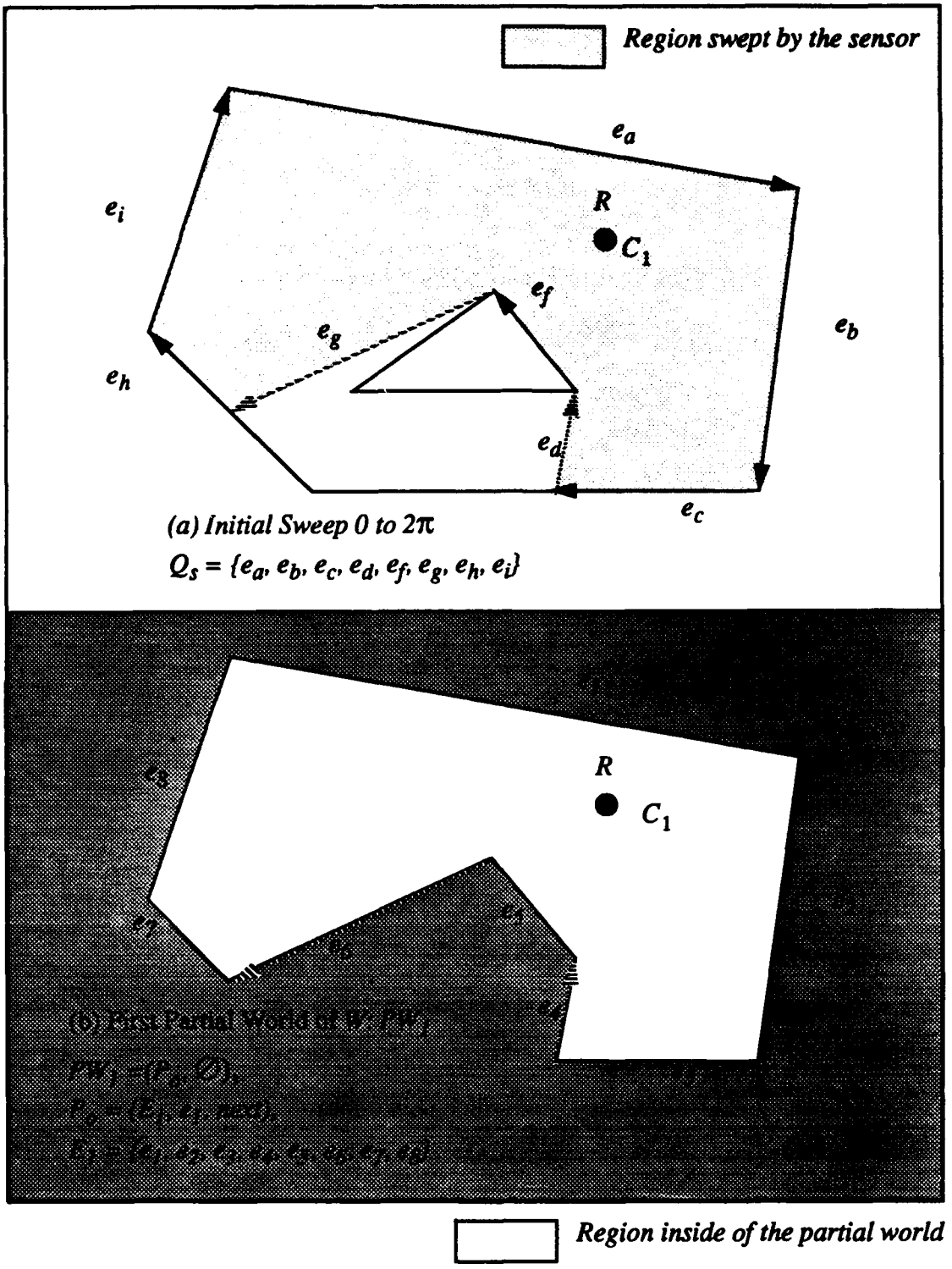


Figure 7.5 - S_1 Sensor Idealized Robot Cartography

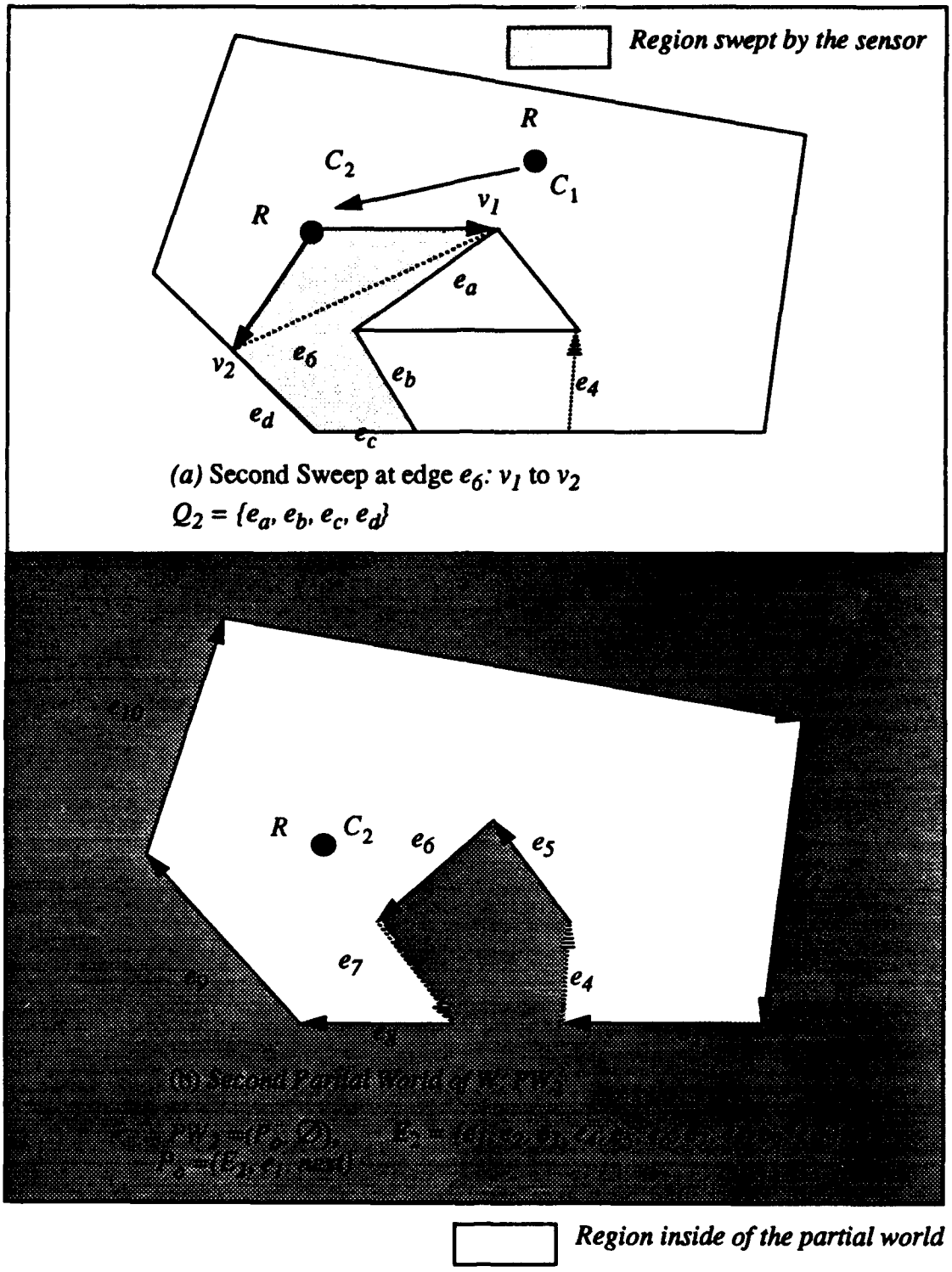


Figure 7.6 - S_1 Sensor Idealized Robot Cartography Example

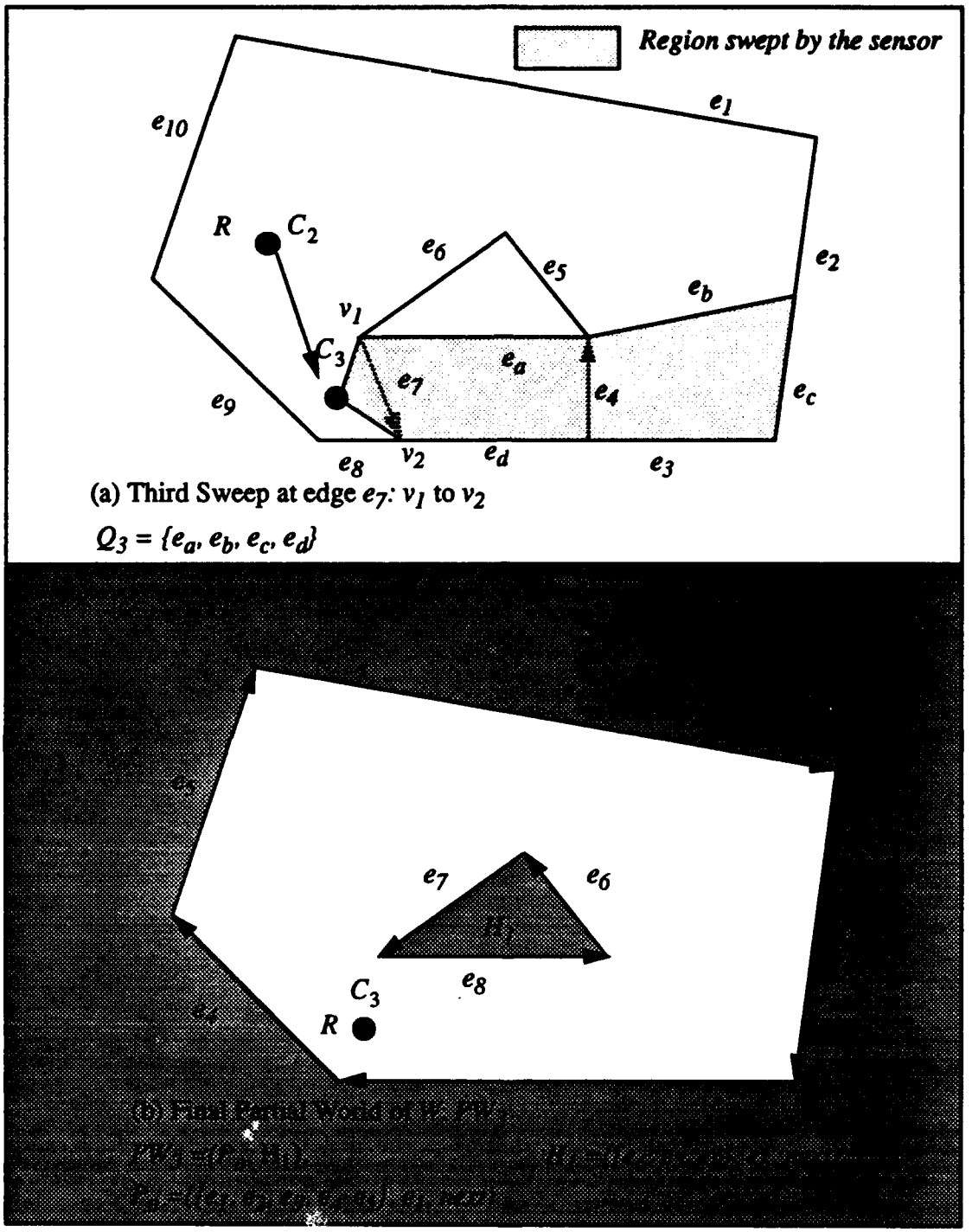


Figure 7.7 - S_1 Sensor Idealized Robot Cartography Example

extracted edge list $Q_2 = \{e_a, e_b, e_c, e_d\}$ is merged with PW_1 in accordance with Equation 7.6.

$$PW_2 \bullet P_o \bullet E_2 = PW_1 \bullet P_o \bullet E_1 - \{e_6\} \cup \{e_a, e_b, e_c, e_d\} \qquad 7.6$$

The second partial world is $PW_2 = \{P_o, \varnothing\}$. Notice that there are still no holes in PW_2 so $H = \varnothing$ as illustrated in Figure 7.6 (b). There are two "inferred" edges e_4 and e_7. In Figure 7.7 (a) the algorithm continues depth first search "inferred" edge e_7 in PW_2. R moves to point C_3 for the next sensor sweep. R sweeps the sensor S_1 from vertex v_1 to v_2 on "inferred" edge e_7. This sweep extracts four new edges; e_a, e_b, e_c, and e_d. Notice that the "inferred" edges e_4 and e_7 are included inside of the region sweep by S_1. Therefore, these edges are removed. Further, the edges e_5, e_6, and e_a bound a closed interior region. This region becomes a hole polygon as shown in Figure 7.7 (b) such that $H_1 = \{\{e_6, e_7, e_8\}, e_6, next\}$. The edge e_b is "inferred" and included inside of PW_2, therefore it is discarded. In Figure 7.7 (a), edges e_c and e_2 from PW_2 are coincident so e_c is discarded. Edges e_d, e_8, and e_3 in Figure 7.7 (a) are also collinear and are combined to form the new boundary edge e_3. Therefore, a new boundary polygon for PW_3 is derived by the Equation 7.7.

$$PW_3 \bullet P_o \bullet E_3 = PW_2 \bullet P_o \bullet E_2 - \{e_6\} \cup \{e_a, e_b, e_c, e_d\} \qquad 7.7$$

The final partial world PW_3 is $PW_3 = \{P_o, \{H_1\}\}$ as illustrated in Figure 7.7 (b). Notice that the edges for the boundary polygon P_o are directed clockwise and the edges of the hole polygon H_1 are directed counterclockwise.

3. Algorithm

This algorithm for idealized automated cartography is listed in Figure 7.8. At the top level, this algorithm is relatively straightforward. Initialization of variables occurs, an

```
S1_automated_cartography()
{
    World PW = (∅, ∅);
    Real σ;
    Path_list PL = ∅;
    Edge_List Q = ∅;
    Point C = (0, 0);
    Edge e;
    Vertex v_0, v_1;

    Q = full_sweep(C);
    PW.P_0.E = Q;
    while (not complete(PW))
    {
        e = DFS_edge(PW, C);
        v_0 = first_vertex(e);
        v_1 = second_vertex(e);
        C = next_position(v_0, v_1, σ)
        move_to(C, PL, PW);
        Ψ_0 = Ψ(C, v_0);
        Ψ_1 = Ψ(C, v_1);
        Q = arc_sweep(C, Ψ_0, Ψ_1);
        PW = merge_edges(e, Q, PW);
    } /* end while */
    return PW;
}
```

Figure 7.8 - The S_1 Idealized Robot Cartography Algorithm

initial full sweep from the starting position is made and then a "while loop" is executed each step of the world space exploration until the world *PW* is evaluated as complete. There are ten basic functions; *full_sweep*, *arc_sweep*, *DFS_edge*, Ψ, *first_vertex*, *second_vertex*, *next_position*, *move_to*, *merge_edges*, and *complete*.

```
full_sweep(C)
Point C;
{
    Edge_List Q;
    Edge e_first, e_last, e;

    Sweep sensor S_1 in an arc clockwise 360°;
    Q = edge list derived from S_1;
    e_first = first_edge(Q);
    e_last = last_edge(Q);
    e = merge(e_first, e_last);
    Q = Q - e_first - e_last ∪ e;
    return Q;
} /* end full_sweep */
```

Figure 7.9 - The *full_sweep* algorithm

The *full_sweep(C)* function takes R's position C as input and returns an edge list Q of all edges of W visible from C. The *full_sweep* algorithm is listed in Figure 7.9. The first and last edges from the 360° sweep are merged together if necessary.

The *full_sweep* function constructs a point visibility polygon [O'Rourke 87]. In Figure 7.10 (b), R performs a S_1 full sweep from 0 to 2π at point C_1. Edge list $Q = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ is extracted. Then edges e_1 and e_6 are merged to form e_1. R moves to the centerpoint of edge e_3 and performs an *arc_sweep* on edge e_3 clockwise from v_1 to v_2. Sensor S_1 detects five new edges beyond edge e_3. The edges $Q = \{e_a, e_b, e_c, e_d, e_f, e_g\}$ replace edge e_3 in $PW_2 \bullet Po$.

The *arc_sweep*(C, ψ_0, ψ_1) function sweeps the S_1 sensor clockwise at point C from the orientation ψ_0 to ψ_1 and returns an edge list Q. The algorithm for this function is listed in Figure 7.11. Q is a list of the newly found edges. The function returns an output of Q which has "real" edges derived from the visible portions of W's edges, and "inferred"

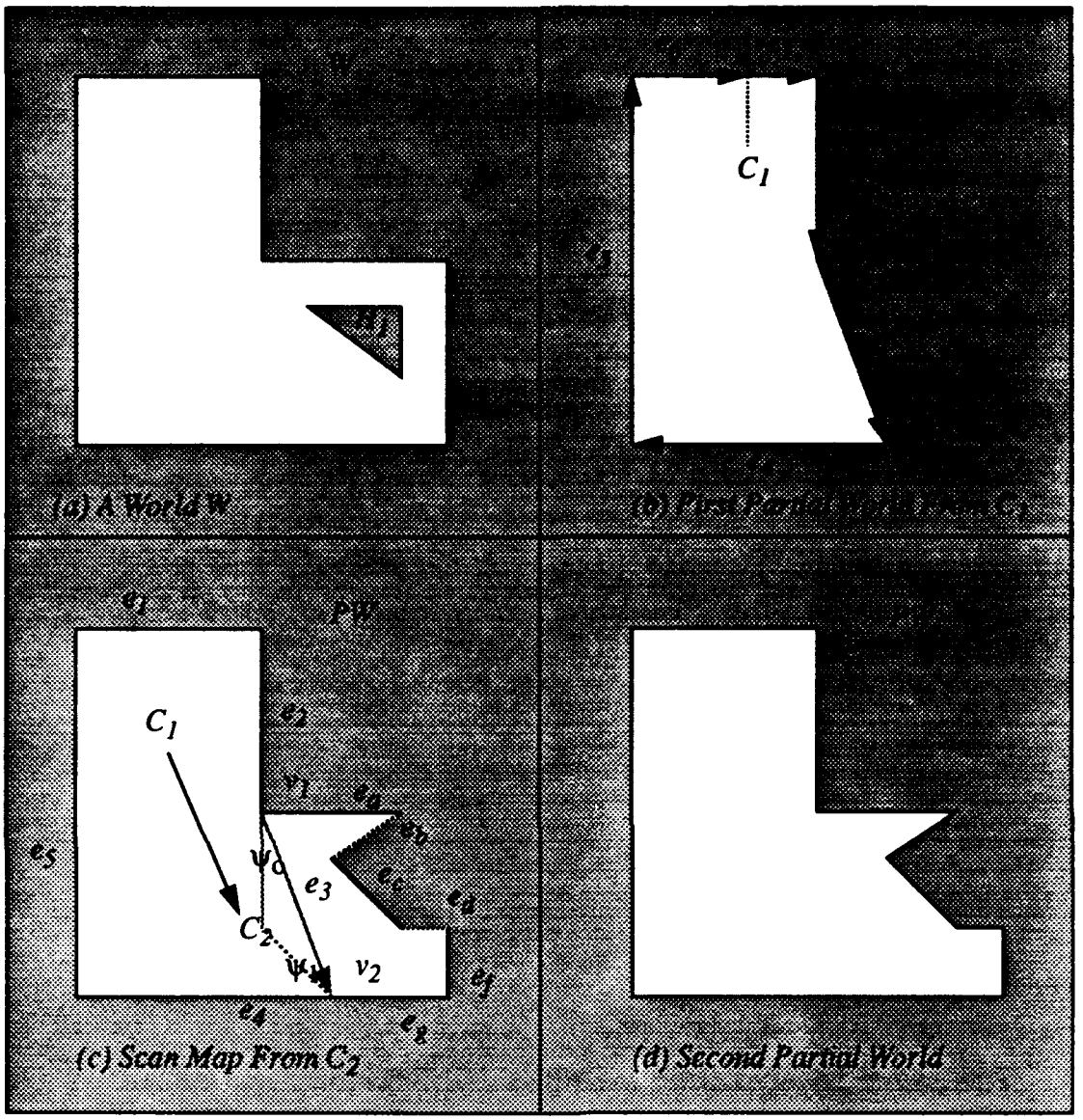


Figure 7.10 - Example of S_1 Sensor Sweep Operation

edges that serve to bound occluded regions of *W*. The "inferred" edges are special edges that are constructed to connect the discontinuity between "real" edges.

The *DFS_edge(PW, C)* function takes the current partial world *PW* and *R*'s current position *C* as input and returns the nearest edge suitable for sensor exploration. The function evaluates each existing "inferred" edge in *PW* and selects the best edge based upon minimizing the overall distance *R* must travel to complete the entire search. A depth first

```
arc_sweep(C, Ψ0, Ψ1)
Point C;
Orientation Ψ0, Ψ1;
{
    Edge_List Q;

    Align S1 sensor with orientation Ψ0;
    Sweep sensor S1 in an arc clockwise from orientation Ψ0 to Ψ1;
    Q = edge list derived from S1;

    return Q;
} /* end arc_sweep */
```

Figure 7.11 - The *arc_sweep* Algorithm

search strategy is used to guide exploration of complex worlds [Manber 89]. In other words, each branch of the world is investigated until an end is found to the branch or a hole is discovered.

The Ψ function returns the orientation between any two input points from the first point to the second point. The *first_vertex(e)* function returns the first vertex in the edge *e*'s ordered pair of vertices. For example, for v = *first_vertex*(v_0, v_1, *type*) such that $v = v_0$. Likewise, the *second_vertex(e)* function returns the second vertex of the edge *e* from its ordered pair of vertices. For example, for v = *second_vertex*(v_0, v_1, *type*) such that $v = v_1$. These functions are used to determine the vertices of any particular edge.

The *next_position*(v_0, v_1, σ) function returns a point suitable for sensor investigation of the edge represented by the pair of vertices (v_0, v_1). The σ parameter is a constant that represents an arbitrary standoff distance for *R* to investigate the edge. A standoff distance is necessary for S_1 to sweep the entire "inferred" edge with a non-zero sensor beam incidence angle. The point *C* is selected on a line constructed on the perpendicular bisector at a distance σ from the edge. This concept is illustrated in Figure 7.12.

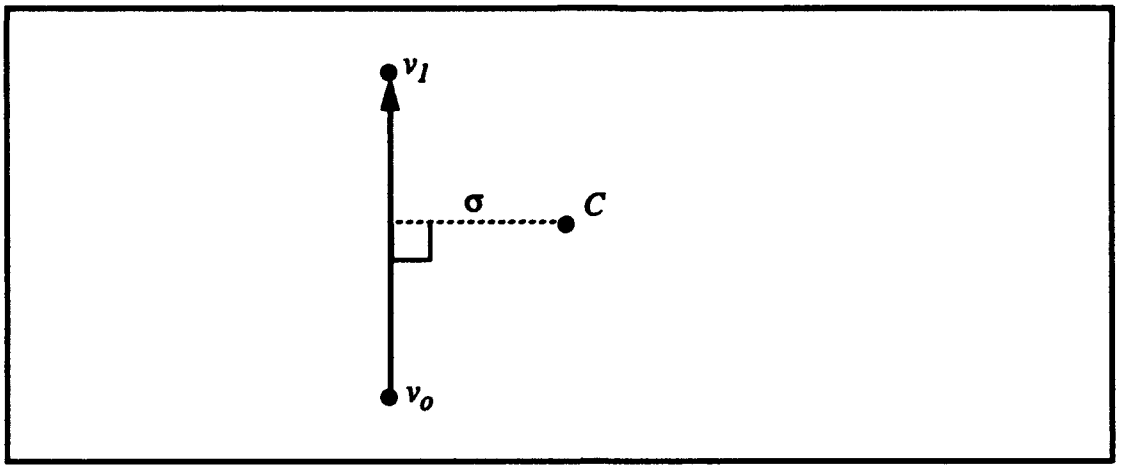


Figure 7.12 - The *next_position* function

The *move_to* function moves R to the new configuration in preparation for the next sensor sweep. This function plans a collision free path for R from the current position C_n to the new position C_{n+1} using the partial world model PW_n and PL. Then the function moves R to the new position using a series of one or more straight line path elements. R's motion falls into two categories; (1) movement to a visible "inferred" edge for an *arc_sweep* or, (2) backing tracking up the DFS tree to seek unexplored portions of the world space behind some "inferred" edge. This is a robot motion planning problem which has been well studied in the literature [Latombe 91] [Hwang 92].

The *merge_edge* function combines an edge list Q from an *arc_sweep* with the existing partial world PW. The algorithm for this function is listed in Figure 7.13. There are three input parameters; the "inferred" edge e under investigation, the derived edge list Q from the S_1 sweep, and the current partial world PW. This function combines the new edge list with the existing partial world PW_n and returns a new partial world PW_{n+1}. This function checks the edge list Q for indications of new hole formation. This is indicated when one of $PW \bullet P_0$ "inferred" edges is within the region swept by the *arc_sweep* function.

The *complete* function examines the existing partial world PW_n for "inferred" edges. If PW_n has no "inferred" edges, then *complete* returns TRUE; otherwise if PW_n has

```
merge_edges(e, Q, PW)
Edge e;
Edge_List Q;
Partial_World PW;
{
    Edge_List H;

    if (new_hole_detected(PW, Q))
        {
        H = extract_hole(PW, Q);
        PW.Po.E = PW.Po.E - H;
        PW.H = make_hole(H, e);
        }
    else
        PW.Po.E = PW.Po.E - e ∪ Q;
    return PW;
} /* end merge_edges */
```

Figure 7.13 - The *merge_edges* Algorithm

one or more "inferred" edges, then *complete* returns FALSE. A returned value of FALSE causes the overall algorithm to continue to run since all "inferred" edges must be resolved. When the *complete* function returns TRUE, this represents the terminating condition for the overall algorithm.

4. Proof of Correctness and Termination

The algorithm's proof requires that the idealized robot R can map any arbitrary world with holes in a finite number of moves with R starting at any point C inside of W. The proof is by induction. The algorithm terminates after a finite number of steps since the W has a finite total edge length and each iteration of the algorithm reveals at least minimum length of new edges of W.

PROPOSITION 7.1: Let PW_n be a partial world obtained by the n^{th} sweep in the world W of the idealized cartography algorithm listed in Figure 7.8. Then PW_n is a correct partial world derived from W.

Proof:

Basis: For sensor sweep number $n=1$, and PW_1 is a correct partial world of W by the assumption that a full S_1 sweep from any point inside of W yields a correct partial world.

Inductive Hypothesis: Assume that for sweep $n=m$ the proposition is correct.

Inductive Step: Then at $n=m+1$, R has a partial world PW_{m+1}. By the inductive hypothesis, the partial world PW_m is correct. By assumption, the partial world PW_{m+1} is also a correct partial world derived from W since PW_{m+1} is derived by merging PW_m with an arc sweep of some "inferred" edge in PW_m. □

LEMMA 7.1 : Given any world W there exists a minimum "inferred" edge length $l_o \geq 0$ such that the length l of any "inferred" edge in a partial world PW_n for some n generated by the S_1 algorithm in Figure 7.8 is greater than or equal to l_o ($l \geq l_o$).

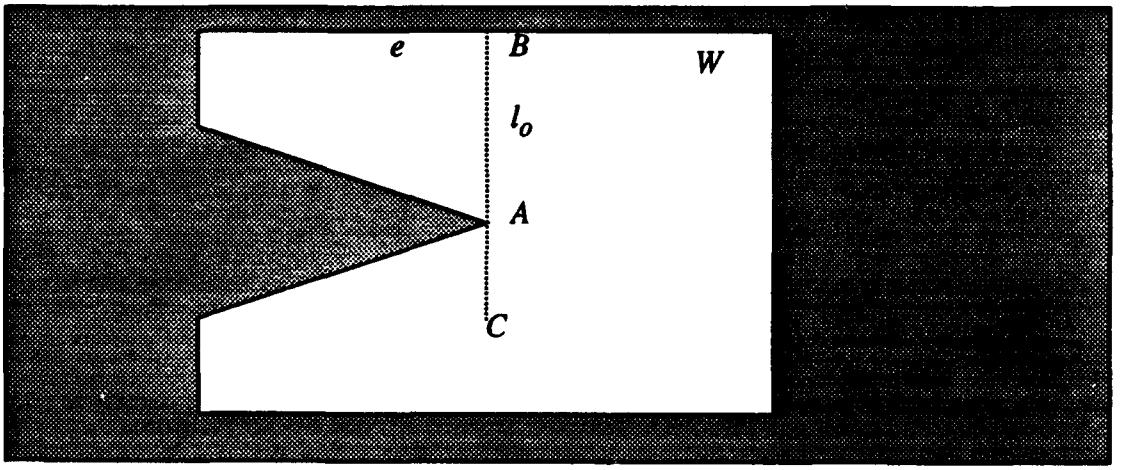


Figure 7.14 - Minimum Inferred Edge Length

For instance in Figure 7.14, in the world W the minimum length "inferred" edge l_o is shown. All other possible "inferred" edges have a length l longer than or equal to l_o.

Proof: An "inferred" edge $\overline{AB}$ generated by the S_1 algorithm in Figure 7.11 has the following properties:

(a) A is a vertex in W,

(b) There is an edge e in W such that B is on e,

(c) Vertex A is not a point on edge e,

(d) A and B are visible from each other.

A sensor position C is assumed as in Figure 7.14. Consider the minimum length "inferred" edge $\overline{AB}$. This is an "inferred" edge with one vertex fixed at A and the other vertex B on some edge of W. A minimum length l_{AO} exists for any A (this is from the definition of polygons and a world in Chapter VI) and $l_{AO} \geq 0$. Therefore if we let $l_O = min_{A \in W}(l_{AO})$ then l_O is a positive constant for any given world W. Therefore, for all "inferred" edges in PW, the length is greater than or equal to l_O. □

LEMMA 7.2 Given any sweep on an "inferred" edge e' by the S_1 algorithm in Figure 7.8 the total length of the "real" edges derived from the sweep is greater than or equal to the length of edge e'.

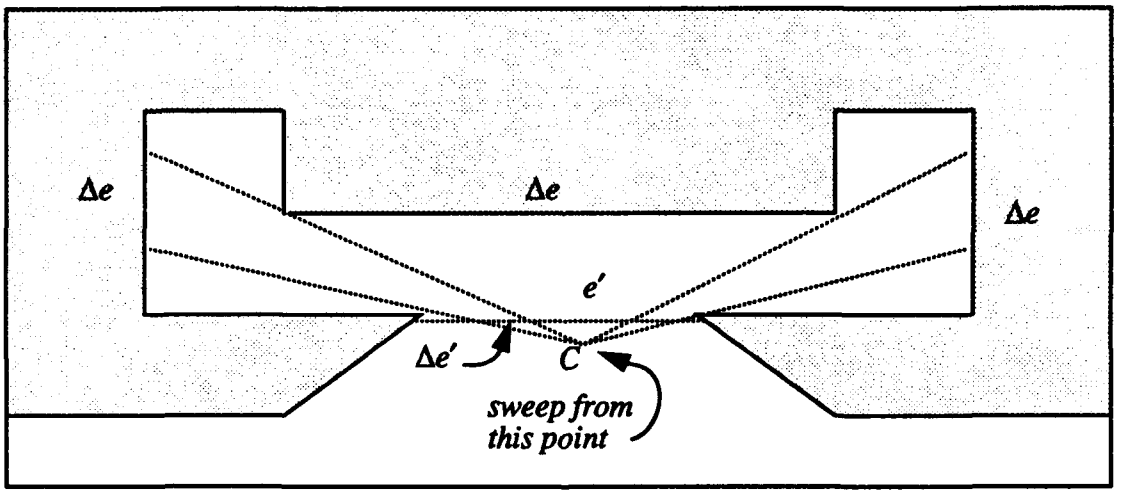


Figure 7.15 - An *arc_sweep* on "inferred" edge e'

Proof: Let the function *length(e)* represent the length of a given edge e. Then a portion Δe of a "real" edge projected onto W corresponds to $\Delta e'$ in the "inferred" edge e',

where $\Delta e \geq \Delta e'$. This concept is illustrated in Figure 7.15. By integrating both sides of the equation $\Delta e \geq \Delta e'$ over the length of e' it is determined that *length(e)* $\geq$ *length(e')*. □

PROPOSITION 7.2: The execution of the S_1 cartography algorithm listed in Figure 7.8 terminates.

Proof: The total length l_w of the edges in the world W is finite. In the n^{th} arc sweep, at least part of the "real" edges of W with a total length of l_n is found where $l_n \geq l_o$. Since l_o is a positive constant by LEMMA 7.2, the execution of the S_1 cartography algorithm terminates with at most $\left(\frac{l_w}{l_o}\right)$ iterations. □

Conclusion: Since any PW_{m+1} produced by the $m+1$ step of the algorithm is a correct partial world of W and since the algorithm terminates after a finite number of iterations, the idealized sensor S_1 cartography algorithm gives a correct complete world derived from W.

B. ALGORITHM FOR IDEALIZED SENSOR S_2

The S_2 sensor algorithm has one more non-ideal constraint placed upon it. This constraint concerns the sensor beam incidence angle. S_2's ability to extract edges from the world is limited by the incidence angle of its beam with respect to the incident edge of W. The S_2 cartography algorithm works only on orthogonal worlds because of the sensor beam incidence angle limitation.

1. Assumptions for the S_2 Algorithm

Idealized Sensor S_2 - The sensor S_2 moves about in W attached to the center point of R. This sensor S_2 is a modified, ray-tracing sensor with infinite range. S_2 differs from S_1 in that S_2 returns only the portion of an edge that is both visible and within the incident angle limits of $\pm\alpha$ of normal to the target surface as illustrated in Figure 7.16. For this reason,

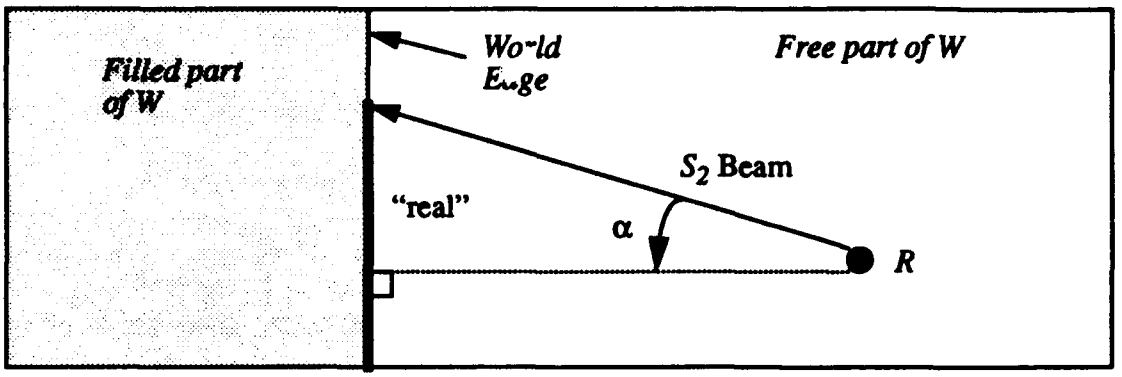


Figure 7.16 - S_2 Sensor Limited Incidence Angle Capability

R performs S_2 cartography primarily by translational scanning. As with the S_1 algorithm, S_2 connects the discontinuities between edges on W by connecting the "inferred" edges. These "inferred" edges bound the occluded regions inside of W that are not visible from R.

Suppose R is placed at any point inside of W, then assume any a translational scan orthogonal to world W that stops at a distance σ from any edge of W yields a correct partial world PW of W. Also assume given a translational scan entirely inside of a partial world PW_n derived from W yields a correct partial world PW_{n+1}. These assumptions are characteristic of the idealized sensor S_2.

World W - The world $W = \{P_o, H\}$ is an orthogonal world with holes as defined in Chapter VI. The world W has a finite number of holes h and the boundary polygon P_o of W has a finite number of edges. All holes $H = H_1, \ldots, H_h$ have a finite number of edges. All edges of W have finite length. The world is restricted to an orthogonal world due to the limited incidence angle capability of S_2. The minimum distance between the boundary polygon P_o and any hole polygon $H_i \in H$ must be greater than σ.

2. Example of Behavior

An example of sensor S_2 cartography is presented starting in Figure 7.17. In Figure 7.17 (a), R is placed at any arbitrary starting point C. R sweeps the S_2 about a full circle and extracts the edge list Q consisting of the four edges; e_a, e_b, e_c, and e_d. In any given

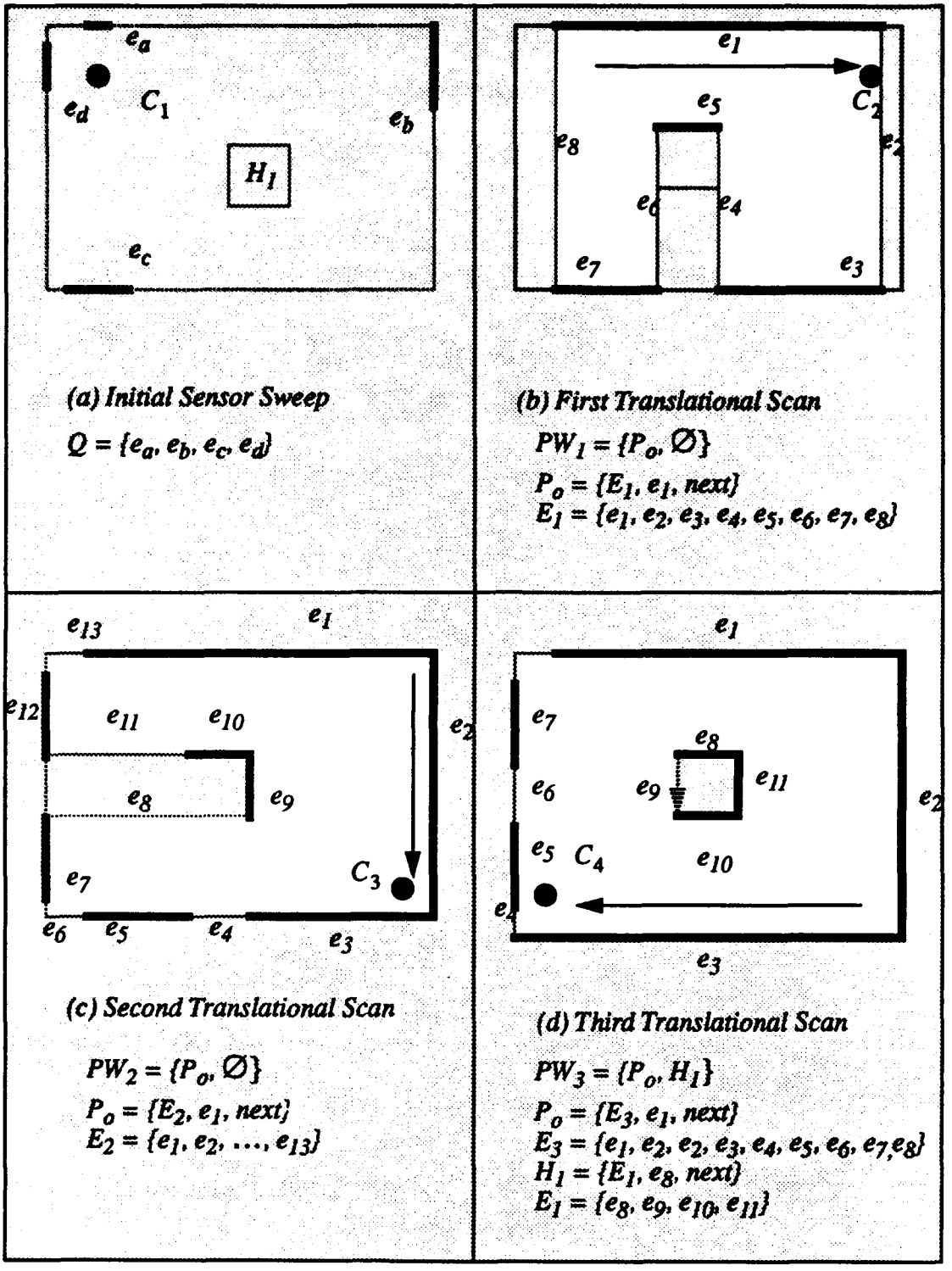


Figure 7.17 - S_2 Sensor Idealized Robot Cartography

world at least four edges are extracted by this full sweep. More than four edges are possible. These edges must meet the incidence angle criteria of $\pm\alpha$ to be visible as described above. The orientation of R's first translational scan is based upon the edge list Q such that R moves orthogonal with the world W. The position of the edges derived from the initial rotational sensor scan are used to determine the path for the first translational scan. The algorithm selects the edge furthest away in edge list Q to move toward for the first translational scan. In this case edge e_b is the furthest C_1. Then R moves from C_1 to C_2 using a straight line path. The first translational scan is shown in Figure 7.17 (b). Notice that a hole polygon H_1 was initially undetected by the S_2 sensor sweep due to the incidence angle limitations. Also notice that the partial world PW_1 extracted by the first translational scan is an orthogonal world.

The S_2 algorithm uses the resulting partial world to determine the path for the next translational scan. Since the right hand side of the first partial world is a long "inferred" edge e_2 in Figure 7.17 (b), R performs the second translational scan from configuration C_2 to C_3. The second translational scan reveals the right hand side of the boundary polygon of the world and the right hand side of the hole polygon H_1. Using the same reasoning process, R continues with the third translational scan from C_3 to C_4. More of the boundary and hole polygon edges are revealed. In Figure 7.17 (a), the final translational scan from C_4 to C_5 completes the map of the world. Notice that "inferred" edges near the four corners of the boundary polygon are incorporated in the "real" edges. The "inferred" edges in the interior corners of P_o are connected so that R maintains a standoff distance σ from any "real" edge in W.

3. Algorithm

The algorithm for the S_2 sensor cartography is listed in Figure 7.19. At the top level the algorithm first initializes all variables. Then a rotational sweep is performed to determine the visible edges of W. The algorithm then enters a "while loop" and iterates until the

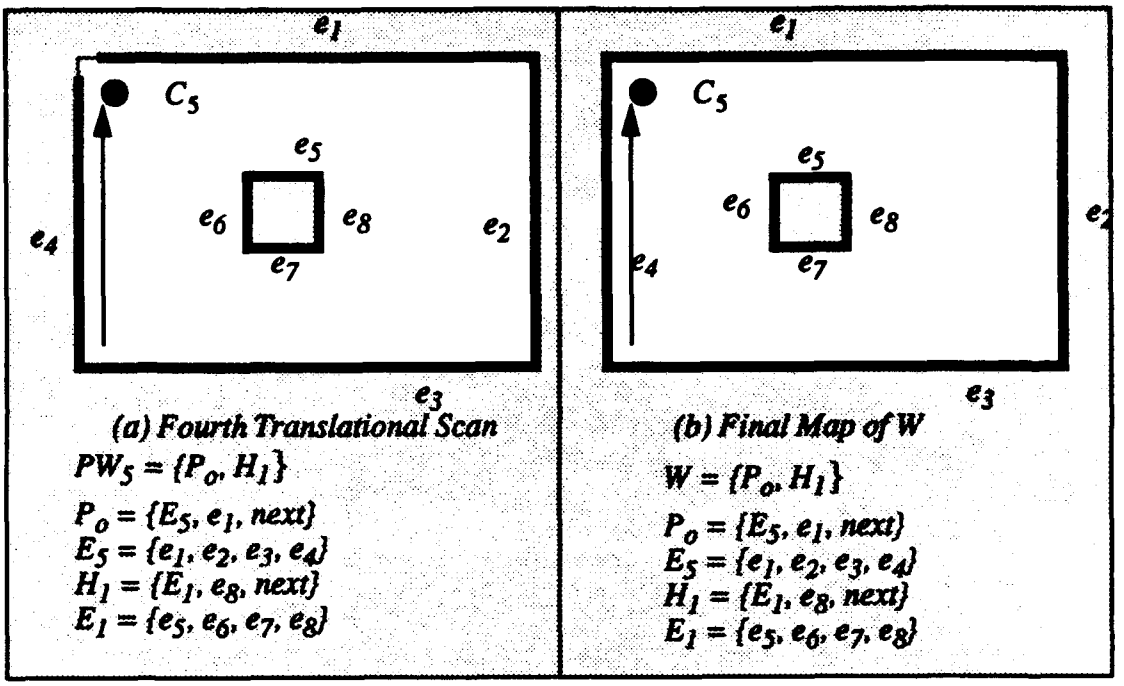


Figure 7.18 - S_2 Sensor Idealized Robot Cartography

partial map *PW* is evaluated as completed. There are six basic functions; *full_sweep*, *find_orthogonal_orientation*, *translational_scan*, *merge*, *next_position*, and *complete*.

The *full_sweep* function sweeps *R*'s S_2 sensor clockwise from 0 to 2π and extracts an edge list *E*. This edge list represents all "real", detectable edges from *R*'s position. In the example, the "real" extracted edge list in Figure 7.17 (a) is $Q = \{e_a, e_b, e_c, e_d\}$. This function always extracts at least four "real" edges since S_2 has infinite range and *W* is an orthogonal world. More than four edges are also possible depending on *R*'s initial position and the geometry of the world.

The *find_orthogonal_orientation* function aligns *R*'s internal coordinate system such that it is orthogonal to the edges in the input list *Q*. This function examines each edge in the edge list *Q* and returns a point *C* such that *R* moves orthogonal to the edges in *Q* when it moves on its first translational scan. This concept is illustrated in Figure 7.20. Edge e_b is the furthest edge, but *R* cannot move to e_b on a single straight line path orthogonal to the

```
S2_automated_cartography()
{
    World PW = (Ø, Ø);
    Edge_list Q = Ø;
    Configuration C = (0, 0, 0);

    Q = full_sweep(C);
    C = find_orthogonal_orientation(Q);
    while (not complete(PW))
    {
        Q = translational _scan(C, PW);
        merge(Q, PW);
        C = next_configuration(PW, C)
    } /* end while */
    return PW;
}
```

Figure 7.19 - S_2 Sensor Idealized Robot Cartography Algorithm

edges in W. Therefore, the function picks the next furthest edge e_a and calculates a point C at a distance σ from e_a as shown in Figure 7.20.

The *translational_scan* is an analog of the S_1 *move_to* function except that S_2 operates while R moves translationally. The reduced capability of the S_2 sensor requires R to use a translational scanning technique to find edges in W. The *translation_scan* function moves R on a straight line path from its current position to the C_n position. The path chosen is always orthogonal to W and runs parallel to one or more of W's edges. The *translation_scan* function operates while R is moving translationally along a straight line path element. During the translational scan, the S_2 sensor scans perpendicular to R's direction of travel on both the left and right sides. The *translational_scan* function returns an edge list Q.

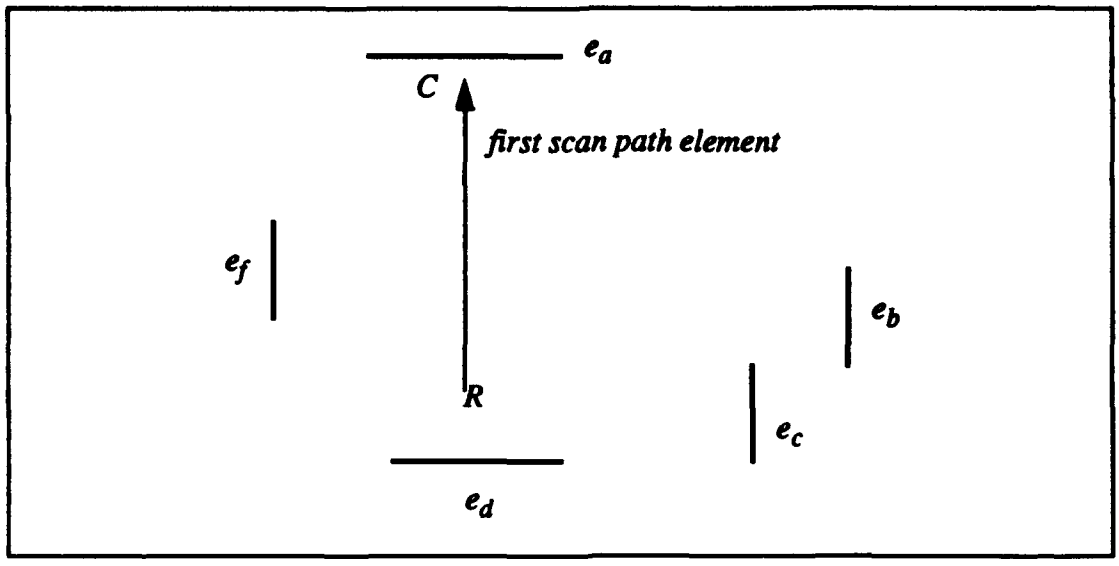


Figure 7.20 - The *find_orthogonal_orientation* function

The *merge* function combines an edge list Q extracted from a single translational scan with the existing partial world PW_n to form PW_{n+1}. The *merge* function must recognize and adapt to holes found in the world. In the example in Figure 7.17 (d), the *merge* function recognizes a hole polygon during the translational scan from C_3 to C_4. The "real" edges e_5, e_6, e_7, and e_8 and the *next* function for the polygon are modified to change the directionality of the hole edges such that $H_1 = \{\{e_5, e_6, e_7, e_8\}, e_5, next\}$.

The *next_position* function takes the existing partial world PW and R's position C_n as input parameters and returns the next point C_{n+1} for R to move to for translational scanning. This function examines all imaginary edges in the current partial world and selects the imaginary edge closest to R's current position. Then R calculates a point inside of PW_n such that R will move parallel to this imaginary edge an inside of PW_n. If there are no imaginary edges left in PW_n, then the *next_position* function returns R's current position.

The *complete* function evaluates PW_n. If any "inferred" edges longer than σ remain in PW, then PW is evaluated as incomplete and FALSE is returned. If $PW = (\varnothing, \varnothing)$,

then *PW* is also evaluated as incomplete and FALSE is returned. Otherwise, if all "inferred" edges in *PW* are shorter than σ, then *PW* is evaluated as completed and TRUE is returned.

4. Proof of Correctness and Termination

PROPOSITION 7.3: Let PW_n be a partial world obtained by the n^{th} translational scan of the world W of the S_2 idealized cartography algorithm listed in Figure 7.19. Then PW_n is a correct partial world derived from W.

Proof:

Basis: For translational scan number $n=1$, PW_1 is a correct partial world of W by the assumption that any translational scan in W yields a correct partial world.

Inductive Hypothesis: Assume for translational scan $n=m$ the proposition is correct.

Inductive Step: Then at $n=m+1$, R has a partial world PW_{m+1}. By the inductive hypothesis, the partial world PW_m is correct. Since PW_{m+1} is derived by merging the edges from a translational scan inside of PW_m, the partial world PW_{m+1} is a correct partial world derived from W. □

Given an arbitrary orthogonal world aligned in general position list the x-coordinates of all the vertical edges listed in ascending order is $x_1, x_2, \ldots, x_p$ and the y-coordinates of all of the horizontal edges listed in ascending order is $y_1, y_2, \ldots, y_q$. The minimum length translational scan length is defined by Equation 7.8.

$$l_1 \equiv min\,(min_{1 \le i \le p+1}\,(x_{i+1} - x_i), min_{1 \le j \le q+1}\,(y_{j+1} - y_j)) \qquad 7.8$$

LEMMA 7.3 : Given any orthogonal world W there exists a minimum translational scan length $l_1 \geq 0$ such that the length l of any translational scan in the world W for some n generated by the S_2 algorithm in Figure 7.19 is greater than or equal to l_1 ($l \geq l_1$).

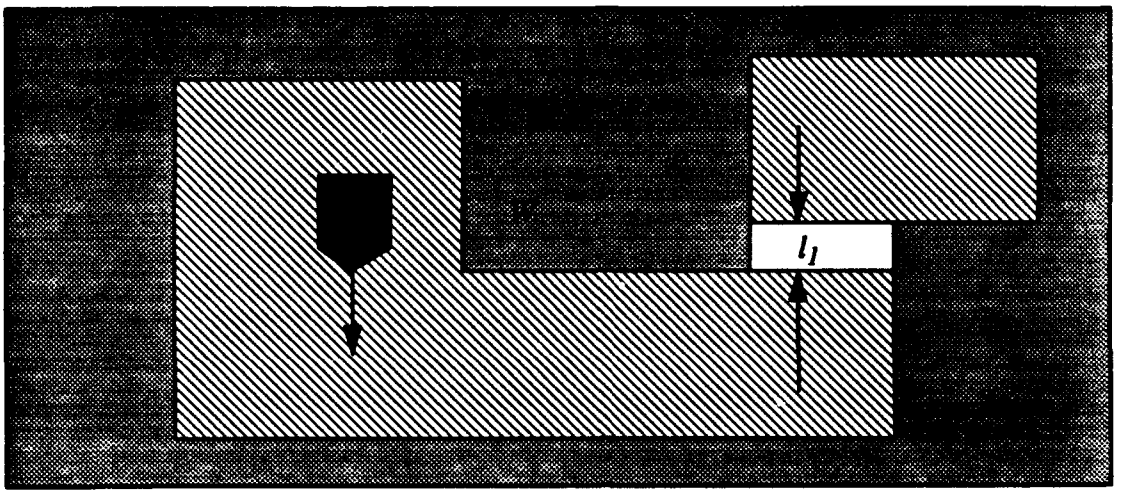


Figure 7.21 - A Minimum Length Translational Scan l_1

Proof: Since translational scans are either horizontal or vertical and since l_1 is the minimum distance between any two successive edges perpendicular to R's path, the minimum length scan is l_1 as illustrated in Figure 7.21. □

LEMMA 7.4 : For any translational scan by the S_2 algorithm in Figure 7.19 the total areas swept by the translational scan is greater than or equal to ${l_1}^2$.

Proof: The minimum length of any translational scan in a world W is l_1 by LEMMA 7.3. By the same token the minimum width of the area scanned is l_1. Therefore the minimum region swept by any translational scan must be greater than or equal to ${l_1}^2$. □

***PROPOSITION 7.4*:** The execution of the S_2 idealized automated cartography algorithm listed in Figure 7.19 terminates.

Proof: Let A represent the finite total area of the region enclosed by a world W. By LEMMA 7.2 the minimum area swept by any S_2 translational scan is ${l_1}^2$. Therefore the execution of the S_2 cartography algorithm terminates with at most $\left(\frac{A}{l_1^2}\right)$ iterations. □

Conclusion: Since any PW_{m+1} produced by the $m+1$ step of the algorithm is a correct partial world of W by Proposition 7.3 and since the algorithm terminates after a finite number of iterations by Proposition 7.4, the idealized sensor S_2 cartography algorithm returns a complete correct partial world derived from W.

C. ALGORITHM FOR IDEALIZED SENSOR S_3

The S_3 sensor is an idealized sensor with capabilities that are less ideal than S_1 or S_2. For this sensor, the range is limited to arbitrary value β. As with the S_2 algorithm this algorithm operates only on orthogonal worlds because S_3 has the same incidence angle limitation as S_2.

1. Assumptions for the S_3 Algorithm

Idealized Sensor S_3 - The sensor S_3 moves about in W attached to R. This perfect sensor S_3 is a modified, ray-tracing sensor with finite range. S_3 differs from S_1 in that S_3 returns only the portions of an edge that are incident $\pm\alpha$ with the sensor's ray. S_3 differs from S_1 and S_2 in that it is capable of limited range β. Therefore, S_3 operates primarily by translational scanning. Further, S_3 connects the discontinuities between edges on W by "inferred" edges and uses "inferred" edges to bound a region out of sensor range. These "inferred" edges bound the occluded regions inside of W that are not visible from the scan position. During translational scanning, S_3 scans both sides of R's path perpendicular to R's direction of travel. S_3 also monitors forward range to any obstacle. In Figure 7.22, R performs a translational scan from C_1 to C_2. The sensor S_3 has a side scanning beam on either side of R. In Figure 7.22, the object on R's right hand side is extracted as a "real" edge since its range is within the maximum range β from R. The left hand side object is beyond S_3's range, so S_3 constructs an "inferred" edge parallel to R's translational scan path. R stops at the point C_2 at a distance σ from the obstruction. The imaginary edges on either side of R are extended to include this barrier. The distance σ is an arbitrary, small standoff distance.

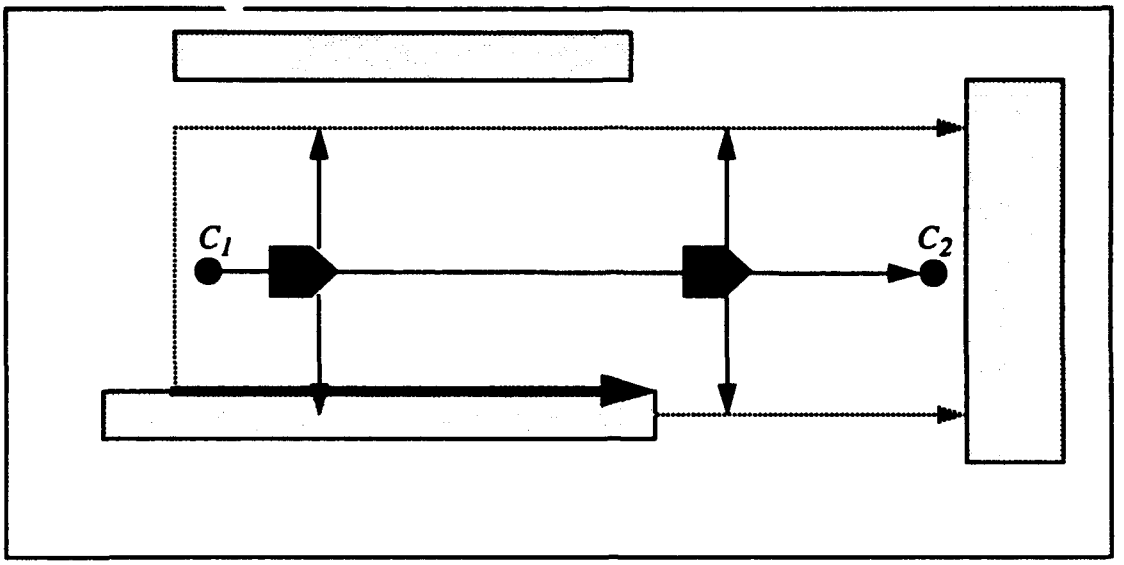


Figure 7.22 - - S_3 Translational Scanning

Suppose R is placed at any point inside of W, then assume any a translational scan orthogonal to world W that stops at a distance σ from W yields a correct partial world PW of W. Also assume given that a partial world PW_n is a correct partial world derived from W, a translational scan that moves R parallel to any existing "inferred" edge yields a correct partial world PW_{n+1}. These assumptions are characteristic of the idealized sensor S_3.

World W - The world $W = \{P_o, H\}$ is an orthogonal world as defined in Chapter VI. The world W has a finite number of holes and the boundary polygon P_o of W has a finite number of edges. All holes $H = H_1, \ldots, H_h$ have a finite number of edges. All edges of W have finite length.

2. Example of Behavior

The S_3 algorithm behavior is illustrated in Figure 7.23. R is initially placed at point C_1. An initial sensor sweep extracts an edge list Q with two edges, e_a and e_b, from the world. Notice that less information is obtained by a sensor sweep due to S_3's limited range and incidence angle capability as shown in Figure 7.23 (a). Based upon these two ini-

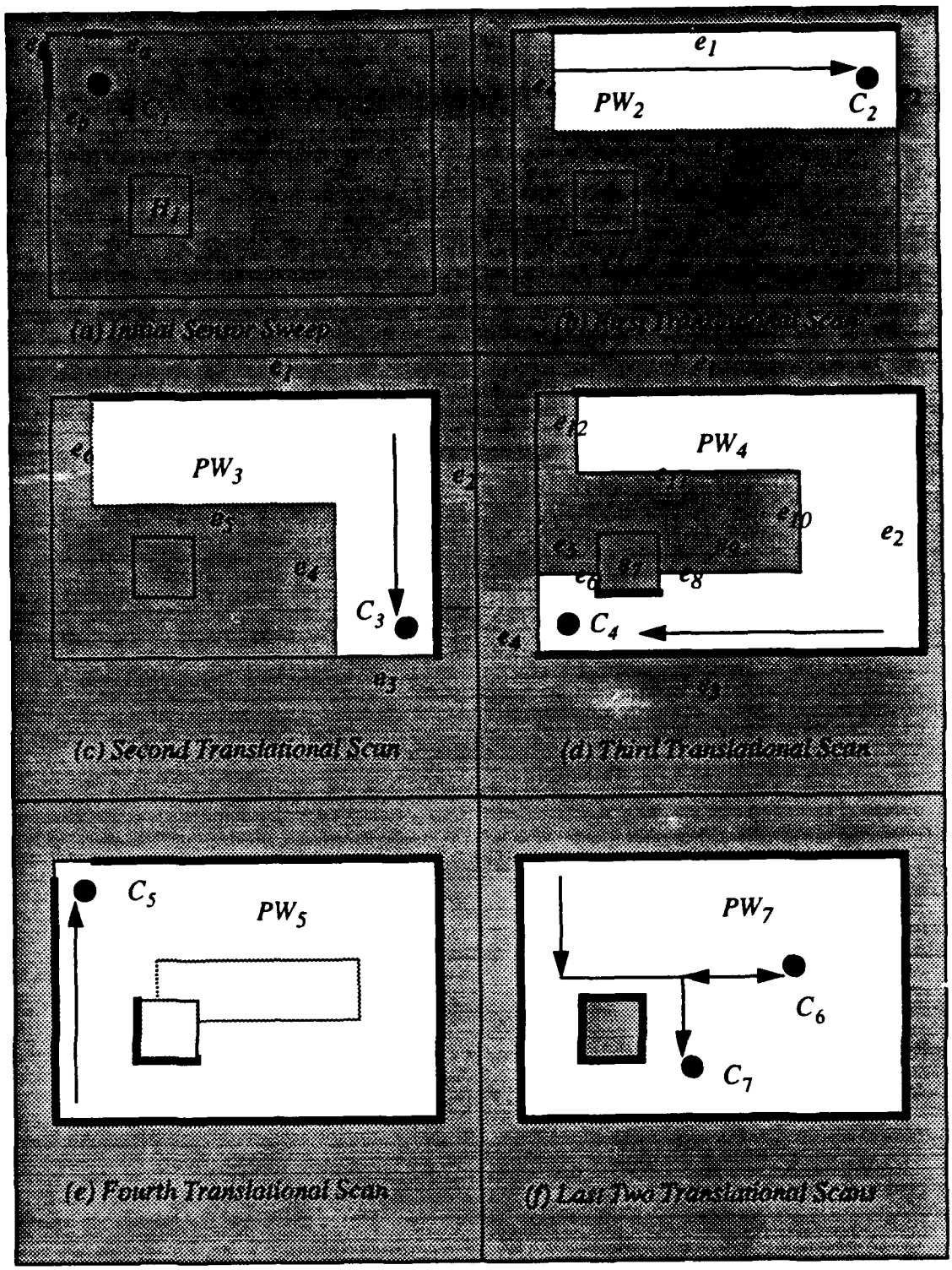


Figure 7.23 - The S_3 Cartography Example

tial edges, R moves parallel to e_a from point C_1 to C_2. This is the first translational scan represented by the white area in Figure 7.23 (b). Notice that the hole polygon H_1 is out of range of sensor S_3. The direction for the second translational scan is chosen based upon S_3's sensor input to R at point C_2. R detects an obstruction forward and to the left. Therefore, it turns right 90° for the second scan. The second translational scan moves R from point C_2 to C_3 as illustrated in Figure 7.23 (c). R stops moving when a barrier is sensed forward at a distance σ. The partial world PW_3 is now composed of two "real" edges and several "inferred" edges. These "inferred" edges require resolution by translational scan. R continues translational scanning by moving from C_3 to C_4 as illustrated in Figure 7.23 (d). Two "real" edges are extracted by this scan, one from the boundary polygon (e_3) and one on the hole polygon H_1 (e_7). R next moves from C_4 to C_5 for the fourth translational scan. This motion is determined by the proximity of the closest inferred edge. Once C_5 is reached, several inferred edges remain inside of the boundary polygon. R moves from C_5 to C_6 to C_7 to complete the map as illustrated in Figure 7.23 (f).

3. Algorithm

The algorithm for the S_3 sensor cartography is listed in Figure 7.24. At the top level the algorithm first initializes all variables. Then a rotational sweep is performed to determine the visible edges of W. The algorithm then enters a "while loop" and iterates until the partial map PW_n is evaluated as completed. There are six basic functions; *full_sweep, find_orthogonal_orientation, next_orientation, merge, translation_scan,* and *complete.*

The *full_sweep* function sweeps R's S_3 sensor through 360° for one circular scan and returns an edge list Q. All portions of W's "real" edges in sensor range that are visible, within range β and that have the correct incidence angle α are extracted. R's internal coordinate system is aligned to the visible edge of Q. In this way R aligns its coordinate system orthogonal to W.

```
S3_automated_cartography()
{
    World PW = (∅, ∅);
    Edge_list Q = ∅;
    Point C = (0, 0);
    Orientation γ;

    Q = full_sweep(C);
    γ = find_orthogonal_orientation(Q);
    while (not complete(PW))
    {
        Q = translational _scan(γ, PW);
        merge(Q, PW);
        γ = next_orientation(PW)
    } /* end while */
    return PW;
}
```

Figure 7.24 - S_3 Sensor Idealized Robot Cartography Algorithm

The *find_orthogonal_orientation* function takes the edge list Q as an input and rotates R to an orientation suitable for translational scanning. This function determines which of the visible edges in the edge list is best for guiding R's first translational scan. The *find_orthogonal_orientation* function is somewhat complicated by all of the possible configurations of edges extracted by the *sweep* function. The algorithm for the *find_orthogonal_orientation* function is shown in Figure 7.25. There are six important cases as illustrated in Figure 7.26. If the input edge list Q is NULL, then there were no visible edges after R's first stationary, circular sweep. In this case R performs a series of circle searches to locate any edges that may be nearby as shown in Figure 7.26 (a). The circles used for the edge search start out small and then grow progressively larger until an edge is located. The first edge

```
find_orthogonal_orientation(Q)
edge_list Q;
{
    if Q = ∅ then
        While (Q = ∅)
            Q = circle_for_edges();
        γ = rotate_parallel_to_one_edge(Q);
    else if (num_edges(Q) = 1)
        γ = rotate_parallel_to_one_edge(Q);
    else if (num_edges(Q) = 2 and edges_parallel(Q))
        γ = rotate_parallel_to_two_edges(Q);
    else if (num_edges(Q) = 2 and edges_perpendicular(Q))
        γ = rotate_to_point_out_of_corner(Q);
    else if (num_edges(Q) = 3)
        γ = rotate_towards_open_space(Q)
    else if (num_edges(Q) = 4)
        γ = rotate_toward_furthest_edge(Q);
    return γ
}
```

Figure 7.25 The *find_orthogonal_orientation* algorithm

found is used to determine the path element for the first translational scan using the *rotate_parallel _to_one_edge* function.

The *num_edges* function processes the edge list Q and determines how many separate edges are visible. This function does not count repeat edges, *i.e.* edges with the same orientation and position twice. When the function *num_edges* returns one, meaning that only one edge is visible and in range, R simply uses this single edge to guide the first translational scan as shown in Figure 7.26 (b). R rotates to align itself parallel to the edge's orientation and uses the extracted edge's position to calculate a suitable path element for the first translational scan.

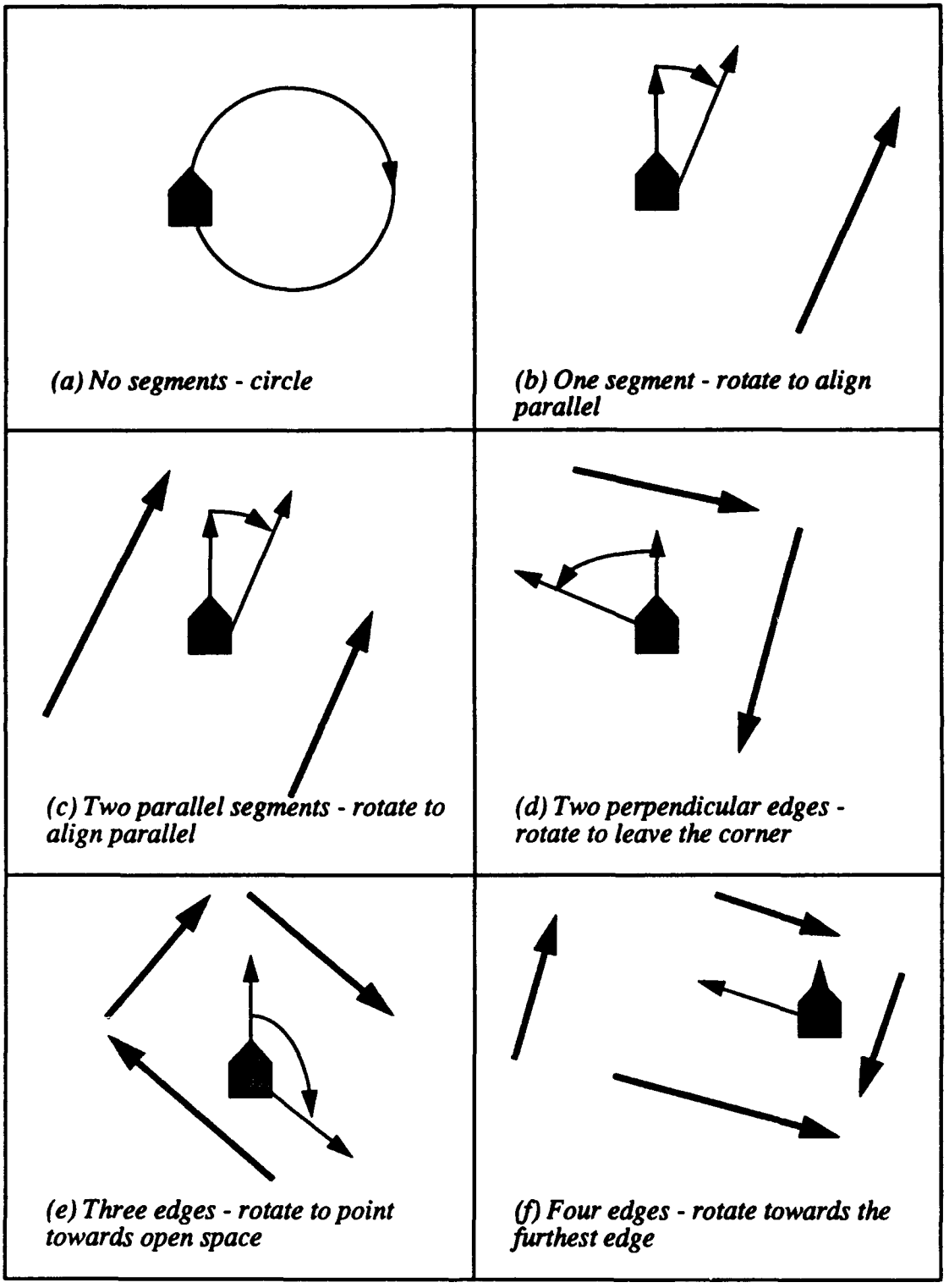


Figure 7.26 The *find_orthogonal_orientation* cases

The *edges_parallel* function examines the edge list Q and returns TRUE if all of the included edges are parallel and FALSE otherwise. If the *num_edges* function returns two and the *edges_parallel* function returns TRUE, then R rotates parallel to the edges in Q and uses the geometry of these edges to determine the best path element for the next translational scan. This case is as shown in Figure 7.26 (c). This behavior is appropriate for when R is starts in a hallway that is less than 2β wide.

In Figure 7.26 (d), R uses the position of the two perpendicular edges to rotate towards open space. This is appropriate when R starts near a corner in its world space. In Figure 7.26 (e) R uses the three edges to rotate towards open space and determine the best first scan path element for translational scanning. In Figure 7.26 (f), four edges are detected, in this case R simply rotates to point towards the edge that is furthest away. This behavior allows the first translational scan to map as much open space as possible.

The *translational_scan* function takes an orientation γ as an input and moves R along a path element with this orientation. Only four values of the input angle are possible 0, $\pi/2$, π, or $-\pi/2$. The reduced capability of the S_3 sensor requires R to use a translational scanning technique to find edges in W. The *translation_scan* function moves R on a straight line path from its current position C_n to the C_{n+1} position. The path chosen is always orthogonal to W based upon sensor input and therefore runs parallel to one or more of W's edges.

The *merge* function merges the extracted edge list Q with the current partial world PW_n to obtain a new partial world PW_{n+1}. The function recognizes hole emergence in PW_{n+1} by examining the existing edges in PW_n.

The *next_orientation* function takes the existing partial world PW as input and determines the path required to reach the next appropriate translational scan. This algorithm steers R parallel to the appropriate "inferred" edge in PW using a depth first search strategy.

The *complete* function evaluates the partial world PW. If $PW = (\varnothing, \varnothing)$ or PW contains at least one "inferred" edge longer than σ, the *complete* function returns FALSE. If all

"inferred" edges in PW are shorter than σ, then *complete* returns TRUE. If $PW \bullet P_o$ is complete and fully scanned and $PW \bullet H = \varnothing$ then PW is evaluated as complete.

4. Proof of Correctness and Termination

PROPOSITION 7.5: Let PW_n be a partial world obtained by the n^{th} translational scan of the world W of the S_3 idealized cartography algorithm listed in Figure 7.24. Then PW_n is a correct partial world derived from W.

Proof:

Basis: For sensor sweep number $n=1$, PW_1 is a correct partial world of W by the S_3 assumption that any translational scan yields a correct partial world.

Inductive Hypothesis: Assume for translational scan $n=m$ the proposition is correct.

Inductive Step: Then at $n=m+1$, R has a partial world PW_{m+1}. By the inductive hypothesis, the partial world PW_m is correct. By the S_3 assumptions, the partial world PW_{m+1} is a correct partial world derived from W. □

There exists a minimum width ($l_2 > 0$) for any translational scan. The value of l_2 is a positive constant less than 2β for any given world W. Given an arbitrary orthogonal world in general position, the x coordinates of all of the vertical edges in ascending order is $x_1, x_2, \ldots, x_p$ and list the y-coordinates of all of the horizontal edges in ascending order $y_1, y_2, \ldots, y_q$. The minimum width of a translational scan (l_2) is defined by Equation 7.9.

$$l_2 \equiv min\left(min_{1 \le i \le p+1}\left(x_{i+1} - x_i - 2m\beta\right), min_{1 \le j \le q+1}\left(y_{j+1} - y_j - 2m\beta\right)\right) \qquad 7.9$$

where $(x_{i+1} - x_i - 2m\beta) > 0$ and $(y_{i+1} - y_i - 2m\beta) > 0$ for non-negative integer values of m.

LEMMA 7.5 : The minimum area swept by an S_3 translational scan is $l_1 \times l_2$.

Proof: Since LEMMA 7.2 states that the minimum length of a translational scan is l_1 for a given world W and since the minimum width of an S_3 translational scan is defined as l_2, the minimum areas swept by a single translational scan is $l_1 \times l_2$. □

***PROPOSITION 7.6*:** The execution of the idealized automated cartography algorithm listed in Figure 7.24 terminates.

Proof: Let A represent the finite total area of the region enclosed by the world W. By LEMMA 7.5 the minimum area swept by any translational scan is $l_1 \times l_2$. The execution of the S_3 cartography algorithm in Figure 7.24 terminates with at most $(\frac{A}{l_1 \times l_2})$ translational scans. □

Conclusion: Since any PW_{m+1} produced by the $m+1$ step of the S_3 algorithm is a correct partial map of W and since the algorithm terminates after a finite number of iterations, the idealized sensor S_3 cartography algorithm gives a correct complete world derived from W.

D. SUMMARY

A series of three algorithms for automated cartography for an idealized robot with progressively less idealized sensors are presented. Examples provide an illustration of the idea behind each of the algorithms. The S_1 sensor is capable of cartography of non-orthogonal worlds since this sensor can extract information from the world regardless of incident angle. The S_2 sensor is an infinite range sensor but with limited incidence angle capability. This limited incidence angle capability restricts the S_2 sensor to cartography of only orthogonal worlds. The S_3 has both limited range and limited incidence angle capability. This sensor is most like the "real" sensors on Yamabico. Therefore, the S_3 algorithm provides the basis for the "real" cartography algorithm developed in Chapter VIII.

All component parts of the three algorithms are explained. The algorithms are each proven by induction on the number of iterations. The total number of "real" edges in the

partial world grows with each iteration of the algorithm until the world W is completely mapped. The odometry error incurred by robot motion and the "real" robot's non-holonomic motion are the non-idealized constraints imposed by R. Finite sensor range, sensor noise, and sensor return limited by sensor beam incidence angle impose "real" constraints on the idealized sensor. These "real" world constraints are investigated in Chapter VIII.

VIII. AUTOMATED CARTOGRAPHY BY YAMABICO-11

In Chapter II, numerous approaches to robot navigation and automated cartography were presented. None of these systems achieved simultaneous robot localization and cartography. Both localization and cartography must be performed at the same time for a robot to build an accurate, spatially-consistent map of an interior space. This system supports simultaneous robot cartography and localization since Yamabico can perform dead reckoning error corrections during translational scans orthogonal to the robot's world space.

In Chapter VII a series of algorithms for idealized automated cartography are presented. These algorithms have no practical utility in the real world since physical principles preclude their implementation. Idealized robot cartography provides a theoretical springboard for the development of the real sensor automated cartography on Yamabico-11. This chapter describes the physical characteristics of a real robot and a real sensor that cause the deviation from the idealized case. The real sensor cartography algorithm is explained in detail in this chapter. The experimental results of real sensor cartography appear in Chapter IX.

Important differences exist between the idealized robot-idealized sensor pair and the real robot with an array of real sensors. In Chapter VII, the idealized robot R moves about in the world space without incurring any dead reckoning error. Therefore, R always has perfect knowledge of its current configuration. Yamabico, on the other hand, cannot move an appreciable distance without accumulating some dead reckoning error. The reasons for this fact are explained in Chapter V. Yamabico's dead reckoning error increases as a function of the total distance traveled since the last odometry reset. Other factors affect the rate of dead reckoning error buildup per unit distance traveled; they are floor smoothness (related to integrated distance traveled), amount of turning the robot does per unit distance (related to wheel slip), and the robot's speed.

Concerning sensor capabilities, the idealized sensors in Chapter VII have progressively more real limitations. The practical range of a real active sensor is limited by spreading

losses and by sampling rate as discussed in Chapter II. For Yamabico's ultrasonic range finders, the practical limit on the range is about four meters for target detection and about two meters for reliable edge extraction. As discussed in Chapter II, there is a trade-off between sonar range gate and the data rate for range returns.

The S_1 sensor extracted edges from the world space regardless of sensor beam incidence angle. In practice, an active emitter sensor has a limit on incidence angle for range returns. Yamabico's ultrasonic range finders have a incidence angle limitation that varies with the target distance and target material. For a close, strongly reflective target sonar is most like the idealized sensor S_1.

All idealized sensors presented in Chapter VII give returns independent of target material and range. None of the idealized sensors in Chapter VII suffered from specular reflections. With idealized sensors, only the primary sensor beam was reflected by the target surface. For the real sensor, secondary or so-called "specular" reflections further complicate the analysis of the sonar data [Leonard 91]. The first reflection of the sensor beam is not always the range returned by a real sonar. This is particularly true when the sonar is aimed into a concave corner. The cartography algorithm presented in this chapter works within the physical limitations of the real robot and the real sensor.

A. REPRESENTATION OF THE WORLD

This section examines the physical limitations of Yamabico's locomotion system and sensors that cause them to be non-idealized. This explanation is important to bridge the gap between the real and idealized robot.

The data structure used for the map representation must efficiently store the current 2D map of the world in a compact data structure. This requirement is based upon the limited space in Yamabico's on board memory. Additionally, a smaller data structure can be more quickly searched. The data structure chosen must be a dynamic-type data structure since the number of features is not known beforehand. Grid-based map storage schemes use a fixed size storage area that cannot be grown dynamically. The type of structure allows for

the inclusion of additional edges as Yamabico builds a map of the world. Features can be added to the map until Yamabico runs out of memory. Yamabico-11 has five megabytes of main memory with a planned upgrade to 16 megabytes. Therefore, a map with a considerable number of features can be stored in Yamabico's main memory since each partial world edge requires only 32 bytes of storage space. If a grid-based scheme with a 10 cm grid size is used a byte of memory is used to store each square, 32 bytes could only store a region 0.32 m^2.

Since the map is used for navigation, it must be quickly processed as a part of the robot's spatial reasoning tasks. A detailed map requiring a small amount of storage space may be quickly processed using Yamabico's on board computing resources. One example is the determination of the next scan path for translational scanning. Also a quick search of the current partial world is frequently required to match newly detected features against existing features to determine if the new feature is a repeat of a previously detected surface or the first time this particular object has been detected. This gives the system the capability to recognize the difference between a new line segment and one that is already part of the map.

Lastly, the map must be transferred back to the host computer to allow for human inspection. This is required for debugging and improving the system. This feature also allows the map to be saved and used later on a subsequent robot missions.

The partial world data structure is the means of map representation in this dissertation. The concepts were introduced in Chapter VI. The partial world data structure is implemented as a doubly linked list of typed edges derived from sonar data and refined by repeated robot scans. Edges are typed as either "real" or "inferred" depending on how they are derived from sonar data. "Real" edges are derived from line segments extracted from sonar linear fitting data. "Inferred" line segments are constructed based upon the geometry of the existing "real" line segments in the *PW*. They serve to bound the unexplored area of the world.

1. Real World Issues

A fundamental problem with the Yamabico's locomotion system lies with regard to its dead reckoning capability. No matter how finely tuned the wheel encoders and dead reckoning algorithm, the robot odometry estimate drifts as a function of the distance traveled. The mathematical principles behind robot odometry are examined in Chapter V. A real robot must use sensor input to correct odometry drift. This is necessary since all sensor input for cartography is recorded with reference to the robot's current odometry configuration. Therefore, the map built by Yamabico is only as accurate as the estimate of its actual configuration in the world space.

In order to explore its entire world space, Yamabico must move to correctly position its sonar array for translational scanning. This motion incurs some dead reckoning error. Yamabico uses a straight line wall assumption with regard to its world space. This assumption allows some dead reckoning errors to be corrected during translational scanning. Using sonar range values, Yamabico's orientation and distance to a long, straight world edge can be corrected. The details of odometry correction during translational scanning are discussed in section B of this chapter and the experimental results are described in Chapter IX.

Given a partial world and sensor input, a robot can match features from the map to sensor input to correct odometry error. Extensive background research on robot localization is examined in Chapter II. In every case previously described, the world map used for localization was derived from *a priori* input. The automated cartography algorithm uses only a sensor derived partial world and indoor building heuristics (orthogonal world assumption) to correct odometry error. Specifically, Yamabico takes advantage of straight walls to correct some dead reckoning error.

Real sensor limitations also force real robot cartography to deviate from idealized cartography. The sensor chosen for this work is the ultrasonic range finder. The ultrasonic range finder is widely used in mobile robots and is readily available. Additionally, the author's work builds upon a considerable library of sonar processing functions already in-

cluded in the MML system [Sherfey 91]. The dissertation test bed robot, Yamabico-11, employs 12 ultrasonic rage-finders as its primary sensor as described in Chapter III. The differences between the abstract idealized sensor described in Chapter VII and the real sensor are described here.

Idealized sensor S_1 and S_2 are capable of providing data regardless of target range. This is not true for the real sensor. All real active sensors have some practical range limitations for a variety of physical reasons. For ultrasonic range finders this range limitation in air is typically about four meters. The range limit is due primarily to spreading losses and attenuation by the propagating medium. This range limit suggests that inferred edges become more important for real robot cartography since the robot's sensor cannot scan as much area as the idealized sensor.

The idealized sensor S_1 provides the robot with input data regardless of the sensor beam incidence angle. This is not true for real ultrasonic range finders as discussed in Chapter III. In Figure 3.3, the real sensor incidence angle for a valid range return varies with target range. In most cases, the sensor axis must be nearly normal to the target surface in order to obtain a valid range return. The real sensor is more ideal at closer range because spreading losses and attenuation are smaller. The incidence angle of $\alpha = \pm 10$ degrees from the normal to the surface typically gives a valid return when range is between $\beta_{min} = 9.3$ to $\beta_{max} = 200.0$ centimeters. The real sonar has a minimum as well as a maximum range value. The minimum range of the sensor was not considered at all for the idealized sensors in Chapter VII. The minimum range value of Yamabico's sonar system is explained in Chapter III.

The final limitation of the real sensor is specular reflections. In many cases, sonar returns the range to the second or third reflecting surface as the range value. This is caused by the fact that a smooth target surface acts as an acoustic mirror. These specularities are typically non-orthogonal line segments in the case of Yamabico-11. Several attempts to fully model specular reflection of sonar range finders have been made [Leonard 91][Kuc 91], but a complete model does not yet exist. Specular returns typically form short line segments

that are not orthogonal to Yamabico's world space. The automated cartography algorithm filters out these bad segments as Yamabico builds a map of its world space.

2. Definitions

World - The definitions of vertex, edge, polygon, orthogonal polygon, world and partial world are unchanged from the definitions given in Chapter VI. A world is a portion of a single floor of an office building.

Yamabico - The robot is the Yamabico-11 mobile robot as described in Chapter III. Yamabico is a non-holonomic robot capable of stationary rotation and translational motion. Robot rotation may be clockwise or counterclockwise. The translational motion is the means of path tracking of straight line segments, circular arcs, cubic spirals and parabolic line segments. Only straight line and circular arc path elements are used for Yamabico's cartography algorithm.

Sensors - The robot's sonar array *S* is treated as an abstract entity in this chapter. *S* is capable of range-finding forward, extracting edges while rotating, and translational side scanning with edge extraction. The Yamabico has an array 12 ultrasonic range finders as shown in Figure 3.2. For the purposes of cartography four sensors are used. Two forward looking sensors (numbers 0 and 3) and one on each side (numbers 4 and 7).

B. THE ALGORITHM

1. Assumptions

Real Robot - The real mobile robot Yamabico-11 measures 54 centimeters square and is 95 centimeters high. It is a power-wheel-drive robot capable of translational motion as well as stationary rotational motion. Yamabico has a nominal velocity of +30.0 centimeters per second but is capable of any velocity between -60.0 and +60.0 centimeters per second.

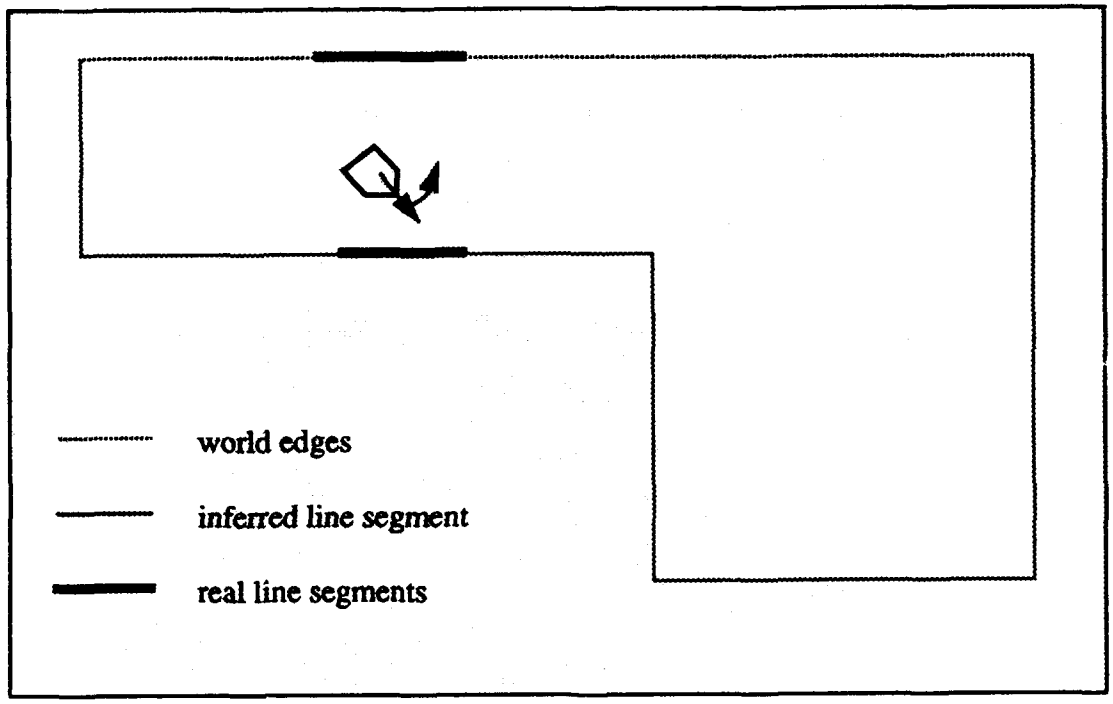
Sensors - These sensors are assumed to have an effective range $\beta_{min} = 9.3$ to $\beta_{max} = 200.0$ centimeters and provide a return if the sonar beam incidence angle is $\alpha = \pm 10°$ of

the normal to the target surface. This assumption is based upon the experimental data plotted in Figure 3.3. The sonar sensors extract data in the form of line segments representing surfaces of the world W. Line segment data closely orthogonal to Yamabico's current partial world that are derived from greater than 10 sonar returns are used for input to cartography. All other line segments data is discarded as specular reflections. The value of $\sigma = 75.0$ cm is used since $\sigma = 54.0 + 9.3 + 9.3 \cong 75.0$ cm. The value of σ represents the minimum opening through which Yamabico can safely navigate. All sonar returns at a range greater than 200.0 cm are discarded since S is only effectively finds edges up to this range.

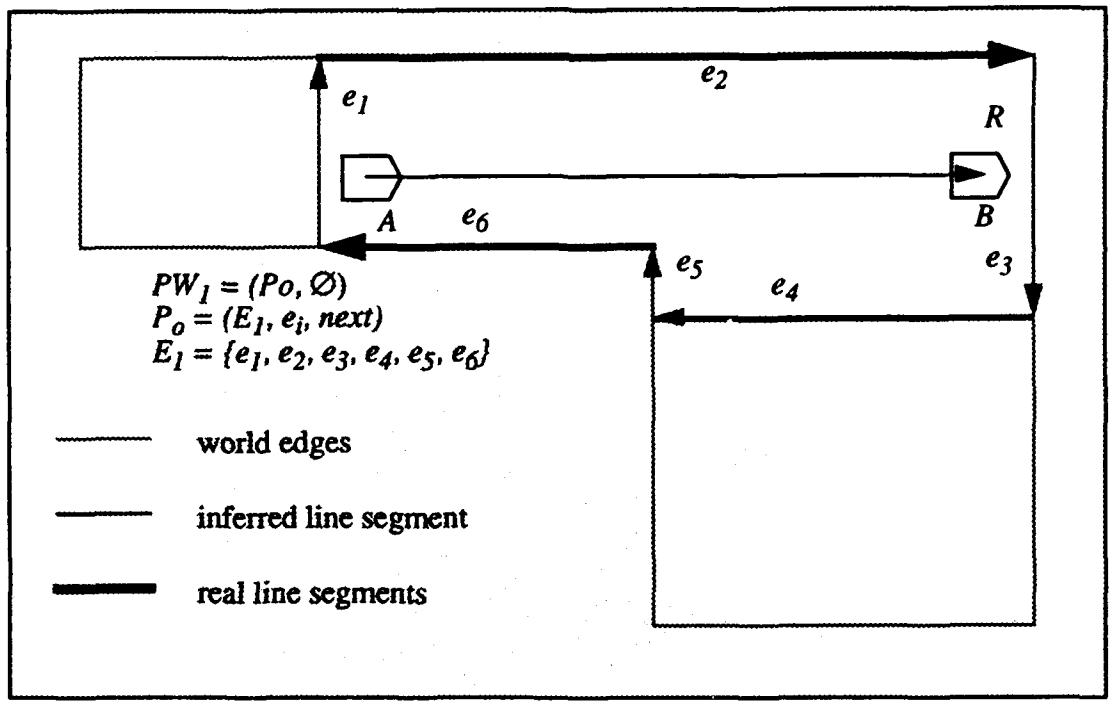
World - The world $W = \{Po, H\}$ is an orthogonal world as defined in Chapter VI. The world has a finite number of holes. The boundary polygon P_0 and all holes $H_1, \ldots, H_h$ have a finite number of edges. All edges in W have finite length. All world surfaces have a target strength greater than 50% at 2.3 meters range in accordance with Figure 3.6.

2. Example of Behavior

A right wall following method of cartography with left turns at obstacles is used by Yamabico to explore its world space. A simple example of the real automated cartography is illustrated starting with Figure 8.1 (a). The world W is a six-sided, L-shaped boundary polygon with no hole polygons. Yamabico is first placed in an arbitrary configuration such that no sensor is less than minimum range from any edge of W. In Figure 8.1 (a), Yamabico rotates counterclockwise to align itself with the closest edge extracted from the rotational scan. Then Yamabico moves translationally until the forward sensor encounters an obstacle forward as illustrated in Figure 8.1 (b). The translational scan ends σ from the obstructing boundary. Yamabico now constructs its first partial world from sonar data. $PW_1 = (P_0, \varnothing)$, with $P_0 = (E_1, e_1, next)$ where $E_1 = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. Edges e_2 and e_6 are "real" and e_1, e_3, e_4, and e_5 are "inferred". Edges e_1 and e_3 are constructed perpendicular to Yamabico translational scan at a distance σ from the endpoints of the translational scan. Edge e_4 is an "inferred" edge constructed to bound the region beyond sensor range. Edge

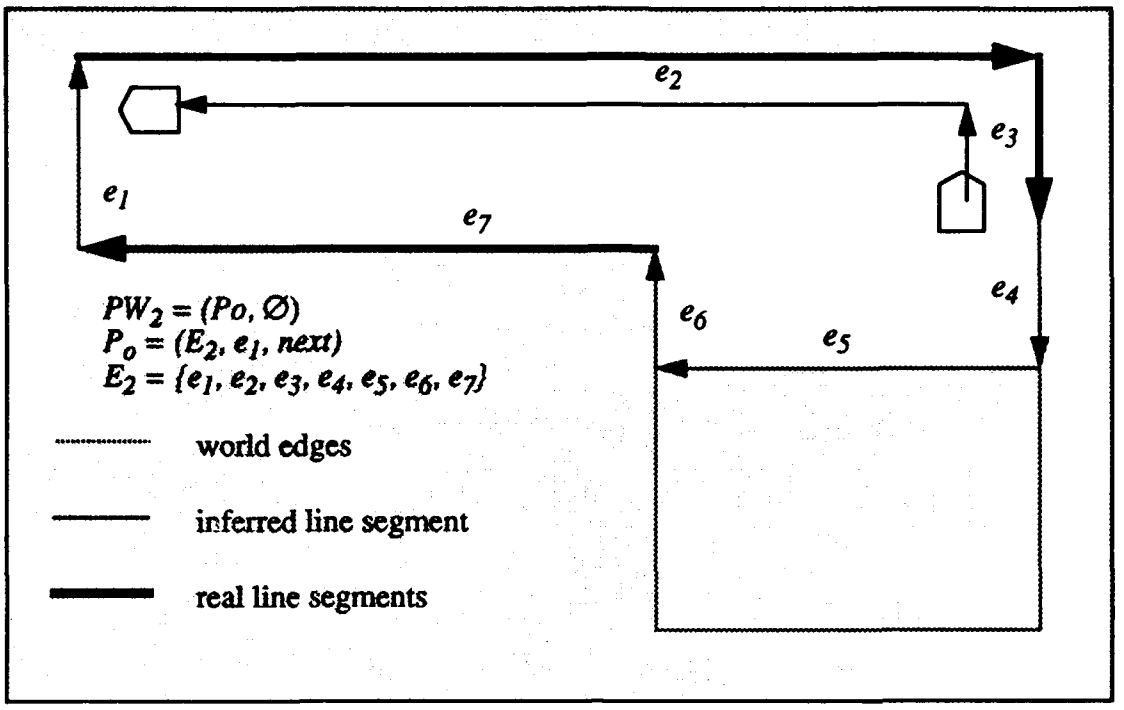


(a) Rotation to Identify Segments

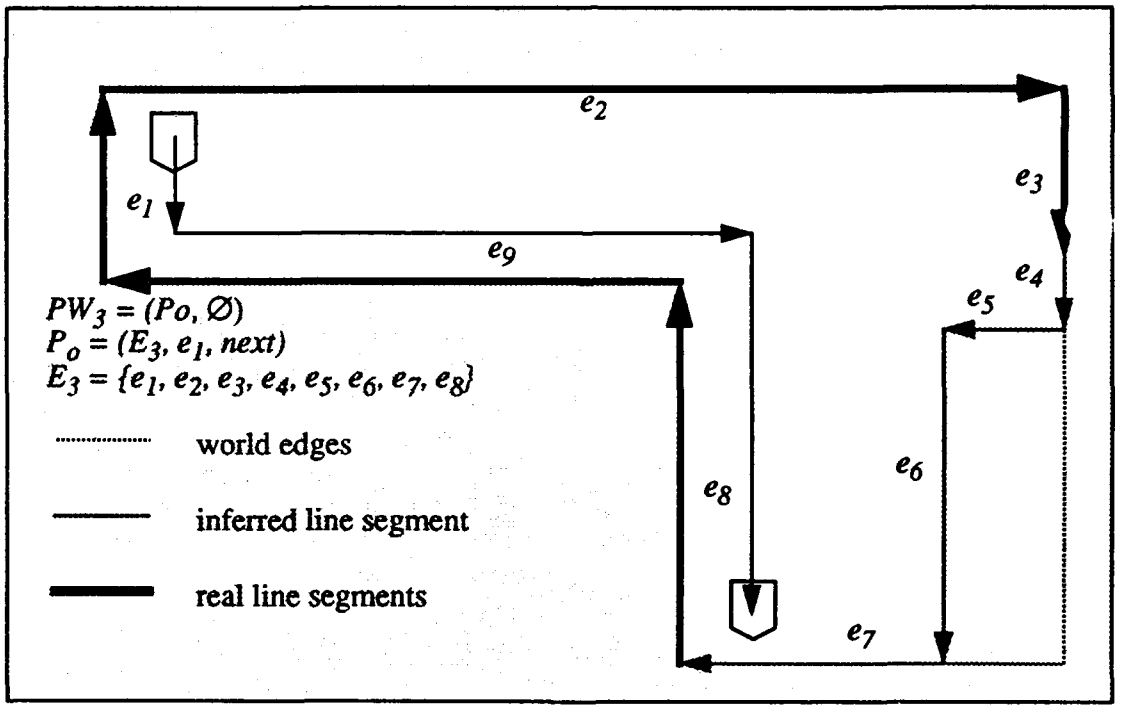


(b) First Translational Scan

Figure 8.1 Example of Yamabico Automated Cartography



(a) Second and Third Translational Scan



(b) Scans to Explore Inferred Edges

Figure 8.2 - Example of Automated Cartography

e_4 is constructed two meters to the right of the translational scan. Edge e_5 is an "inferred" edge constructed to connect the discontinuity between e_4 and e_6.

In Figure 8.2 (a), Yamabico turns left and scans "inferred" edge e_3 and then turns left and investigates "inferred" edge e_1. The partial world is now $PW_2 = (P_0, \varnothing)$, with $P_0 = (E_2, e_1, next)$ where $E_2 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$. Edges e_2, e_3, and e_7 are "real" and edges e_1, e_4, e_5, and e_6 are "inferred".

In Figure 8.2 (b), Yamabico turns left and performs a translational scan on edge e_1, then the *next_scan_config* function plans a path to investigate edge e_5 in Figure 8.1 (b). The resulting partial world is $PW_3 = (P_0, \varnothing)$, with $P_0 = (E_3, e_1, next)$ where $E_3 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9\}$. Only four "inferred" edges remain; e_4, e_5, e_6, and e_7 in Figure 8.1 (b), the rest of the edges are "real".

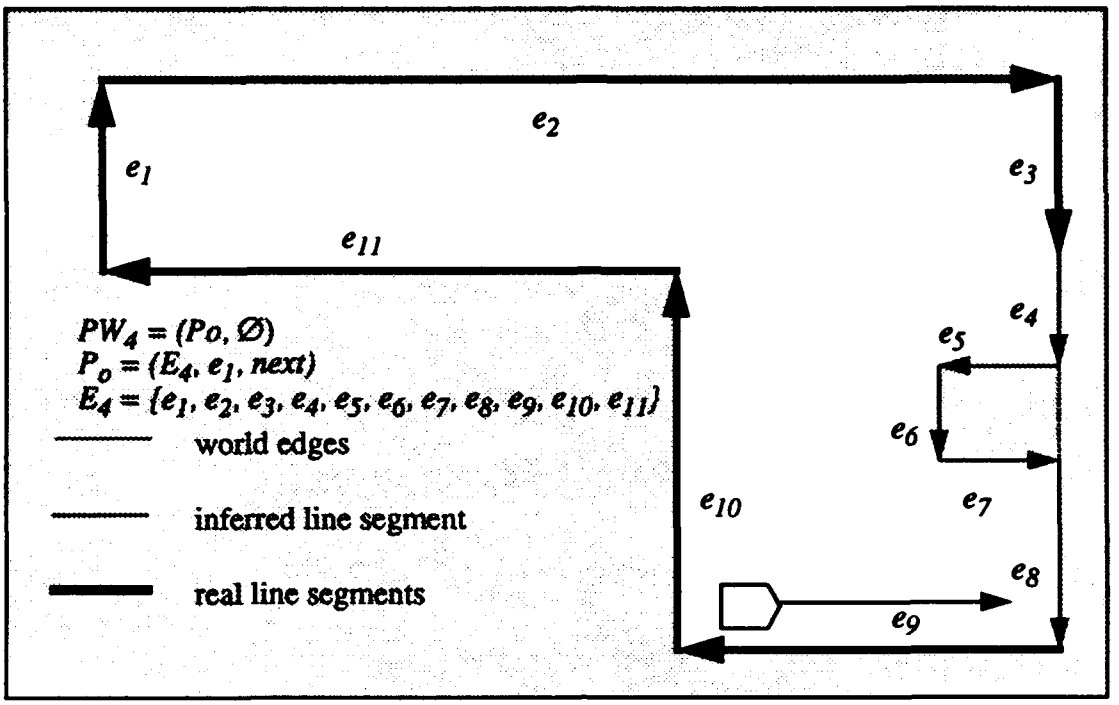
In Figure 8.3 (a), Yamabico turns left and scans "inferred" edge e_7 of PW_3. The new partial world is $PW_4 = (P_0, \varnothing)$, with $P_0 = (E_4, e_1, next)$ where $E_4 = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$. The remaining "inferred" edges are e_4, e_5, e_6, e_7, and e_8.

In Figure 8.3 (b), Yamabico turns left again and scans parallel to "inferred" edge e_8. This scan moves Yamabico through the unexplored region bounded by edges e_5, e_6, and e_7 in PW_4. The resulting partial world $PW_6 = W$ since there are no remaining "inferred" edges. $PW_6 = (P_0, \varnothing)$, with $P_0 = (E_6, e_1, next)$ where $E_6 = \{e_1, e_2, e_3, e_4, e_5, e_6\}$.

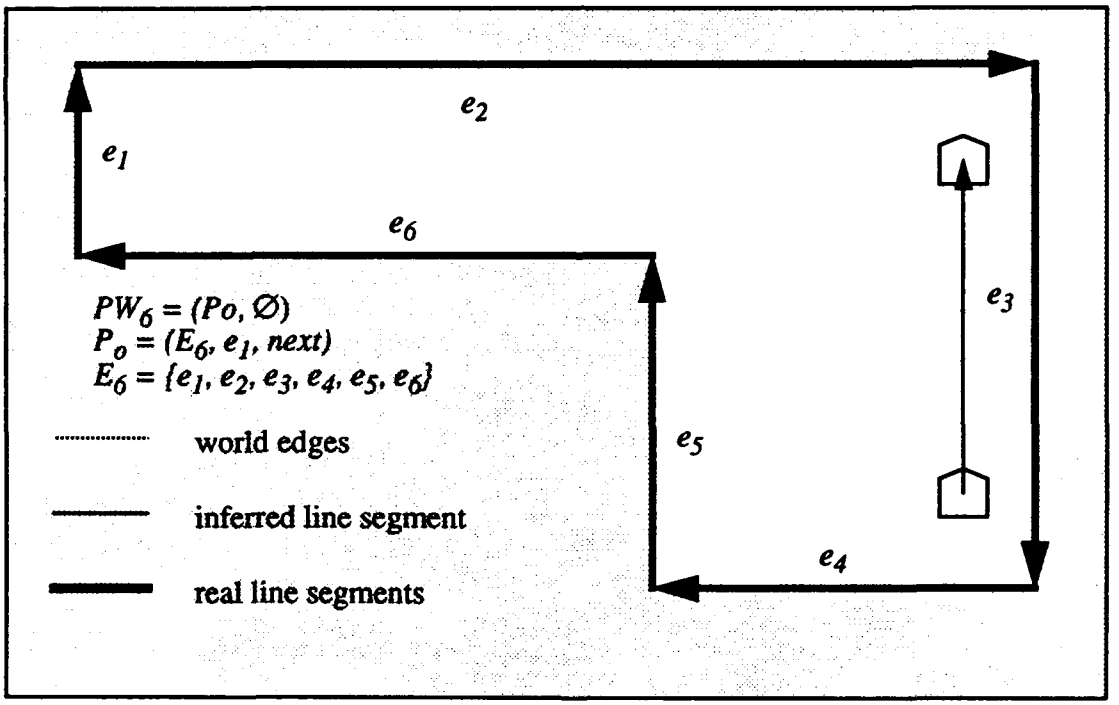
3. Algorithm

The global spatial learning algorithm is designed to allow Yamabico to intelligently move about in a given world space and build a model of its environment. The algorithm is shown in Figure 8.4. This is a top down, global overview of the algorithm. The purpose of each subroutine is explained in the remaining parts of this section.

The *initialize* function initializes Yamabico's sensor system for cartography. The appropriate sensors are also enabled. Linear fitting and data logging for the enabled sensors are enabled for the side scanning sonars. Since the nominal robot speed of 30.0 centimeters



(a) Left Turn to Examine an Inferred Edge



(b) Last Translational Scan

Figure 8.3 - Example of Yamabico Automated Cartography

per second is too high for cartography, the proper robot speed is set. Files for storing sonar data and the robot's location trace data are set up and initialized.

The *find_orthogonal_orientation* function orients the robot with respect to the nearest flat surface in the world space. Since Yamabico starts with no prior knowledge of the world, an initial rotational scan is required. This rotational scan extracts visible portions of the detectable edges of W and provides sufficient information to determine the best path element for the first translational scan. The same rules that were used for the S_3 version of the algorithm are used in this algorithm. Figure 8.1 (a) shows a robot placed in an arbitrary configuration in a simple closed workspace. The *find_orthogonal_orientation* function commands Yamabico to slowly rotate 360°. Sonar scans the world during the rotation. The extracted line segments are examined to determine the proper initial orientation of the robot and to chose an appropriate path element for the first translational scan. All of the extracted line segments are evaluated and the orientation of the closest line segment of sufficient

```
spatial_learn()
{
    CONFIGURATION C = (0, 0, 0, 0);
    Partial_World *PW = (∅, ∅);
    Path_List *PL = ∅;

    initialize(PW);
    find_orthogonal_orientation(&C);
    while(not complete(PW))
    {
        translational_scanning(&PL, &C, &PW);
        PL = next_scan_config(&PW, &C);
    } /* end while */
    return PW;
} /* end spatial_learn() */
```

Figure 8.4 Yamabico Cartography Algorithm

length is adopted as the robot's initial orientation. This function returns a path element suitable for the first translational scan.

Next the algorithm enters a "while loop". The *complete* function evaluates all edges of the current *PW*. This function checks the *PW* for spatial consistency and for the absence of "inferred" edges. A NULL partial world or one with any "inferred" edge greater than σ in length is considered incomplete. If the boundary polygon P_o consists of all "real" edges and the hole list is NULL, then the world is evaluated as complete. If the P_o has all "real" edges and all of the hole polygons on the hole list $H_i \in H$ have all "real" edges, then the *PW* is considered complete. The function returns *TRUE* if the *PW* is complete or returns *FALSE* if more scanning is required.

The *translational_scan* function moves Yamabico on a straight line path element parallel to the edge identified by the *find_orthogonal_orientation* function or the *next_scan_config* function. Yamabico continues the translational scan until one of the two forward looking sensors detects an obstacle. Following the translational scan, all sonar extracted line segments from the side looking sonars are merged into the partial world model (*PW*). The *PW* constitutes a free space model of the area Yamabico has just scanned during a single straight line translation scan. Figure 8.1 (b) shows a *PW* derived from sonar data from Yamabico's scan from point *A* to point *B*.

During the automated cartography, dead reckoning errors accrue due to Yamabico's motion. The automated cartography algorithm corrects this error in two ways; (1) wall following dead reckoning error correction and, (2) automated landmark selection and correction during DFS backtracking.

The wall following dead reckoning error correction operates under the assumption that the world space contains some long straight edges. When wall following is started, a path element parallel to the wall and at a distance less than β from the wall is calculated. Yamabico follows this path element and corrects some dead reckoning error using linear fitting sonar data extracted by scanning the wall. Yamabico's distance to the wall and its

orientation with respect to the wall are periodically corrected using the odometry correction algebra presented in Chapter V.

During world space exploration Yamabico records landmarks suitable for odometry correction. A good landmark is characterized by a detectable discontinuity in a world space edge. During the DFS backtracking process, these landmarks may be used for full odometry correction. Unfortunately, an automatically recognized landmark has a configuration error tied to Yamabico's dead reckoning error at the time it is recorded. Therefore, a dead reckoning error correction using an automatically recorded landmark reduce dead reckoning error to a value close to the value at the point where the landmark was originally recorded.

The *next_scan_config()* function evaluates the geometry of the current partial world to determine the best path for the robot to follow to continue the world space exploration. The *next_scan_config()* function runs at the end of each translational scan. The function examines all edges in the current partial world and determines which edge is the best "inferred" edge to investigate next. The function performs a simple path planning function to plan a path from the robot's current configuration to the next translational scan. This path is planned using a depth first search strategy in which Yamabico stores it path using the *Path_List PL* as the exploration progresses and backtracks back along *PL* to the next inferred edge. This behavior is illustrated in Figure 8.5. This backtracking behavior allows the robot to explore its world using a depth-first-search strategy which is equivalent to an "in order" traversal of the free space in the world [Manber 89].

In Figure 8.5 Yamabico starts out at point 1 and moves to point 2 for a translational scan. During this scan, three "inferred" edges are identified; e_a, e_b, and e_c. Since e_c is the closest "inferred" edge to point 2, Yamabico backtracks back along its first translational scan to point 3 and then turns right to investigate edge e_c. Yamabico scans from point 3 to point 4. At point 4 the next closest "inferred" edge is e_b. At this point Yamabico turns around again an scans from point 4 to point 5. At point 5 an open area exists to Yamabico's

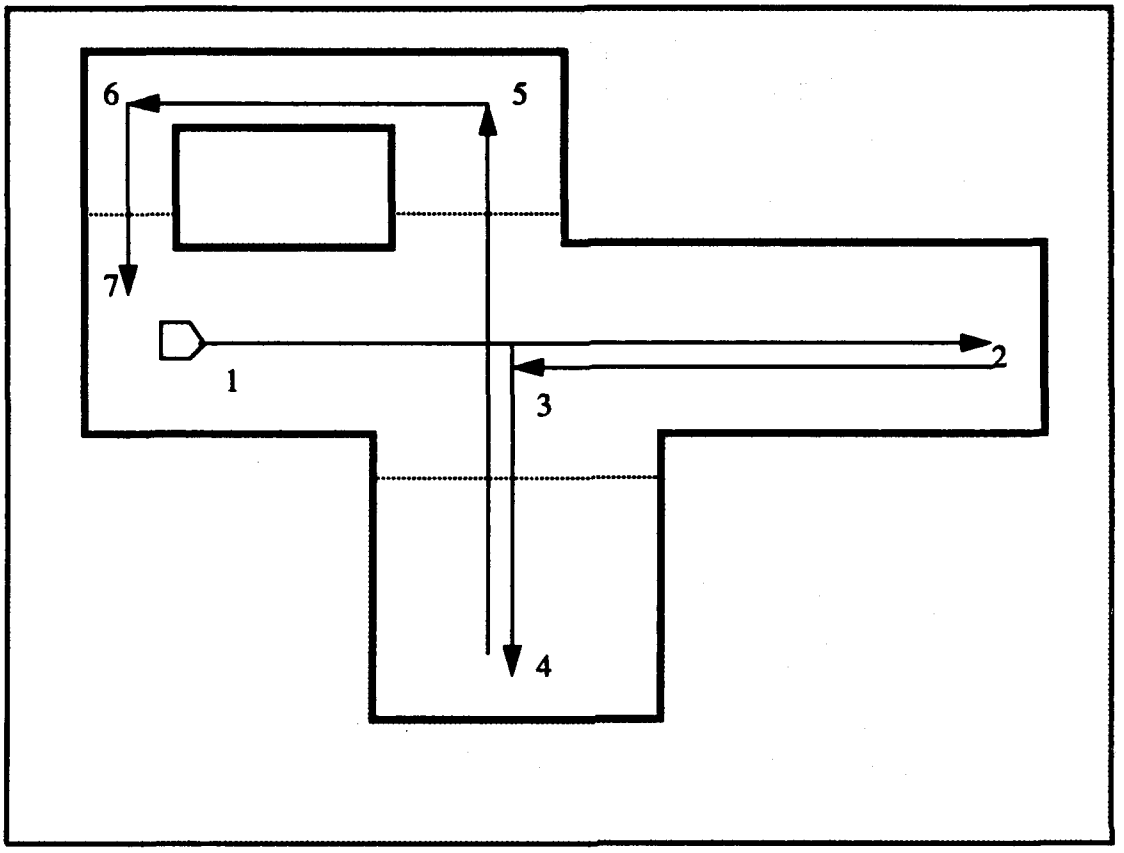


Figure 8.5 - The *next_scan_config()* Depth First Search Exploration Behavior

left, so it turns left and scans to point 6. At point 6, open area again exists to the left so Yamabico turns left and then scans from point 6 to point 7.

The *next_scan_config* function algorithm is given in Figure 8.6. This function guides Yamabico to scan "inferred" edges using a depth first search strategy. This function favors paths that increase the overall mapped area as quickly as possible. If the *PW* is evaluated as incomplete, the robot uses the *next_scan_config()* function to determine where to next move for translational scanning. This feature underscores the fact that the robot must move to a new scanning configuration (*C*) based upon the derived partial world model (*PW*). The next scan configuration is selected to optimize the open area mapped by Yamabico. The following rules apply:

```
next_scan_config(PW, PL, C)
Partial_World *PW;
Configuration *PL;
Configuration C;
{
    EDGE *closest_edge;
    CONFIGURATION *path_element;

    if (obstacle forward and clear left) {
        path_element = turn_left();
        add_path_to_list(path_element, PL);
        }
    else {
        closest_edge = find_closest_edge(PW, PL, C);
        DFS_backtrack(PW, PL, C);
        move(PL);
        }
} /* end next_scan_config() */
```

Figure 8.6 - The *next_scan_config* Algorithm

(1) If an obstruction is forward and the left side sensor reports no obstructions to the left, then Yamabico turns left 90° for the next translational scan. The path element for the scan is added to *PL*.

(2) Otherwise Yamabico turns 180° and backtracks along *DFS* to the closest inferred edge suitable for investigation.

C. SUMMARY

An algorithm for Yamabico's automated robot cartography using ultrasonic range finders is developed in this chapter. The algorithm acts in a greedy fashion to continuously increase the planar area mapped by the vehicle. Intelligent vehicle motion is required to reach all portions of the world space. Robot motion to explore the world space results in odometry

error. The algorithm uses heuristic and spatial reasoning to correct some odometry error in real time as the cartography is performed.

The real sensor limitations of finite range, limited incidence angle for returns, and specular reflection tend to complicate the cartography problem. The algorithm presented in this chapter provides a means to work within these physical limitations. The real cartography algorithm is implemented on the autonomous mobile robot Yamabico-11 and the experimental results are explained in Chapter IX.

IX. EXPERIMENTAL RESULTS AND CONCLUSIONS

Experimental results from Yamabico motion control, dead reckoning error correction, and cartography experiments are presented in this chapter. Some experimental results from the development of the motion control portion of the MML language are presented. Several representative experiments are included, however, hundreds of additional successful and not so successful experiments were conducted for this dissertation. Space considerations prohibit including all of the experiments. A "user.c" robot command file and a plot of the vehicle's observed trajectory are included with each experiment.

The dead reckoning error correction experimental results are presented in the form of two representative examples. One is a single landmark experiment with Yamabico moving in a nine meter long racetrack pattern and the second is a more complex three landmark experiment with approximately 30 meters of travel per revolution.

The automated cartography experiments demonstrate Yamabico's ability to map a closed world space using only ultrasonic range finders. Yamabico path tracking capability and periodic odometry correction using landmarks are required to derive a high quality map from an indoor world. The conclusions drawn from the experimental results are then summarized.

A. MOTION CONTROL EXAMPLES

1. Observation Plan

Early experiments were designed to test Yamabico's ability to follow straight and curved paths. These tests revealed that an extensive redesign of most of the existing MML kernel was required. The general plan was to evolve the previous configuration-to-configuration tracking system into the new path tracking system [Kanayama 91a]. The first goal was to program Yamabico to track along a straight line or a circular arc path elements. Observations and measurements of Yamabico's motion were planned using marks made on

the floor of the test area. Yamabico's expected and actual position were compared. Path tracking was thought to provide smoother vehicle motion and improved odometry correction capability as a result of changing the method of tracking.

The observation plan is a series of representative tests of the robot's ability to track the path elements described in Chapter IV. The "user.c" specifications for several simple test programs are given along with plots of the robot's motion. These test programs are samples from the test battery designed to test Yamabico's locomotion functions. Many combinations of locomotion functions were not tested, because exhaustive testing is intractable due to the large number of possible combinations of vehicle motion commands. Adequate testing was conducted to eliminate bugs in the most used parts of the motion control software and to provide a reasonable user confidence level. Troubleshooting and improvement of the motion control software continues in the Yamabico research group.

2. Observation Results

The first locomotion test for MML was simple line tracking. The "user.c" file and Yamabico's trajectory plot is shown in Figure 9.1 Yamabico is given its starting configuration in the user's global coordinate system using the *set_rob* command. This starting configuration is called *start* in this program and any valid 'C' variable is an acceptable configuration variable in the MML language. The *def_configuration* function is used to define configurations needed in the course of the "user.c" program. In this program, the *start* configuration is defined by the parameters $x = 0.0$, $y = 100.0$, $\theta = 0.0$, and $\kappa = 0.0$. The *set_rob* function is required at the beginning of every "user.c" file to initialize Yamabico's odometry configuration. Yamabico is then commanded to track the *first* path element using the *bline(&first)* function. This function commands Yamabico to track the straight path element starting at its current configuration and passing through the configuration $x = 100.0$, $y = 100.0$, and $\theta = 0.0$. Since $\kappa = 0.0$ for both *first* and *second* and $\kappa = 0.0$ means a zero curvature path element, therefore these path elements are straight line path elements. Smooth motion from the *first* configuration onto the *second* path element is observed as

```
#include "mml.h"

user()
{
     CONFIGURATION start;
     CONFIGURATION first;
     def_configuration(0.0, 100.0, 0.0, 0.0, &start);
     def_configuration(100.0, 100.0, 0.0, 0.0, &first);
    def_configuration(0.0, 0.0, 0.0, 0.0, &second);

     set_rob(&start);
     bline(&first);
     line(&second);
}
```

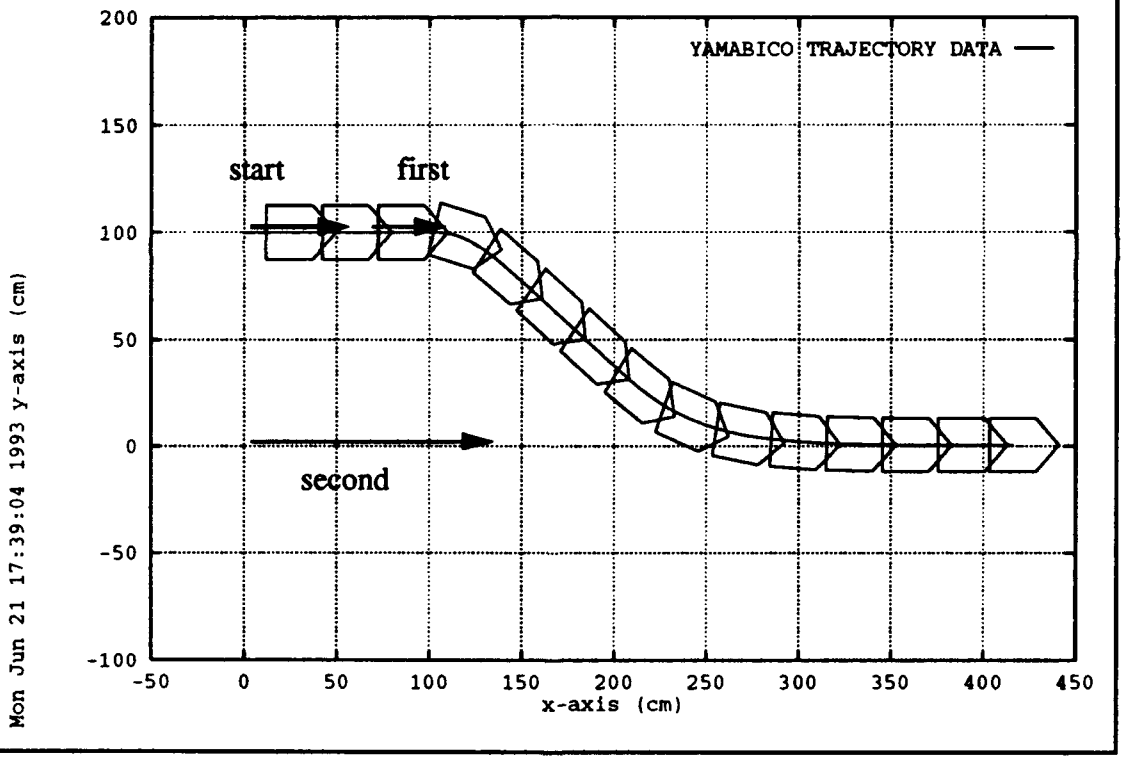


Figure 9.1 Simple Line Tracking

shown in Figure 9.1. The vehicle was observed to closely track the commanded line segments based upon hand measured marking on the floor. In no case did the vehicle's position deviate from the expected path by more than 2.0 centimeters. In this experiment, vehicle

guidance commands were issued at a 100 Hz rate, Yamabico's velocity is 30.0 centimeters per second, and the size constant is 20.0. The speed and size constant default to nominal values since they are not specified by the user. Program illustrates simple lane changing behavior that could be used for obstacle avoidance.

The second experiment demonstrates circular path element tracking. A circle is specified in the same manner as a straight line using the *line* function, except the curvature of the circular path element is not zero ($\kappa \neq 0$). The "user.c" file and the plot of the robot's trajectory is shown in Figure 9.2. Notice that only six lines of MML code are used to specify this complex robot behavior. Yamabico is given the initial configuration *start* such that $x = 0$, $y = 0$ and the vehicle initial orientation is θ = HPI which means 90° with respect to the x-axis. Yamabico is commanded to track a circular path *first* with the configuration $x = 200$, $y = -100$, $\theta = 0$, and $\kappa = -0.1$. The x and y parameters specify the starting point for the circle, θ gives the orientation, and $\kappa = -0.1$ means the circle's radius $r = 1/\kappa = 1/-0.1 = -100.0$ centimeters. The negative value of the radius indicates a clockwise direction for the *first* path element. Further MML implementation details appear in Appendix A. Yamabico leaves its starting configuration and turns sharply to the right. The sharpness of this turn is determined by the value of the size constant (s_o). Once again the nominal speed and size constant values are used by default. Once started Yamabico immediately begins tracking the circular path element *first*. It continues tracking the circle indefinitely since no stopping command is issued. At no point during this experiment did Yamabico's trajectory deviate by more than 1.0 centimeter from the expected path element trajectory.

The third example demonstrates circular backward line tracking. Yamabico tracks the smaller inner circle as illustrated in Figure 9.3. The end of the backward line is located at the configuration $x = 100.0$, $y = 0.0$, and $\theta = 0$. When Yamabico reaches the end of the backward line it then switches to tracking the outer circle (*second*). If *first* was the last path in the "user.c" file Yamabico would have stopped at the end of the backward line. The second path element is a counterclockwise circular path element with radius $r = 1/\kappa = 1/-0.1 = +200.0$ centimeters. In this experiment robot velocity is set to 15.0 centimeters per second

```
#include "mml.h"

user()
{
    CONFIGURATION start;
    CONFIGURATION first;
    def_configuration(0.0, 0.0, HPI, 0.0, &start);
    def_configuration(200.0, -100.0, 0.0, -0.01, &first);

    set_rob(&start);
    line(&first);
}
```

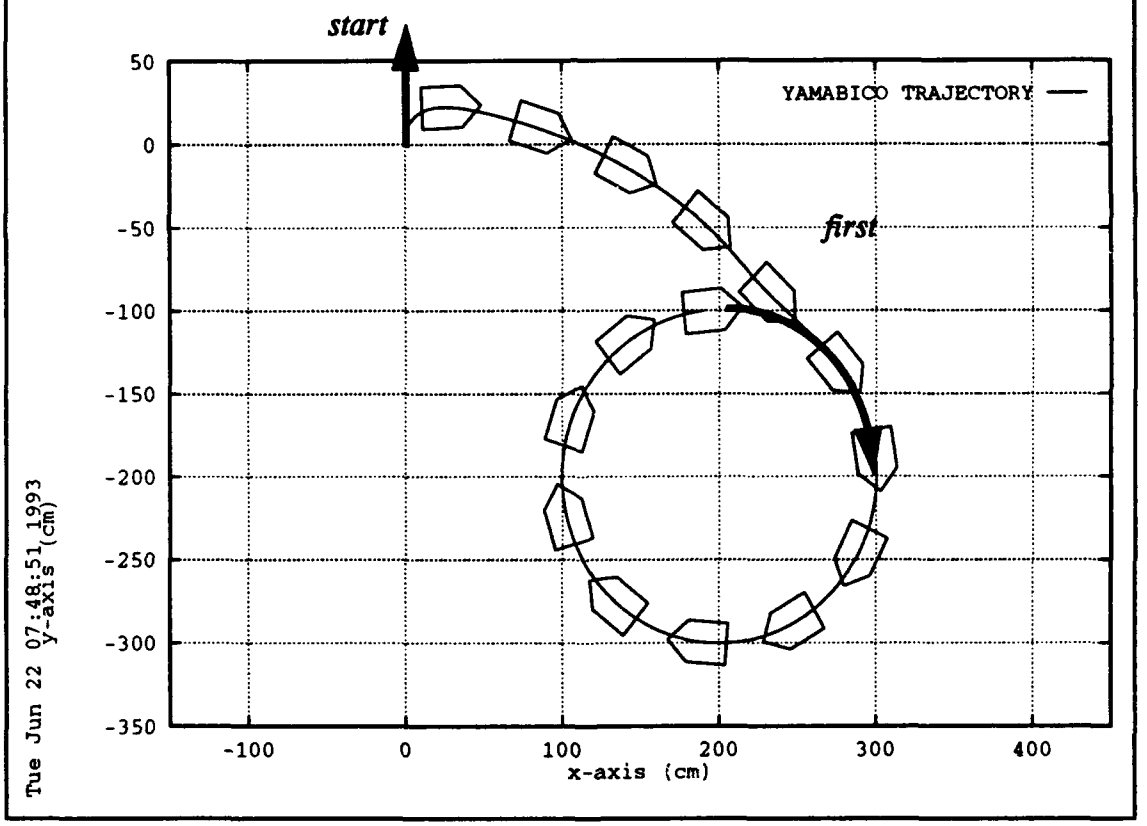


Figure 9.2 Circle Tracking

```
#include "mml.h"

user()
{
    CONFIGURATION start;
    CONFIGURATION first;
    CONFIGURATION second;

    def_configuration(0.0, 100.0, -HPI, 0.0, &start);
    def_configuration(100.0, 0.0, 0.0, 0.0, 0.01, &first);
    def_configuration(100.0, -100.0, 0.0, 0.005, &second);

    size_const(20.0);
    speed(15.0);
    set_rob(&start);
    bline(&first);
    line(&second);
}
```

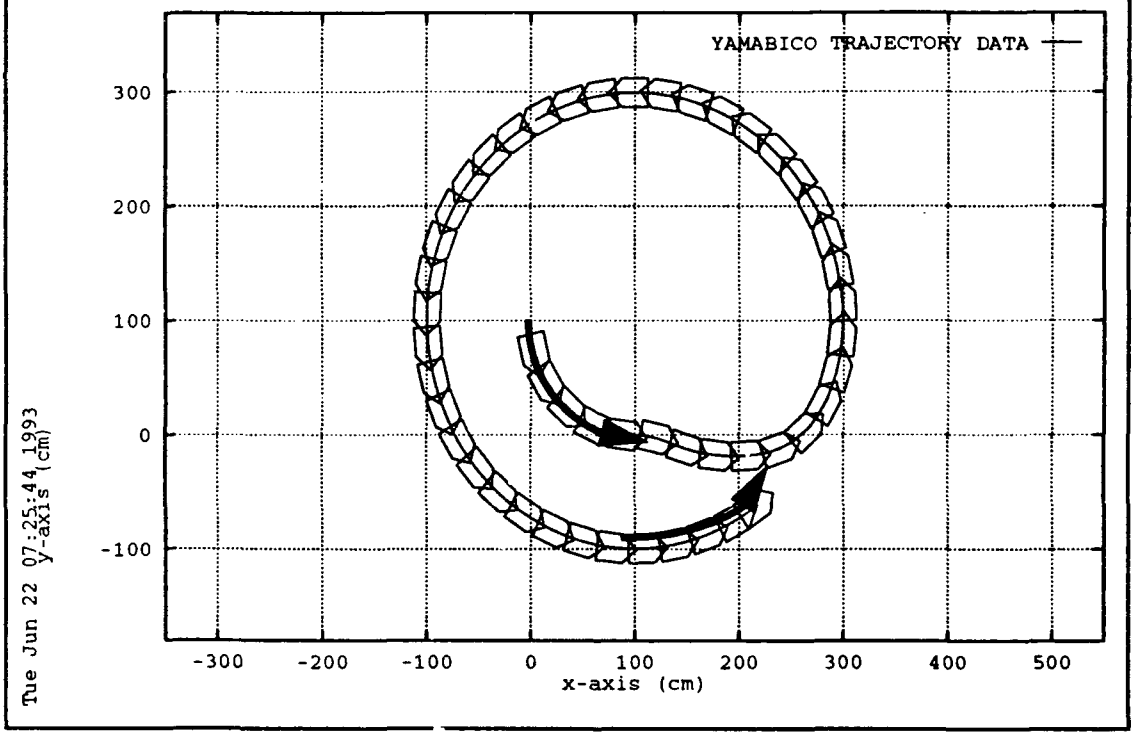


Figure 9.3 Circular Backward Line Tracking

and the size constant is 20.0. Yamabico's deviation from both of the intended path elements was less than 2.0 centimeters.

In Figure 9.4, Yamabico is commanded to track a path consisting of four sequential path elements. The first path is a straight line path given by the configuration *start*. The second path element is a parabola given by the *first* specification. The parabola's focus is at $x = 300.0$, $y = 100.0$, and the directrix is given by the configuration $x = 0.0$, $y = 0.0$, $\theta = 0.0$. The parabolic path element *first* has five component parameters as discussed in Chapter IV and Appendix A. The path elements *start* and *first* intersect at the point $x = 200.0$, and $y = 100.0$. Yamabico calculates a leaving point on the *start* path element to allow for a smooth transition to the *first* parabolic path element. Then Yamabico tracks the parabola *first*. The *first* and *start* path elements intersect and Yamabico calculates the next leaving point on the parabolic path element *first*. When this leaving point is reached Yamabico switches to tracking the straight line path element *start*. The path elements *start* and *third* also intersect. Yamabico then calculates the next leaving point on the *start* path element. When the last leaving point is reached, Yamabico switches to tracking the *third* path element. The entire program represents 30.0 seconds of robot motion at the default nominal velocity of 30.0 centimeters per second. The distance constant (s_o) controls the path element transition sharpness, the default value is 20.0 centimeters.

The last example is a demonstration of cubic spiral tracking. In Figure 9.5 Yamabico is started at the origin and commanded to track a cubic spiral from the *start* configuration to the *first* configuration. Two cubic spirals are actually tracked, one before the point of inflection and another after it. At no point during this experiment did Yamabico's trajectory deviate by more than 2.0 centimeter from the expected path element trajectory. Cubic spiral path tracking is described in greater detail in another publication [Fish 93].

Yamabico closely tracked the intended paths in all five experiments. The motion control results indicate that the path tracking technique yields extremely small motion control errors for vehicle travel over a given distance. Odometry error is experimentally measured in section B of this chapter.

```
#include "mml.h"

user()
{
    CONFIGURATION start;
    PARA first;
    CONFIGURATION third;

    def_configuration(0.0, 200.0, 0.0, 0.0, &start);
    def_parabola(300.0, 100.0, 0.0, 0.0, 0.0, &first);
    def_configuration(700.0, 200.0, -HPI, 0.0, &third);

    set_rob(&start);
    line(&start);
    parabola(&first);
    line(&start);
    line(&third);
}
```

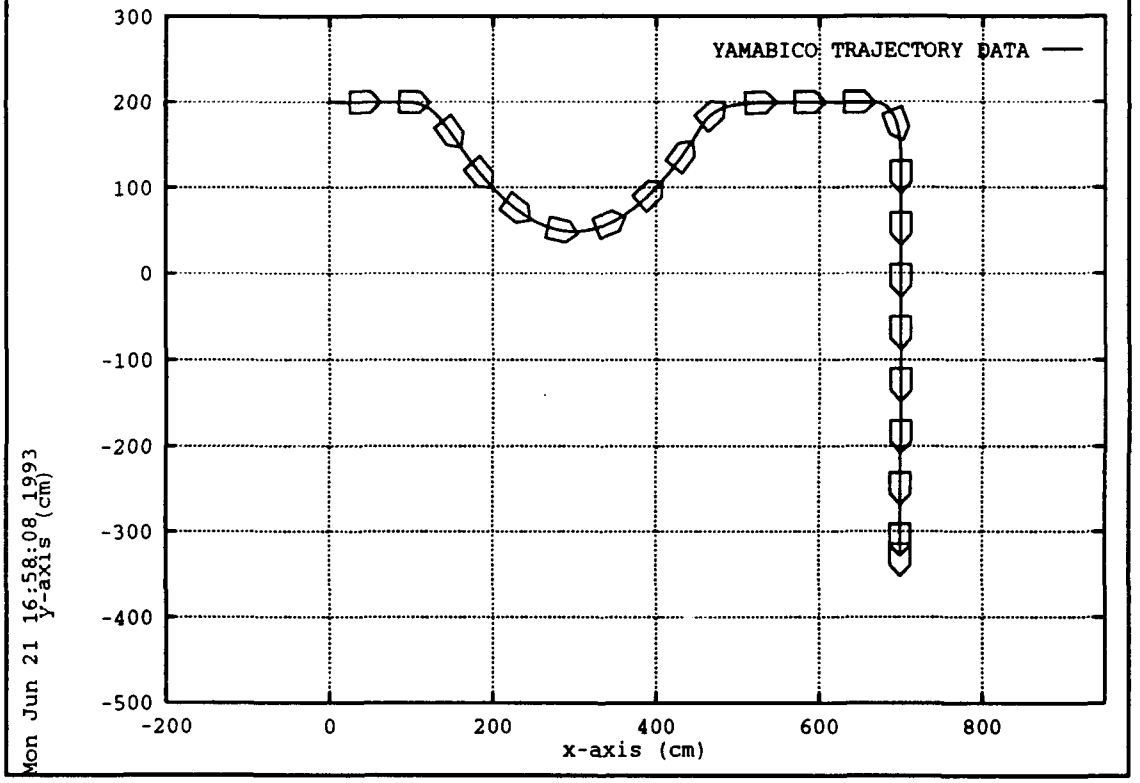


Figure 9.4 Parabolic Tracking

```
#include "mml.h"

user()
{
    CONFIGURATION start;
    CONFIGURATION first;

    def_configuration(0.0, 0.0, 0.0, 0.0, &start);
    def_configuration(200.0, 0.0, HPI, 0.0, &first);

    size_const(20.0);
    speed(15.0);
    set_rob(&start);
    config(&first);
}
```

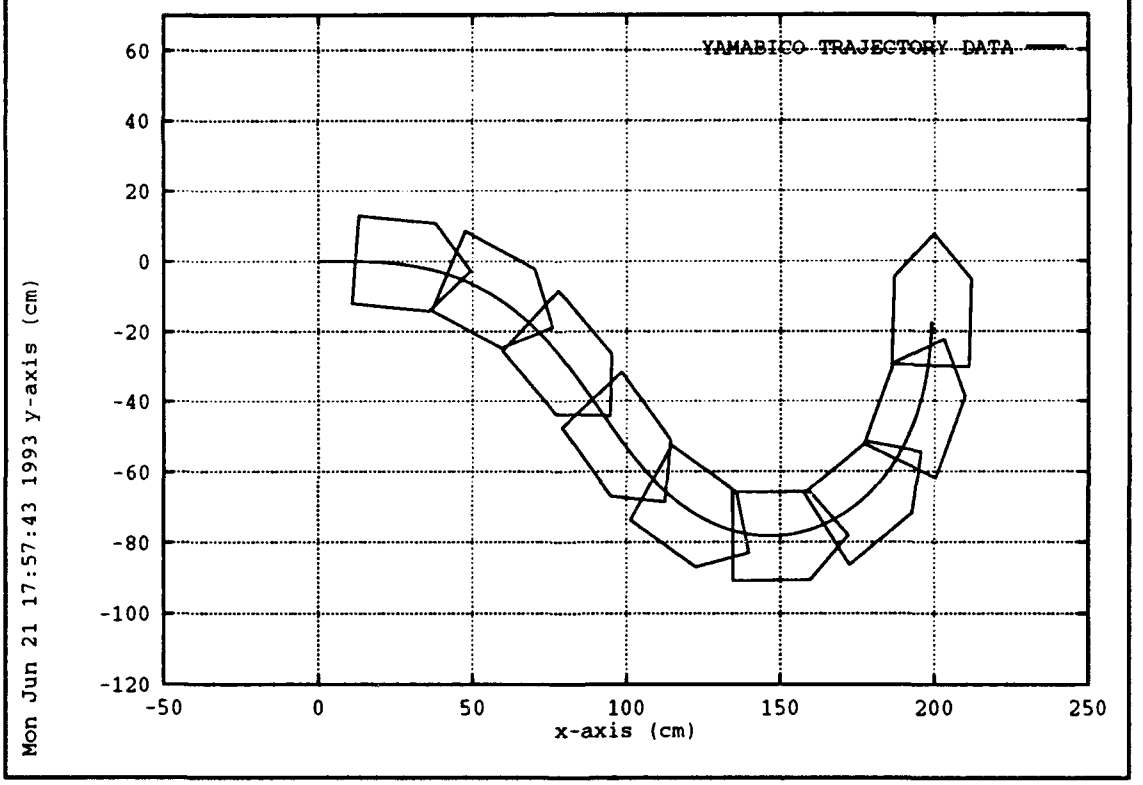


Figure 9.5 Cubic Spiral Tracking

B. ODOMETRY EXPERIMENTAL RESULTS

1. Experimental Plan

Two kinds of odometry correction experiments are performed using Yamabico-11. To verify the fundamental correctness of the algorithm, first a simple racetrack path with a single landmark is used. Yamabico moves repeatedly around this racetrack path which is composed of four separate path elements. Yamabico is programmed to make an odometry correction once per lap using a single landmark. The amount of odometry error is measured by hand and by examining logged sonar data. These measurements serve to establish a correlation between type of motion and rate of accumulation of odometry error.

The second experiment is slightly more complex. The multiple landmark experiment is conducted to prove the utility of this algorithm for more sophisticated vehicle navigation. The sparse assignment of landmarks serves as a worse case to prove the utility of the odometry correction and precise vehicle tracking algorithm in an indoor environment. The experimental plan is: (1) program the robot to move in a repeating pattern with about five left and right turns; (2) use several fixed landmarks that are scanned by the side looking sonars to perform odometry error corrections.

2. Experimental Results

The first MML program executes an oval path for Yamabico. This is a skeleton of the whole program which is much longer and contains other functions. The motion control portion of the program and the resulting robot motion are shown in Figure 9.6. The dead reckoning correction code appears in Appendix D. In each lap of this oval path execution, the odometry error correction is performed and the error configuration ε is recorded. The total distance traveled is 9.14 meters per lap. In this experiment, only one landmark is used for the odometry error detecting purpose. Table 9.1 shows the raw experimental data for Yamabico traveling ten laps at 25 cm/sec. Notice that the results show the error configuration for each lap is small and nearly equal. This proves that Yamabico's motion control and

```
#include "mml.h"

user()
{
    CONFIGURATION start, first, second, third, fourth;
    int laps = 10;
    int lap_count = 0;
    def_configuration(1200.0, 65.0, 0.0, 0.0, &start);
    def_configuration(1100.0, 65.0, 0.0, 0.0, &first);
    def_configuration(1500.0, 65.0, 0.0, 0.02, &second);
    def_configuration(1700.0, 165.0, PI, 0.0, &third);
    def_configuration(1200.0, 165.0, PI, 0.02, &fourth);

    set_rob(&start);
    while (lap_count < laps)
    {
        line(&first);
        line(&second);
        line(&third);
        line(&fourth);
        ++lap_count;
    }
}
```

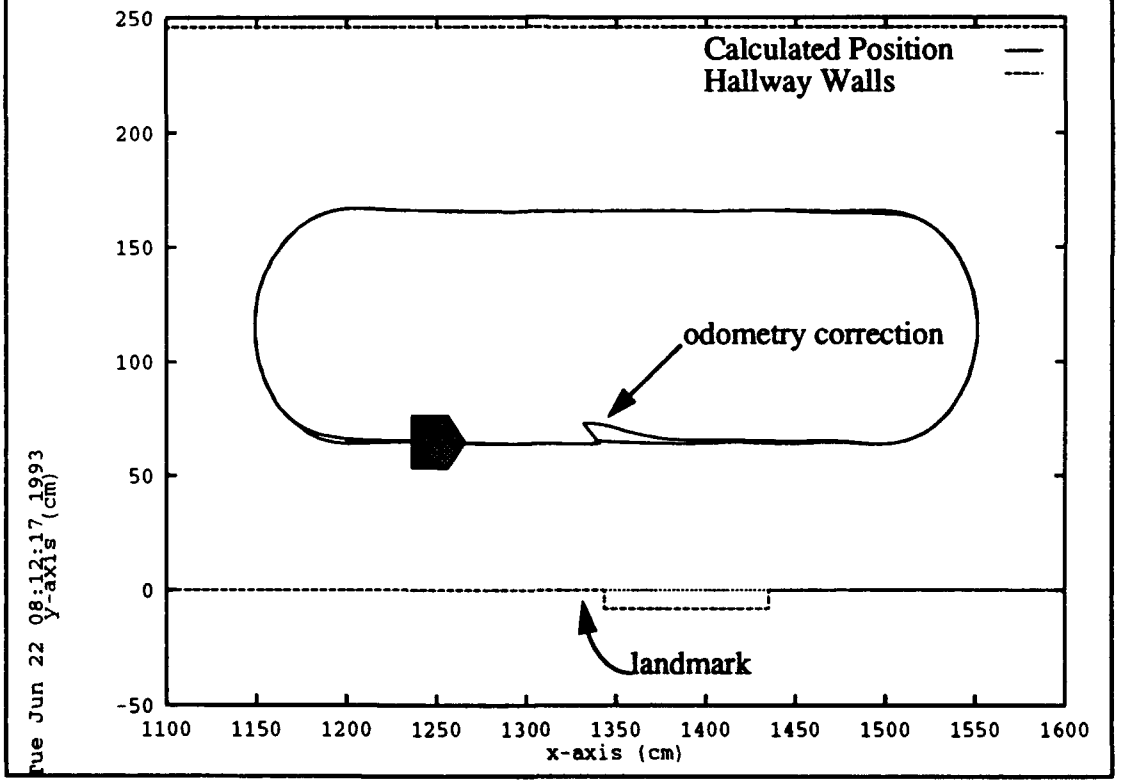


Figure 9.6 Single Landmark Odometry Correction Experiment Code

Table 9.1 ODOMETRY ERROR CORRECTION (25 CM/SEC)

Lap	Δx (cm)	Δy (cm)	Δθ (radians)	Δθ (degrees)
1	-1.591	-0.620	0.0120	0.6875
2	-1.924	-0.828	0.0120	0.6875
3	-2.191	-0.671	0.0110	0.6303
4	-1.181	-1.143	0.0290	1.6616
5	-2.401	-0.298	0.0100	0.5730
6	-2.152	-0.936	0.0290	1.2032
7	-2.067	-0.905	0.0150	0.8594
8	-2.054	-0.975	0.0170	0.9740
9	-2.409	-0.793	0.0130	0.7448
10	-1.297	-1.153	0.0370	2.1199

odometry functions are precise and that the odometry error correction algorithm is working as desired.

This experiment was repeated at various robot speeds. The average error configuration over ten laps at speeds of 10, 15, 20, 25, and 30 cm/sec are shown in Table 9.2 below. As in the previous experiment, error correction was made each lap. Notice that the average dead reckoning error per lap tends to increase with increasing robot velocity. In Fig-

Table 9.2 AVERAGE ODOMETRY ERROR AS A FUNCTION SPEED

Speed	Δx (cm)	Δy (cm)	Δθ (radians)	Δθ (degrees)
10	-0.207	-1.148	0.0020	0.11459
15	-0.775	-0.765	0.0045	0.25783
20	-1.222	-0.696	0.0084	0.48128
25	-1.927	-0.832	0.0177	1.04135
30	-2.289	-1.112	0.0167	0.95684

ure 9.7 the odometry corrections for ten laps on the racetrack pattern are plotted. The largest odometry correction is the first correction since the robot is intentionally displaced from the proper starting configuration (*start* = (1200.0, 65.0, 0.0, 0.0)). The odometry error correction code corrects this initial placement error as if it were a large odometry error. Subsequent dead reckoning error corrections are smaller and consistent indicating Yamabico's odometry parameters could be tuned to improve the results of this experiment. This experiment also shows that odometry error on the robot is small and repeatable. Odometry error could be reduced by self correction of odometry parameters in software, however this experiment was not conducted.

Yamabico's odometry error for ten laps at 25.0 centimeters per second is plotted in Figure 9.8. The location of the tail of each arrow is the x, y magnitude of the odometry error and the orientation of each arrow is the θ component of the odometry error. Notice that most odometry errors fall within a small cluster of values. The general trend observed was that Yamabico tends to turn slightly more than π radians programmed. Overall the er-

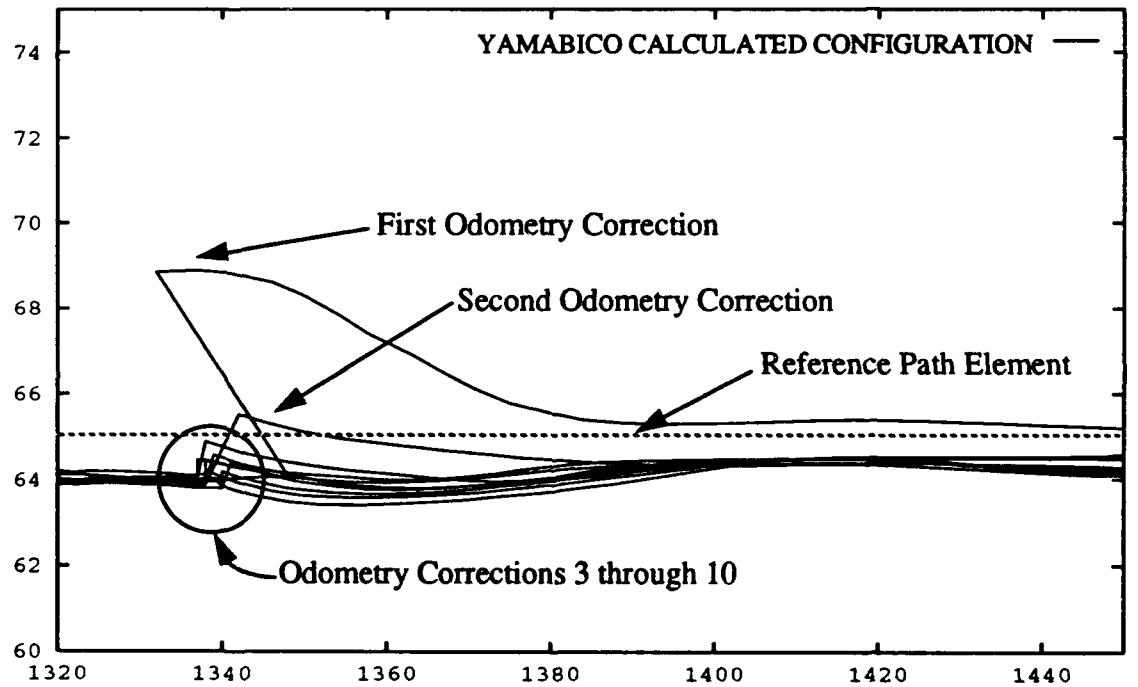


Figure 9.7 Ten Robot Odometry Corrections at 15 centimeters per second

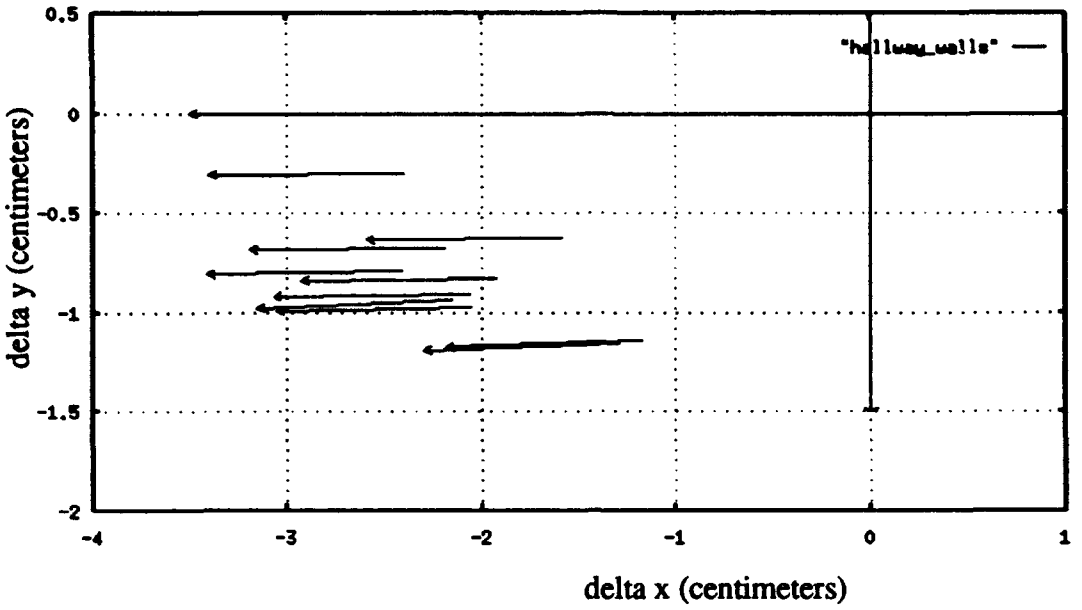


Figure 9.8 Robot Odometry Error at 25 cm/second

ror is small, but the author believes that wheel slippage during the turns contributes to the larger odometry errors at the higher speeds. Mismatch between the left and right wheel drive motors also causes some error because the same pulse width modulation curve is used for both motors.

Table 9.3 MULTIPLE LANDMARK ODOMETRY CORRECTION

Landmark	Δx (cm)	Δy (cm)	Δθ (radians)	Δθ (degrees)
1	+13.843	-4.741	+0.0166	+0.9511
2	-14.498	+7.695	-0.0006	-0.0343
3	+7.326	-21.183	+0.0135	+0.7735

The multiple landmark odometry experiment is conducted in an unmodified indoor environment. Three landmarks constitute the robot's abstract geometric model of the world and are used for the purpose of odometry error detection and correction. Multiple

landmark correction behavior is closer to the desired goal of autonomous robot navigation. A section of the hallway on the fifth floor of a building at the Naval Postgraduate School is used. Average odometry corrections for the multiple landmark experiment are shown in Table 9.3. A row in the table stands for a correction result at a numbered landmark. Relatively small and consistent odometry corrections are also observed in this experiment. One lap of the multiple landmark correction experiment is shown in Figure 9.9. The landmarks are

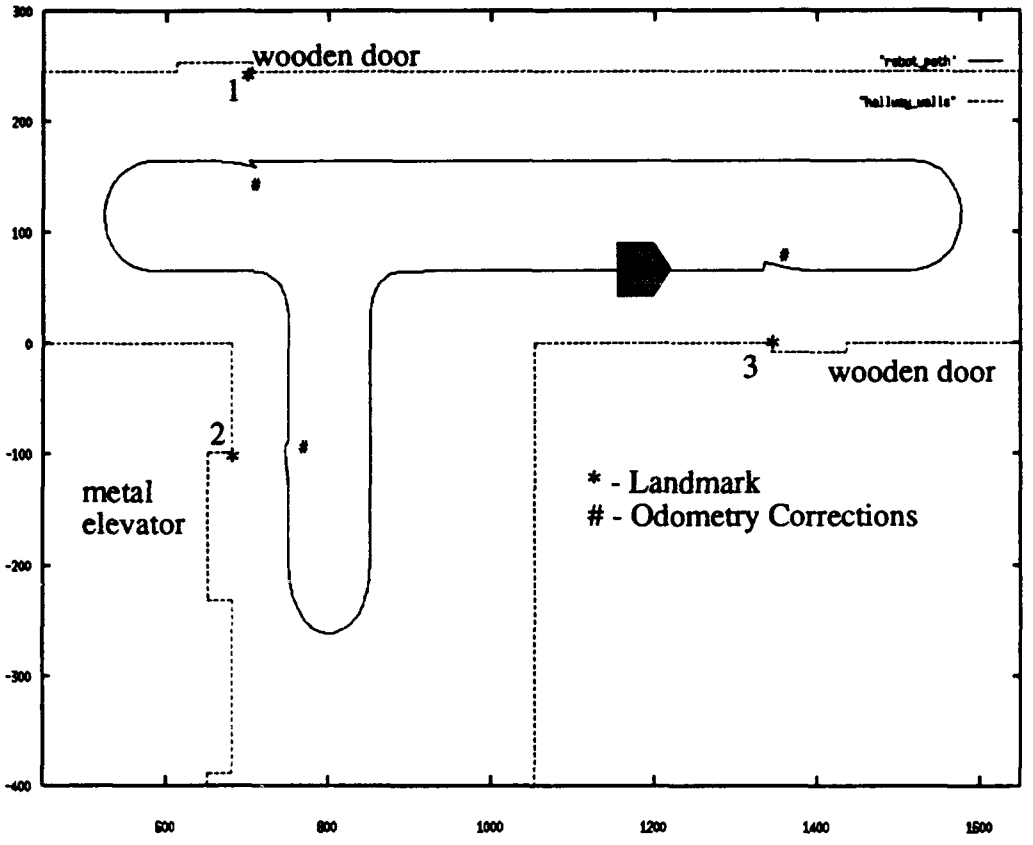


Figure 9.9 Multiple Landmark Odometry Correction

numbered to correspond with the landmark numbering in Table 9.3. This result shows that only three landmarks provide enough dead reckoning error correction for Yamabico to navigate a 30 meter circuit in an ordinary indoor environment.

C. AUTOMATED CARTOGRAPHY EXPERIMENTAL RESULTS

1. Experimental Plan

The following automated cartography experiments are planned to test the algorithm's implementation on a small, orthogonal world using Yamabico-11;

(a) **Orthogonal world alignment** - Place Yamabico in a variety of configurations in the test world. Run the *find_orthogonal_orientation* function alone at each configuration. Quantify the implementation's precision with regard to orthogonal alignment.

(b) **Straight hallway following** - Program Yamabico to follow a hallway with obstacles and doorways, and record its path. The goal is to have Yamabico travel down the centerline of the hallway while correcting dead reckoning errors. Line segments extracted from both the left and right hand walls of the hallway are used to align Yamabico to the centerline of the hallway. Line segments must have sufficient length (100 cm) and be located near their expected configuration to be acceptable for correcting dead reckoning errors.

(c) **Partial world building by a single translational scan** - In this experiment Yamabico is commanded to move several meters on a path orthogonal to the world space. A partial world is built by Yamabico based upon the line segments extracted from this sonar scan. The partial world built is compared with hand measured maps for accuracy.

(d) **Full automated cartography**- The full partial world building automated cartography experimental plan involves perform cartography experiments on a small orthogonal world with holes. The experimental area is an office building hallway, modified with artificial barriers to keep the size of the world reasonably small. Cardboard boxes serve as the obstacles holes in this world space. Yamabico is placed at an arbitrary configuration inside of this world space and the cartography program is started. The cartography program runs to completion. The map built by Yamabico is then transferred back to the host computer for analysis. The polygonal world map derived from the robot's world space explo-

ration is compared to the actual hand-measured map of the world space. Reasons for discrepancies are analyzed.

2. Experimental Results

The code for the robot automated cartography experiments is listed in Appendix E due to its length.

(a) The *find_orthogonal_orientation* function aligns Yamabico's internal coordinate system orthogonal to the world space based upon the line segments extracted from a 360° rotational scan. This function chooses the closest extracted line segment longer than 20 cm for alignment. Since four line of the segments are parallel, the initial scan may properly take one of two directions. Yamabico is commanded to rotate the shortest angular distance to align parallel with the closest extracted line segment. In Figure 9.10, Yamabico is placed in the hallway with no prior knowledge of the world space. Yamabico rotates clockwise 2π radians and extracts five line segments from sonar. These line segments are interpreted to allow the robot to determine the orientation of the first translational scan. In all experiments conducted Yamabico's orientation was within ±3° of parallel to the closest orthogonal surface. Certain situations were observed to improve the accuracy of this function. For instance the close proximity of a long flat wall improves accuracy.

(b) In Figure 9.11, a hallway following experiment is shown. In this experiment, Yamabico used a straight hallway assumption to provide odometry correction as it moved approximately 20 meters down the hallway. This same hallway following experiment has been used to map a hallway 70 meters long. The hallway is 246.0 cm wide. Yamabico started at the configuration $x = 0.0$, $y = 123.0$, $\theta = 0.0$ and tracked a straight line path element down the center of the hallway. Side mounted sonars were used to scan the right and left hallway walls. The sonar data from the side scanning sonars is used for periodic odometry corrections based upon the assumption that the hallway walls on either side of the robot are straight. The sonar data is extracted and processed as line segment data. Line segments (or edges) of sufficient length provide distance and orientation information for Yamabico to

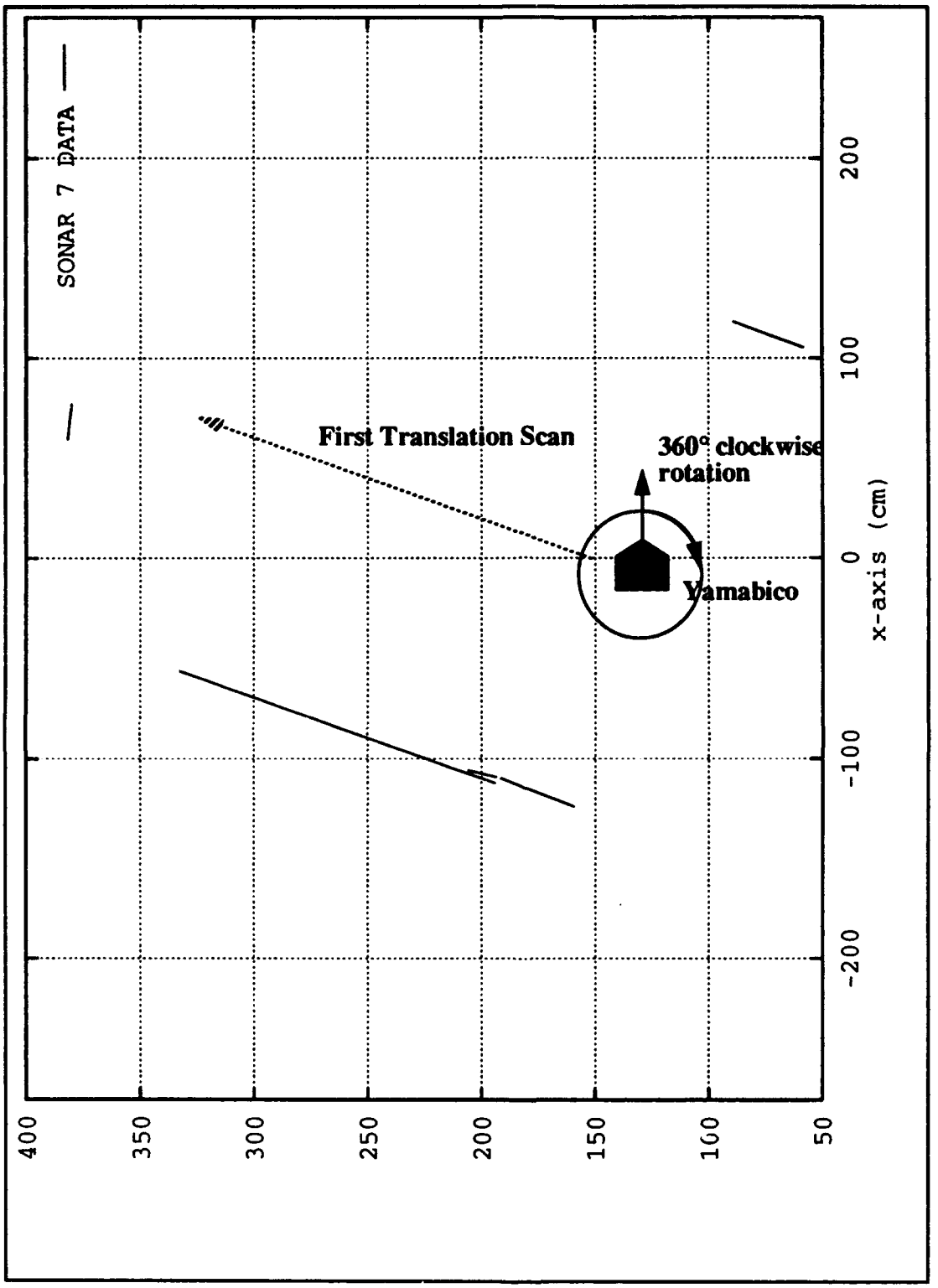


Figure 9.10 The Find Orthogonal Orientation Experiment

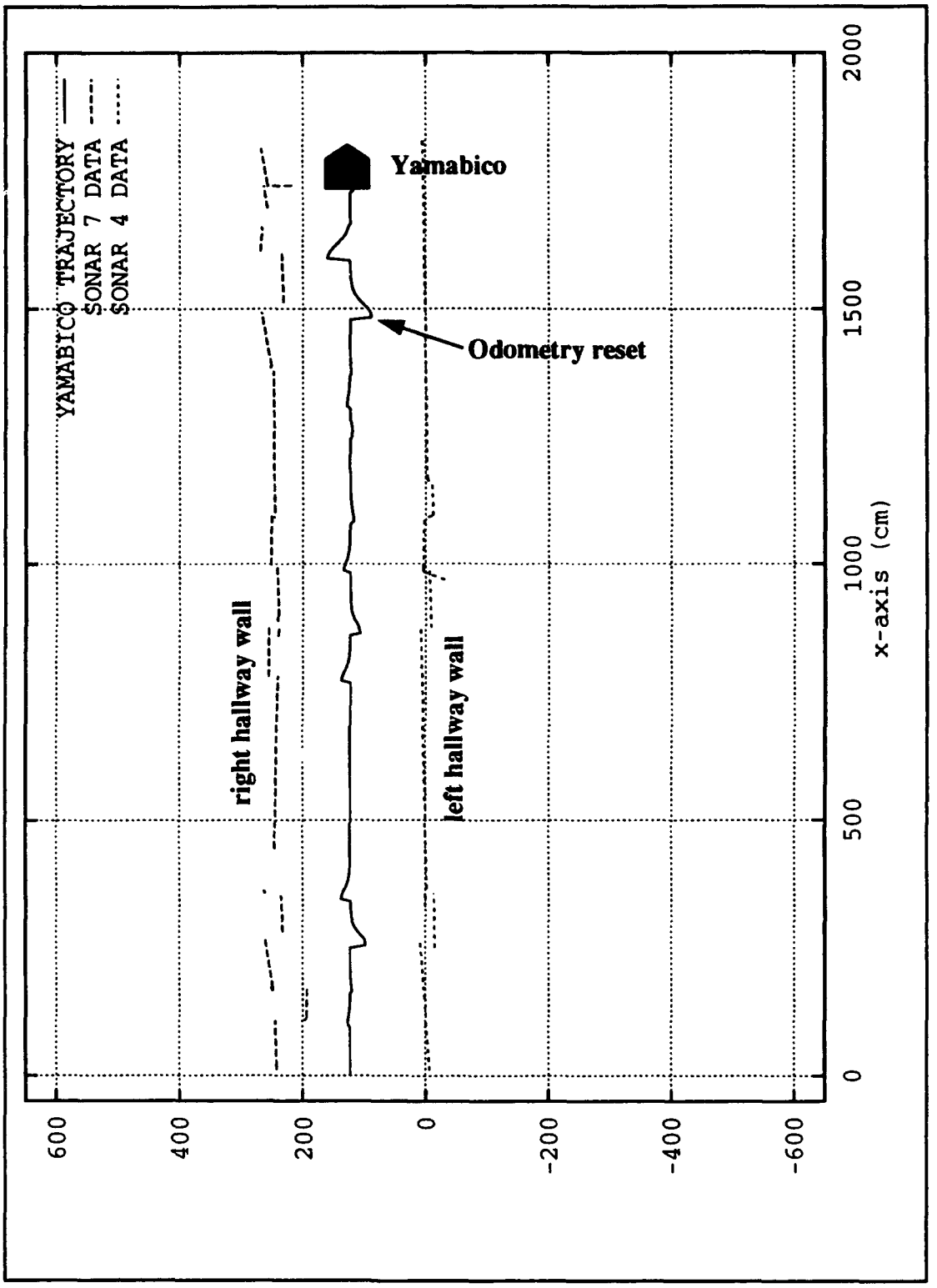


Figure 9.11 Hallway Following Cartography Experiment

fix its position relative to the center line of the hallway. Yamabico then performs an odometry estimate reset to correct its configuration with respect to the center line of the hallway. Since Yamabico has no prior knowledge of the hallway, the translational error (absolute distance moved down the hallway) in the odometry estimate cannot be corrected.

The sonar data extracted from the side-scanning sonars is also stored as cartography data. Hallway following with odometry correction is an important component of automated indoor cartography since many office buildings have hallways narrow enough so Yamabico can travel down the center of the hallway and sense both walls. This program is robust since Yamabico stays in the center of the hallway despite some sensor noise. Noisy segment data arises from specular reflections, sensor noise, and people walking in the hallway. This noise is filtered out in part by (1) rejecting non-orthogonal sonar segments, (2) rejection of short segments, and (3) rejection of segments beyond the sonar's effective linear fitting range. Yamabico configuration did not deviate from the reference path element by more than 40.0 centimeters during these experiments.

(c) In Figure 9.12 a partial world constructed from a single 14 meter translational scan down a hallway is shown. All "real" edges are constructed from extracted sonar segment data. The "inferred" edges are constructed at the beginning and at the end of the translational scan to bound unexplored area. The elevator foyer area on the right hand side of the translational scan appears as 60.0 centimeter deep break in the right hand wall. An "inferred" edge parallel to Yamabico's translational scan path and two meters from the robot's path is constructed to indicate a portion of the world is out of sonar range. The elevator foyer is about five meters deep. The partial worlds constructed in this experiment did not deviate from hand measured map by more than 10.0 centimeters.

(d) The full partial world experiment derived more complete maps of the world space. In Figure 9.13, Yamabico builds a more complex map of the hallway portion of an office building. Yamabico is placed at position 1 with an arbitrary orientation in Figure 9.13. Yamabico rotates 360° and extracts several line segments representing portions of the hallway within sonar range. Yamabico aligns itself with the parallel edges extracted from

the rotational scan and then performs a translational scan from position 1 to position 2. Yamabico stops at position 2 because an obstacle is detected forward. This first translation scan is about 14 meters long and a partial world similar to the one shown in Figure 9.12 is built in Yamabico's memory.

Yamabico then turns around to the right and moves from position 2 to position 3 to investigate the "inferred" edge bounding the opening to the elevator foyer. The next translation scan move Yamabico from position 3 to position 4. The doorways for the two elevators on Yamabico's right hand side are extracted as edges. Yamabico stops at position 4 because an obstacle is detected forward and then turns left 90°.

Next Yamabico performs a translational scan from position 4 to position 5. Then takes another left turns to scan to position 6. The entire elevator foyer region has been scanned. Next automated cartography algorithm seeks the next "inferred" edge for exploration. This edge is near the starting position (position 1). So Yamabico moves from position 6 down the centerline of the hallway to position 7. It stops at position 7 due to the presence of a barrier. The derived partial world matches a hand measured map of the same world space within 25 cm for all extracted line segments. With additional experiments and program tuning this error value could be further reduced.

D. SUMMARY

Reliable path tracking control using path elements has been experimentally tested. In no case did Yamabico deviate significantly from the expected path. The maximum observed deviation was less than two centimeters. Vehicle dead reckoning error correction has been experimentally verified. These experiments used Yamabico's side scanning sonars to detect static landmarks in the world space. An algebraic comparison between the expected and actual landmark configuration was used to determine dead reckoning error. Errors were small and consistent. Yamabico's configuration after a each odometry correction was within one centimeter of the correct configuration which indicates dead reckoning error correction works well with a precision limited by sensor resolution.

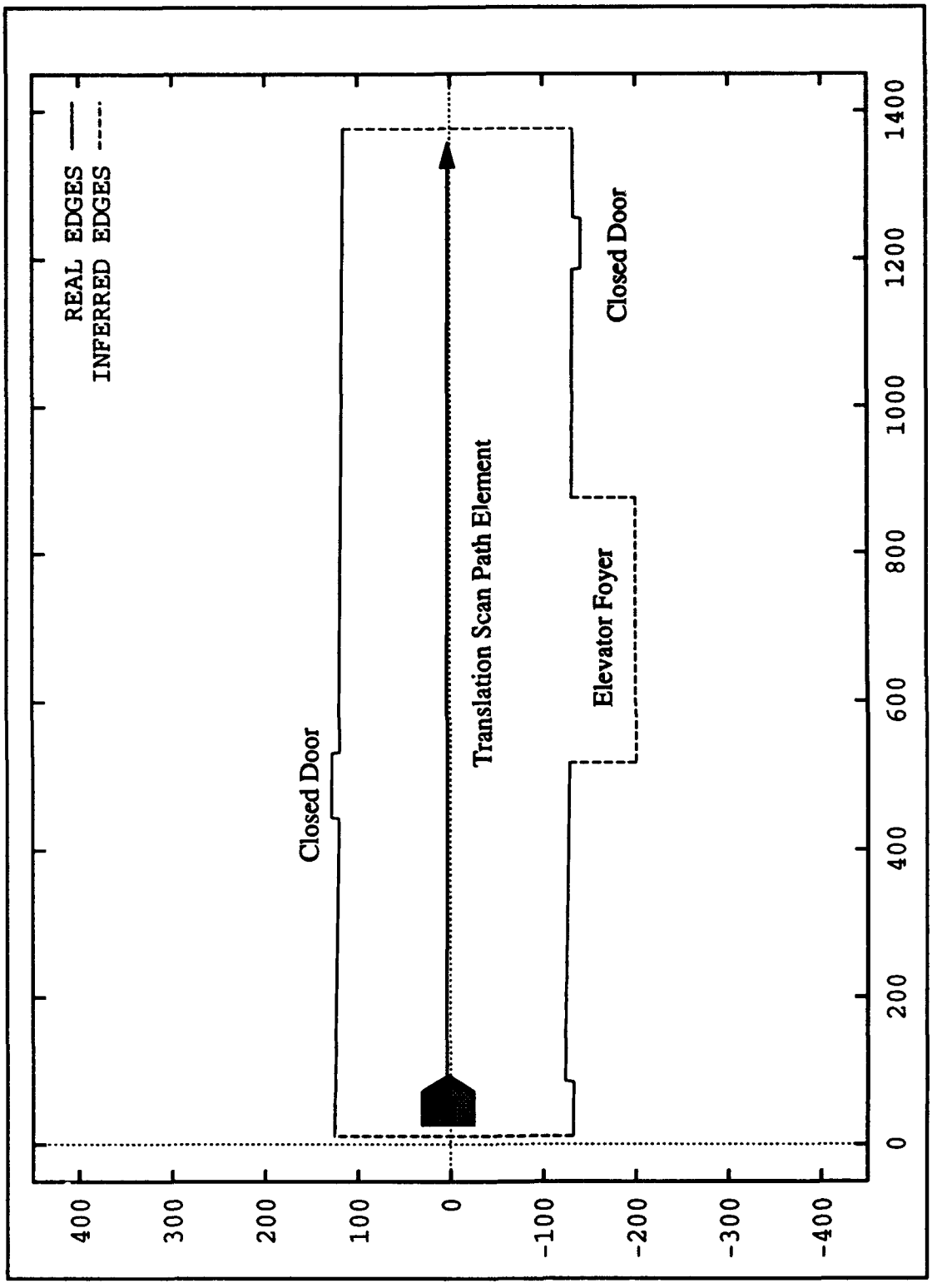


Figure 9.12 Single Translational Scan Experiment

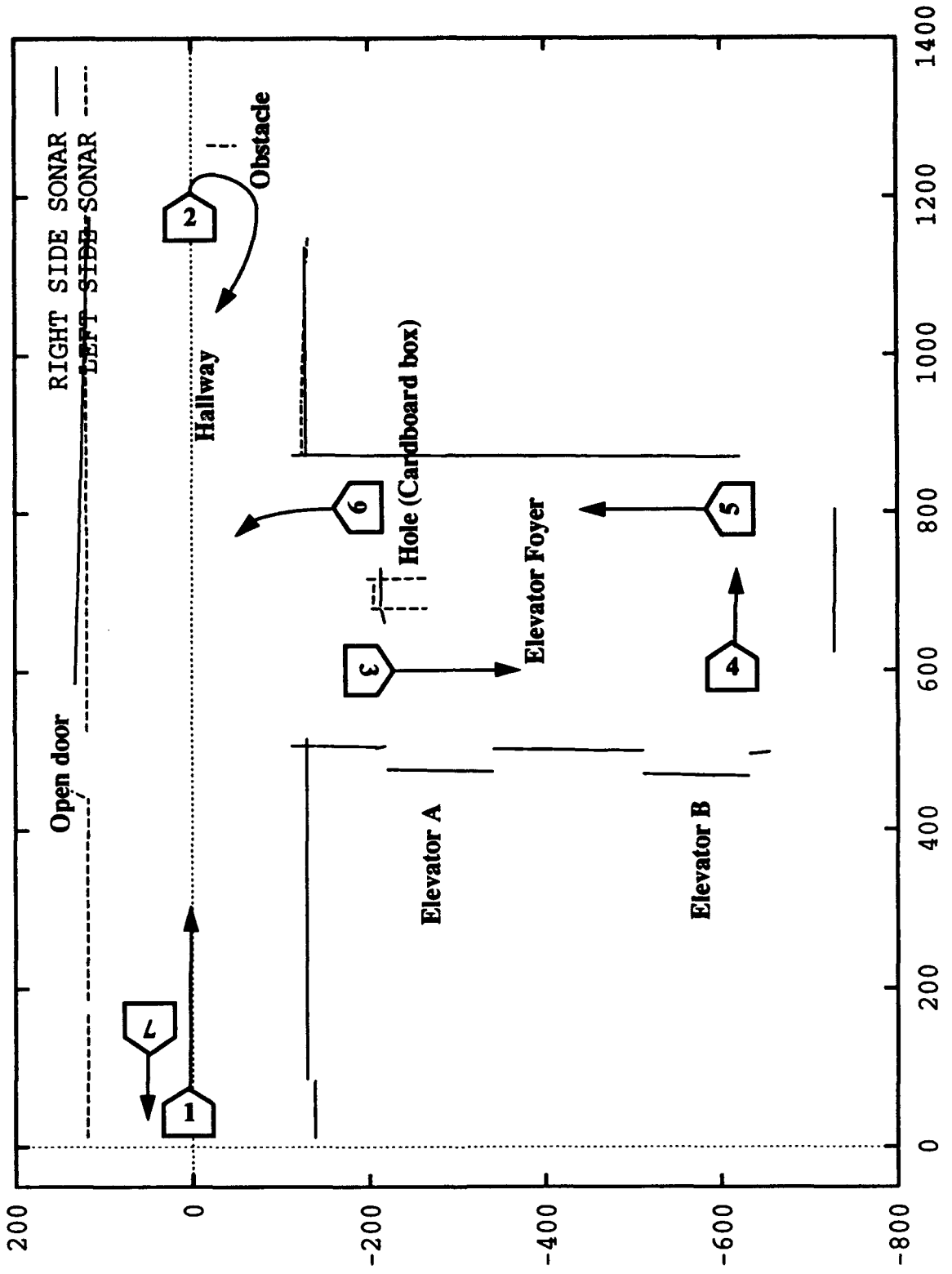


Figure 9.13 Robot Cartography Experiment

The automated cartography experiments demonstrated simultaneous dead reckoning error correction and world space mapping. The *find_orthogonal_orientation* function reliably aligns Yamabico's coordinate system orthogonal to the world space. Maps built by Yamabico using the automated cartography algorithm closely match hand measured maps.

X. CONCLUSIONS

In this dissertation, a complete automated cartography system for an autonomous mobile robot is defined, instantiated, and evaluated. The component parts of this system are path tracking control, dead reckoning error correction, world space exploration, and map representation using a polygonal world model. MML provides the mid-level vehicle control commands necessary for smooth dead reckoning error correction and obstacle avoidance using geometric path elements. MML also provides the necessary sonar, odometry correction and input/output functions for mobile robot experiments. The vehicle odometry correction provides the periodic dead reckoning error correction needed for precise cartography. The automated cartography algorithm controls robot motion so all accessible areas of the world space are examined by the robot's sensors. The algorithm provides the constructs for proper map representation as the world model is built.

A. SUMMARY OF CONCLUSIONS

A series of algorithms for ideal automated cartography are developed. These algorithms are used to study the structures for map representation for cartography as well as the necessary robot motion rules for world space exploration. The correctness of each algorithm is proved. The complexity of these algorithms was also examined to ensure cartography could be accomplished in a reasonable amount of time.

Successful robot cartography experiments using a real autonomous mobile robot are conducted for this dissertation. Abstract 2D maps of a world space are built by Yamabico-11. The features of this map closely match maps built from hand measurements. The automated cartography experiments prove that ultrasonic sonars can be used exclusively to build useful maps of an indoor orthogonal environment. Despite data scatter and odometry error, Yamabico builds a fairly good map of its environment from sonar data.

A 2D transformation algebra for analyzing mobile robot motion is developed in this dissertation. This algebra is used to develop an abstract, general-purpose means of mobile robot localization and odometry correction. This method is a simple application of group theory that requires very little computational overhead. Dead reckoning error correction in real-time is experimentally verified using Yamabico-11. These experiments show that odometry error can be held to a small, relatively constant value despite repeated vehicle motion. In the worst case, average odometry error is 2.54 centimeters in distance and 1.04 degrees in orientation over a 914 centimeter course. The multiple landmark experiment proves that the odometry correction algorithm works well even with sparsely placed landmarks. The algorithm is robust, as shown by continued odometry correction despite landmarks occasionally being missed. Periodic, automatic odometry correction allows for sustained autonomous robot operation that is not limited by dead reckoning error buildup. This automatic, low-overhead dead reckoning error correction task is implemented in the MML software system as one of the available functions that can be activated by a function call. This capability is extremely useful when a sustained robot motion is required. Dead reckoning error correction using abstract algebra supports automated cartography.

The MML motion subsystem is an effective and abstract method to specify robot motion. Short, simple command listings provide surprising complex robot motion. Experimental results from path tracking tests on Yamabico-11 demonstrated precise mobile robot control. The smooth robot motion resulting from the path tracking technique reduces wheel slip, thus improving the overall robot odometry. The path element concept allows higher level command modules to specify robot motion using a short list of path element commands. Path elements allow a user or an artificial intelligence application to command the robot to execute a wide range of motion. Path tracking locomotion control using MML supports automated cartography.

B. CONTRIBUTIONS TO MOBILE ROBOTICS

This dissertation describes software and hardware developments that enable an autonomous mobile robot to perform automated cartography using ultrasonic range finders. This capability is limited to orthogonal, indoor regions. There are several important components required to perform this task. The capability to move in a purposeful manner based upon sensor input is required. Leonard used a preprogrammed "seed-spreader" pattern to control robot motion for gathering sensor data [Leonard 92]. This means some *a priori* knowledge of the shape of the world space was used to command the robot to move about in the appropriate pattern. This author believes that true automated cartography requires the robot's exploratory path to vary with the size and shape of the area being mapped. Tight coupling of the sensor input and the vehicle motion is therefore required. The algorithm used for automated cartography in this dissertation performs automated world space exploration. Using this algorithm, the robot scans all accessible portions of the world with its sensors in order to perform the cartography. Since sensor input and robot motion are tightly coupled, no prior world space knowledge is needed to perform cartography.

Robot motion control using a new path tracking algorithm is developed for controlling robot motion during the world space exploration phase. This control algorithm is described in detail in Appendix A. This technique allows Yamabico to perform odometry corrections in a very smooth and natural fashion. Odometry corrections are performed with respect to a geometric path instead of a single robot configuration. Periodic odometry estimate corrections using landmarks on the robot's map allow for sustained robot operations with small, constant positional error.

As a robot moves about exploring its assigned space, dead reckoning errors build up. The ability to perform localization using naturally occurring landmarks in a typical indoor area is another important contribution of this dissertation. While other authors have demonstrated localization [Leonard 92][Nehmezow 92][Cox 89][Crowley 85a], this dissertation extends this capability to automated landmark recognition. In other words, not only does the robot have the ability to self-correct dead-reckoning errors using external land-

marks, it also recognizes and catalogs landmarks appropriate for subsequent odometry correction. Vehicle configuration calculations using the abstract algebra described in Chapter V greatly simplify robot odometry error correction coding.

Since robot exploration of a world space requires robot motion and since this motion induces dead reckoning error, mapping and error correction must be accomplished together. The challenge of truly autonomous navigation lies in the fact that map building and odometry correction must be undertaken simultaneously [Leonard 92]. This dissertation develops both of these capabilities in parallel in order to allow the robot to build precise maps. All previous work has focused on either one or the other of these two tasks. This author knows of no instance where map building competence and odometry correction have been performed simultaneously. Their interdependent nature bears out the fundamental need to develop these capabilities concurrently.

C. SUGGESTIONS FOR FUTURE RESEARCH

This dissertation research discussed the development of one system to perform automated cartography in an indoor setting. Limiting this system to one sensor type placed severe, unrealistic restrictions on the robot's cartography ability. Additional sensors added to Yamabico-11 such as vision, tactile sensors, and infrared range finders are needed to handle cartography in a more cluttered indoor environment. Sensor sweeps of a 2D plane during translational scanning could possibly extent robot cartography to build 3D maps of indoor spaces.

Improved robot mobility such as the ability to open doors and climb stairs would greatly improve robot cartography capability. Research on a special robot morphology is needed to map ventilation ducts and crawl spaces in buildings. In naval applications, cartography can be extended to inspections of void spaces and tanks on naval ships. These spaces periodically require hazardous human inspections for corrosion and loose fittings.

A voice interface is currently under development to provide a more intuitive robot/human. This system will enable Yamabico to receive voice commands. Additionally, voice re-

sponses by the robot would make it a more useful platform. Voice commands such as "go remap region B" could make Yamabico a good assistant to a human making a floor plan of a building.

The improved utilization of ultrasonic sound is an important extension that hold a great potential. The existing sonars on Yamabico could be better utilized if the diagonal sonar elements were employed to examine hard-to-reach areas such as concave corners. Vastly superior sonar utilization in the animal kingdom tells us that much of the resolution and capability of 40 KHZ sound waves in air is significantly under-utilitized [Nachtigall 86]. The investigation of frequency modulated sonar sensors in air is a useful extension to this dissertation. A software variable range gate and pulse length could significantly enhance the capability of ultrasonic sonar as a sensor. A shorter range gate is more appropriate in a close, cluttered part of a building whereas a long range gate and a longer pulse interval work better in open spaces. An automatic, adaptive sonar range gate and pulse width could significantly enhance cartography.

A more powerful on board computer is desired to allow Yamabico-11 to perform faster processing. The video image processing is desired for navigation and object recognition in a cluttered indoor area. This is a computationally demanding task. A SPARC4 single-board supercomputer is currently being installed to support faster computations [Ironics 91]. A significant portion of the robot's hardware device drivers are being rewritten to support the new central processing unit.

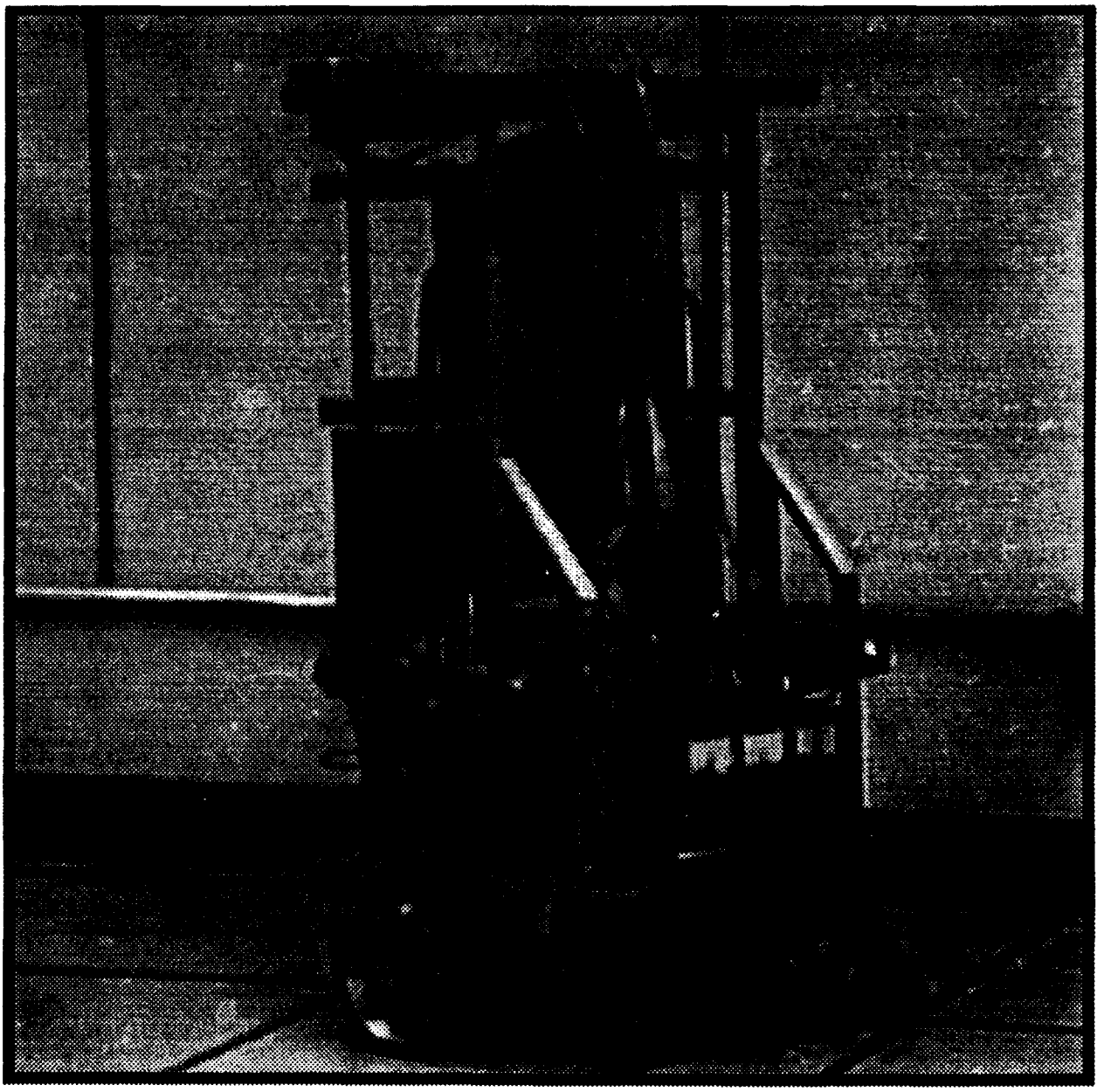
A genetic algorithm approach to world space exploration using the Yamabico simulator to evolve a good exploration strategy in different world spaces might be worthwhile. The evolved cartography algorithm would need to be validated on the robot. This would require an extremely accurate simulator sonar model to predict sonar returns, the simple ray tracing model used in the current simulator is insufficient for this task. Lastly, specular reflection would have to be modeled more completely.

This dissertation has addressed the problem of automated cartography in autonomous mobile robots. The experimental results presented here are encouraging and show that the

automated cartography is suitable for the intelligent robot exploration of an indoor world space. Future robots will undoubtedly draw 3D models of existing buildings by exploring their interiors with sophisticated sensors. This dissertation is one step on the road to this goal.

APPENDIX A. YAMABICO USER'S MANUAL

YAMABICO USER'S MANUAL



INTRODUCTION

Yamabico-11 is an autonomous mobile robot powered by two 12-volt batteries and is driven on two wheels by DC motors. These motors drive and steer the wheels while four shock absorbing caster wheels balance the robot.

The master processor is a MC68020 32-bit microprocessor accompanied by a MC68881 floating point coprocessor. (This is the exact same CPU as a Sun-3 workstation). This processor has one Mbyte of main memory and runs with a clock speed of 16MHz.

All programs on the robot are developed using a Sun 3/60 workstation and UNIX operating system. These programs are first compiled and then downloaded to the robot via a RS-232 link at a baud rate of 9600. The software system consists of a kernel and a user program. The kernel is currently 65,000 bytes and only needs to be downloaded once during the course of a given experiment. A user program "user()", can be modified and downloaded quickly to support rapid development. An on board laptop console is provided to accomplish command level communication to and from the user.

Twelve 40 kHz ultrasonic sensors are provided to allow the robot to sense its environment. The sonar subsystem is controlled by an 8748 microcontroller. Each reading cycle takes approximately 24 msec. Yamabico is used as a testbed for the development of MML, the model-based mobile robot language

EXPLANATION OF FILES

This **file is a framemaker document in new_mml/mml/macpherson/users1.doc** in the yamabico account.

A. Files (n/gemini/work2/yamabico/new_mml/mml10)

[Makefile] Unix makefile for the real-time program.

B. Source Code files

control.c - Tracking feedback control system of vehicle position. Each vehicle control cycle this module reads both optical wheel encoders and determines how to steer the vehicle based upon the current path element and the vehicle's odometry estimate.

cst.h - cubic spiral lookup table (angle [degree] vs. length [cm]).

geom.c - geometric and temporal functions ("set_rob", timer etc.).

init.s - initialize hardware, exceptions.

interrupt.s - Interrupt handler for sonar, timer. This assembly language module is the MML scheduler for the multi-tasking system.

intersection.c - Computes the intersection of sequential path elements in real time.

immediate.c - Immediate locomotion functions that change global locomotion variables.

main.c - main program is linked to "user.c" for real-time execution and for simulation.

Makefile - Makes the kernel and user objects in the UNIX environment.

math.s - Assembly code floating point mathematical functions.

mml.h - MML header file for external variables and constants.

motor.s - Assembly code for drive DC motors.

rosyio.asm.s - Assembly code for conversion of internal codes to ASCII etc.

rosyio.c - input and output through console terminal.

sequential.c - Sequential locomotion functions that load the instruction buffer.

sonar.c - Functions on sonars. Described further in section XXX.

track.c - real-time calculation of reference data as posture, velocity etc.

leaving-point.c - Calculates the leaving point between sequential path elements.

user.c - An application program written by a user. This file consists of the user.c commands to the robot.

C. Object modules

The object modules are the actual executable machine code that is downloaded to Yamabico.

kernel* - system kernel (load address 304000 ~ 30d000).

user - (load address 330000 ~).

D. Application Programs

sonartest3.out - allows user to test all sixteen sonar transducers in groups of four. To load the program type ***lo=dload sonartest.out***, when the program is loaded type "***g 304000***" to start it running, then select 0,1,2, or 3 to indicate which logical sonar group you desire to test. Logical group 0 consists of sonars 0,2,5, and 7; group 1 of sonars 1, 3, 4 and 6; group2 of sonars 8, 9, 10, and 11; and groups 3 is a "virtual" group which consists of four permanent test values. The program displays to the screen the detected range to the four transducer in the group, type e to exit or 0,1,2,or 3 to select another group of sonars.

ROBOT OPERATING INSTRUCTIONS

A. Compiling a program

[Compiling procedure for the real-time program]

(1) Establishing the Yamabico environment.
Login the yamabico account.
Execute "dm9" and go to the "mml9" directory.
The prompt "[mml9]:" will be displayed on the screen.

(2) Create or modify "user.c" by the editor.

(3) Key in "mu" (alias for "make user").
Makefile takes care of all the system files modified.

(4) If the message "kernel relented at 0x00304000" is output on the screen, the kernel should be downloaded at the later stage.
Otherwise only user file has been edited and needs to be recompiled.
See (2) of [Download the kernel & user].

B. Robot setup procedures

(1) Turn on the power switch in the front panel of Yamabico leftside down.

(2) Confirm the voltage of battery, which should be indicating between 23 to 30 volts. If the voltage is below 23 volts, recharge the robot's battery. See ROBOT BATTERY AND POWER SUPPLY PROCEDURES below

(3) Connect the serial port cable (blue connector) from Sun3, the baud rate is 9600 bps.

(4) Press the reset button at the top of Yamabico.

C. Execution of users program

(1) Download the kernel & user

(a) Via the serial port, the kernel and/or the user program is downloaded. When "7920BUG>" is displayed on the console of Yamabico, key in "lo=dluk" + <cr> then "dluk" will be echoed back and the system will start to download the kernel and user.

(b) If other messages are displayed or no response, press the reset
button on the VME board (the reset button on top of the robot does the same function) and try again, or check the cable. If you want to load only user program, key in "lo=dlu". It is important, for it takes about 6 minutes to download the kernel, but 30 seconds for user, so if you modified only user program, it is better to load only user program.

(2) Execution of the user program

Type "g 304000" + <cr>, then the user program will start to execute. The same program stays in the robot's memory and can be restarted with the same command.

[Cabling Instructions]

(a) Blue 4 pin connector(RS232) is for program load and data upload.
(b) Black many-pin connector is terminal hookup.
(c) An additional program load cable is located in the floor by the door to Sp 511 Lab. This allows the programmer to download programs while Yamabico is in the hallway.

[Placing the robot up on the blocks]

(a) Place Yamabico in the desired position.
(b) Carefully lift the front of the robot and place the wooden block underneath.
(c) Carefully lift the back of the robot and place the wooden block underneath. Ensure the robot's two large drive wheels do not touch the ground. Get some assistance if possible.

D. Laptop procedures

(1) The black, many-pin connector is terminal hookup. This can be connected to the vt220 terminal on the tabletop to the right of Yamabico or can be directly connected to the laptop terminal on top of Yamabico. The MacIntosh Powerbook145 is the current laptop terminal. Please read the owner's manual prior to operating this computer. A Voice Navigator voice interface system is also available for voice recognition experiments.
(2) When "7920BUG>" is displayed on the laptop screen this means that the terminal is functioning normally.
(3) If "7920BUG>" is not displayed, the user must set up the terminal by starting the communications software package. Consult the software manual for further guidance.

E. Battery charging and power supply procedures

ROBOT OFF
(1) The MAIN SW circuit breaker on the Robot power panel is OFF.
(2) The BATTERY circuit breaker on the Robot power panel is OFF.
(3) The External Power Supply is switched OFF.
(4) The Battery Charger is unplugged.
(5) The Battery Charger connector is disconnected from the Robot.

ROBOT OFF, CHARGING BATTERIES
(1) Make sure the BATTERY circuit breaker on the Robot power panel is OFF.
(2) Connect the Battery Charger output connector to the robot.
(3) Plug the Battery Charger into the AC outlet.

ROBOT POWERED FROM EXTERNAL POWER
(1) Make sure the BATTERY circuit breaker on the Robot power panel is OFF.
(2) Connect the External Power Supply output connector to the adapter on the Robot power panel.
(3) Turn the External Power supply ON.
(4) Turn the MAIN SW circuit breaker on the Robot control panel ON.
(5) You may leave the Battery Charger connected and operating. The Robot will not load the batteries.

SWITCHING THE ROBOT FROM EXTERNAL POWER TO BATTERY POWER
(1) Unplug the Battery Charger from the AC outlet.
(2) Disconnect the Battery Charger output connector from the Robot.
(3) Turn the BATTERY circuit breaker on the Robot power panel to ON.
(4) Turn the External Power Supply OFF.
(5) Disconnect the External Power Supply output connector from the adapter on the Robot power panel.

SWITCHING THE ROBOT FROM BATTERY POWER TO EXTERNAL POWER

(1) Connect the External Power Supply output connector to the adapter on the Robot power panel.
(2) Turn the External Power supply ON.
(3) Turn the BATTERY circuit breaker on the Robot power panel to OFF.
(4) Connect the Battery Charger output connector to the robot.
(5) Plug the Battery Charger into the AC outlet.

ROBOT SIMULATION PROGRAM

A. Files in Simulator

yam_sim - The graphic simulator menu for selecting simulator top level commands.

sim - The executable simulator produced by the makefile. The compile menu button updates sim and the run button runs the sim.

sim_info - Output of reference posture by the simulation program.

axis_data - Data for "gnuplot" program.

B. Operations for the Simulation Program

(1) **Compiling procedure for the simulation program.**
(a) Login MML working directory.
aquarius login: **yamabico** (You may also use pegasus for gcc compiler.) Read the file AAAREADME for any new simulator developments.

(b)If the prompt is displayed on the screen "yamabico@aquarius%", key in "ys".
If logged in MML working directory, the prompt "[mml]:" is displayed on the screen.

(c) Create or modify "user.c" by the using the EDIT button which invokes the "vi" editor. If you desire to use "emacs" or some other editor, then open a separate window and edit "user.c" using the editor of your choice.

(d) When you have finished editing the user.c file, save the file and compile it by clicking the left mouse button on the CMPL button on the command menu. This executes the command make sim which cause the Makefile to recompile all files in the simulator that have been edited including user.c.

(2) **Execution of the simulation program**
(a) Click the left mouse button on the RUN button and the simulation program will start.

(b) At first, the program will fill the instruction buffer and then the graphics portion of the program will display the robot trajectory on the fifth floor of Spanagel Hall.

(c) When the simulation starts, the program will display the message and the instruction stack. For example, simulation output:

Instruction Stack:
Class argr1 argr2 argr3 argr4 argr5 argr6 mode1 mode2
SET_ROB 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0 0
STOP 147.717624 0.000004 69.098301 95.105652 1.884956 0.000000 0 0
SET_ROB 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0 0
STOP 125.656116 0.000006 58.778525 80.901699 1.884956 0.000000 0 0

Total Number of Instruction : 4

(4) When the program has finished, gnuplot is automatically called to plot the robot's whole trajectory to the screen or to the "ss1" printer. The robot's trajectory information is stored in the axis_data file that is built as the robot simulator runs.

PROGRAMMER GUIDE AND EXAMPLES

Motivation

The technique of path specification control is a fundamentally new way of specifying Yamabico motion commands. Previously, all robot motion commands were specified as configurations consisting of $p = (x, y, q)$. In other words, the robot was commanded to move from point to point with the requirement that it attain the orientation q at each specified point. This technique has been shown to lead to difficulties when an odometry correction was imposed upon the robot while it was in motion. These difficulties included reverse motion during the *set_rob0* function using the configuration to configuration tracking method and non-smooth motion, resulting in wheel slippage that increased odometry error since the vehicle was accelerated to a higher speed until the new configuration was obtained.

Early experimental work on robot odometry correction and wall following revealed that the *set_rob0* function frequently caused jerky, non-smooth vehicle motion. This problem was particularly prevalent when the new odometry estimate fell behind the robot. An odometry reset to a position behind the vehicle caused the vehicle to back up to attain the correct configuration on the Cartesian plane. The vehicle also was programmed to accelerate to a higher speed than the current operating speed in case the correction required the vehicle to "catch up" to the correction configuration. This acceleration caused increased vehicle wheel slippage that resulted in increased odometry error.

A better way to specify robot motion is by a series of planar path elements that are connected to obtain the robot's desired path. These planar elements can be straight lines, arcs (constant curvature portions of a circle), or parabolic line segments. In this way, the vehicle odometry reset is performed with respect to a planar path vice a single odometry configuration. This allows the robot to smoothly return to the specified path when the odometry estimate is reset using *set_rob0*. No

change in speed is required so the overall vehicle wheel slippage is reduced. This path specification control modification has required some significant modifications to the MML language.

PROBLEM STATEMENT AND GOALS

This section describes the problems that the new version of MML path tracking functions intend to solve. The design goals and the system constraints are explicitly stated.

Initial Problem Statement

The purpose of the Yamabico tracking control system is to allow the Yamabico robot to follow a path specified by configurations, lines, circular arcs and parabolas. The robot must automatically determine the transition point from one path element to the next. This includes recovery from consecutive non-intersecting path elements. Additionally, the robot motion must be smooth. No discontinuity in the robot's motion is allowed. Since the absolute value of $d\kappa/ds$ is always finite, the robot's path curvature (κ) is continuous with respect to the distance travelled (s).

Robot odometry correction is improved by performing moving odometry correction while the vehicle is in motion following a path. Overall vehicle motion is enhanced by using a function to command the vehicle to follow a parabolic path for obstacle avoidance.

Goals

The primary goal of the vehicle path tracking technique is to achieve smooth robot motion. This is accomplished by commanding the robot to follow straight, curved and parabolic line segments as elements of the path.

Automated path tracking includes automatic path element to path element transitions. Robust robot path tracking including exception handling and intelligent error recovery are built into MML to make robot programming easier.

Constraints

Physical limitations on robot motion are a characteristic of the vehicle's size, shape and weight distribution. We must design to prevent such things as very sharp curvature at high speed.

Time is a critical factor in the sense that a finite amount of CPU time is required for the robot to calculate the current path image and determine the best q and κ for each step.This new computer will perform Yamabico's computations approximately 25 times faster than the Sun3 board.

Robot Main memory size is another limitation, currently the robots main memory is five megabytes.The Sparc4 single board computer has 16 megabytes.

Path to path transition planning must take a reasonable amount of time. Successive refinement of the optimum leaving point is used to ensure the best possible transition point is available at any given time. The first rough approximation is the intersection point, next is the transition point within one s_0 of the optimum leaving point and finally the final refined optimum leaving point, determined within one eighth of s_0. An obvious trade-off exists between the time spent planning and the accuracy of the outcome.

The number of Central Processing Unit (CPU) interrupt levels is limited to eight. This limits the number of separate tasks that can run at different levels in the single CPU, multitasking system.

VEHICLE KINEMATICS

Linear Vehicle Motion Control

The Yamabico-11 mobile robot is a power wheeled steering robot [Kanayama 91]. This means that each drive wheel has its own independent motor. Steering is accomplished by variation in the relative speeds of these two motors. Vehicle linear velocity control is provided by the trapezoidal speed versus distance profile shown in Figure A.1. The velocity control algorithm for the vehicle is based on ramped, linear acceleration to a constant nominal velocity (*vel_g*). To move in a straight line to a given point in the work space, the vehicle accelerates at a constant nominal acceleration rate as in region I in Figure A.1. MML provides a function for the user to change the nominal acceleration rate. Once the vehicle reaches the nominal velocity value the vehicle maintains a constant velocity as in region II of Figure A.1. The vehicle maintains this constant velocity unless it is commanded to stop at a particular point by a backward line (*bline*) or *stop* function. In this case the vehicle automatically calculates its distance to the stopping point. When Equation A.1 holds then the vehicle decelerates at a constant rate where d is the remaining path distance to

the stopping point, a is the robot's nominal accleration rate, and v is the robot's current velocity. This is region III in Figure A.1. In region IV of Figure A.1 the vehicle is stopped.

$$2ad \leq v^2 \qquad \text{A.1}$$

The required velocity of the left (v_L) and right (v_R) wheels are calculated by Equations A.2 and A.3.

$$v_L = (1 - W\kappa)\, v \qquad \text{A.2}$$

$$v_R = (1 + W\kappa)\, v \qquad \text{A.3}$$

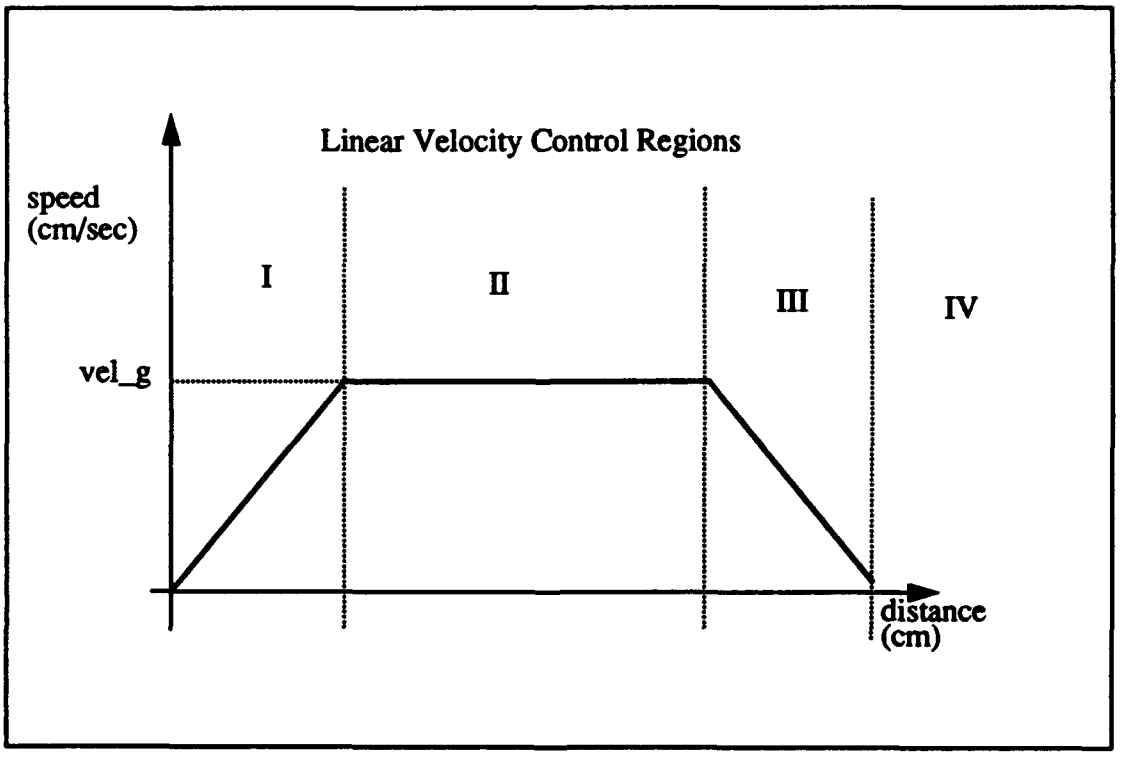


Figure A.1 Vehicle Speed Control

Once the desired linear velocity is determined, the vehicle image on the intended path element is calculated. The nature of the image calculation depends on the path element type. The value of the instantaneous path curvature, κ, is determined using the value of the size constant s_0. Based upon the Equations A.2 and A.3 the left (v_L) and right (v_R) wheel velocities are calculated, where W is the distance between the wheels and κ is the vehicle's instantaneous path curvature. The desired left and right wheel velocities are converted into an appropriate pulse width modulation (*pwm*) value using an empirically derived approximation function. The *pwm* values for the left and right wheels are stored in a 16 bit motor control word (*mcw*). This *mcw* is sent as a command to the left and right wheel motors every vehicle control cycle.

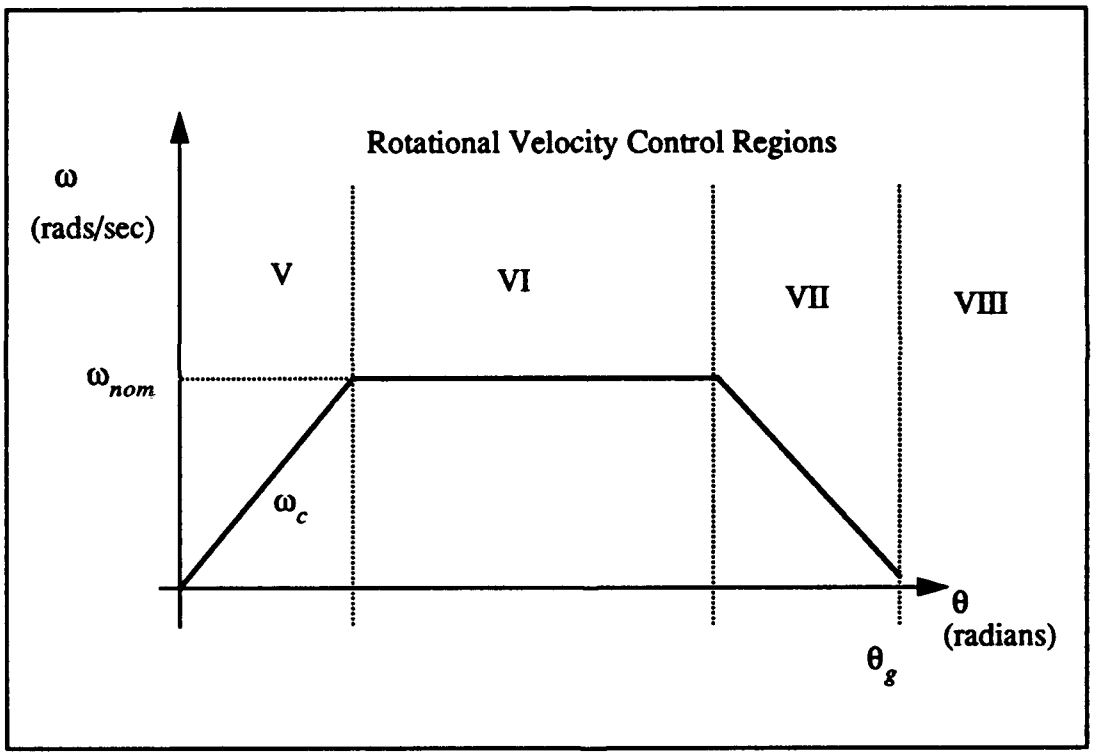


Figure A.2 Vehicle Rotational Speed Control

Rotational Velocity Control

Vehicle rotational velocity control can be initiated only when the vehicle is in a stopped state. To begin rotation the vehicle must be in the stopped state in order to prevent possible wear or damage to the robot's drive train gears. The power wheel steered robot has a translational motion

state and a rotational motion state. The method of speed control is the same as for linear velocity control. The speed control is illustrated in Figure A.2. The rotational velocity control description is provided for completeness. The robot's current rotational velocity ω_c is calculated each vehicle control cycle by the odometry software. In region V, the robot starts out a with $\omega_c = 0$ and accelerates to the nominal rotational velocity ω_{nom}. In region VI, the robot rotates at a constant angular velocity until the vehicle's orientation approaches the goal orientation θ_g. When the robot orientation satisfies Equation A.4 , the vehicle enters region VII and begins a ramped decrease of the current rotational velocity θ_c where a_{rot} is the robot's current nominal rotational accleration.

$$\omega_c^2 < 2a_{rot}(\theta_g - \theta_c) \qquad \text{A.4}$$

In region VIII, the robot's current rotational velocity is reduced to zero when it reaches θ_g and the robot's goes back to the "stopped" state, ready to perform more rotations or translation motion functions.

VEHICLE CONTROL ALGORITHM

The method of low level vehicle control is described in this section. The MML scheduler is an assembly language routine that calls the appropriate interrupt handler based upon a serial board timer or an external interrupt to the main central processing unit (CPU). Figure A.3 illustrates the MML task scheduler. This section gives a detailed description of the locomotion task which is executed every vehicle control cycle (100 Hertz). The high level locomotion task algorithm is given in Figure A.4 . This algorithm runs every vehicle control cycle. Each step of this algorithm is described in the rest of this section.

Vehicle Odometry

The vehicle odometry function determines a new dead reckoning configuration for the vehicle during each vehicle control cycle. This information is required to determine the proper steering commands for the vehicle. The distance each wheel has moved during the last vehicle control cycle is computed by reading the storage register for each of the optical wheel encoders and

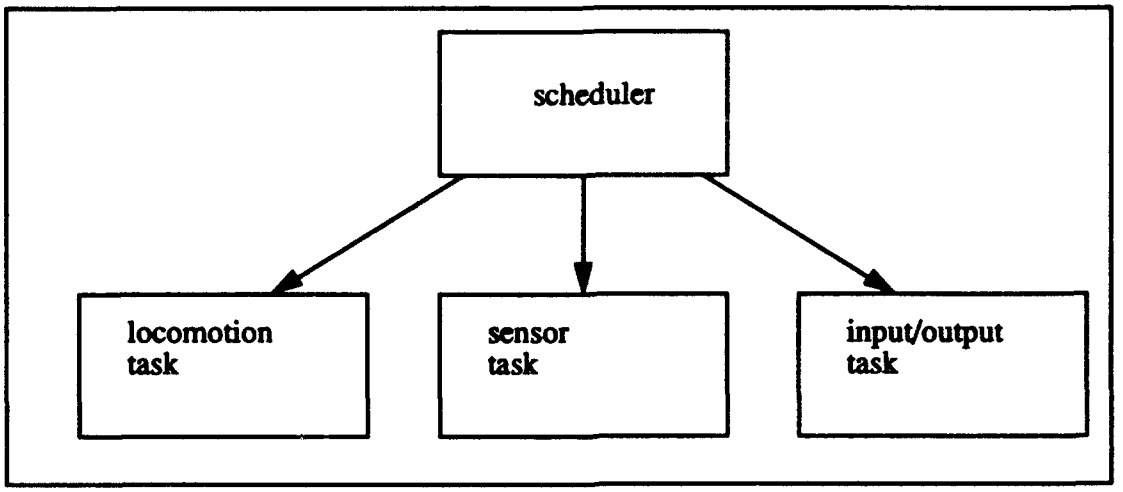


Figure A.3 MML Scheduler

```
locomotion task()
{
    perform vehicle odometry
    store location trace data
    calculate commanded vehicle velocity
    calculate commanded vehicle image and kappa
    calculate vehicle v_L and v_R based upon kappa and velocity
    calculate pwm and mcw
    return pwm, mcw
}
```

Figure A.4 Locomotion Task Algorithm

multiplying this value by an encoder-to-distance conversion factor. The wheel encoders record a full wheel rotation in 512 discrete steps. This value is then filtered using a recursive digital filter [Hamming 83].

The vehicle's instantaneous change in orientation, $\Delta\theta$ is computed by the Equation A.5

$$\Delta\theta = \frac{\Delta s_R - \Delta s_L}{W} \qquad \text{A.5}$$

where Δs is the signed incremental distance each wheel moved as determined by reading each wheel encoder and W is the vehicle's effective tread width. The vehicle's instantaneous distance traveled, Δs is computed by taking the average of the incremental distance traveled by the left and right wheels. The value of Δs is determined as shown in Equation A.6.

$$\Delta s = \frac{\Delta s_R + \Delta s_L}{2.0} \qquad \text{A.6}$$

The vehicle's current translational velocity (v_c) and rotational velocity (ω_c) are determined next by the Equations A.7 and A.8 where Δt is the time interval between vehicle control cycles.

$$v_c = \frac{\Delta s}{\Delta t} \qquad \text{A.7}$$

$$\omega_c = \frac{\Delta\theta}{\Delta t} \qquad \text{A.8}$$

Finally, the vehicle's current dead reckoning configuration $q_1 = (x_1, y_1, t_1, k_1)$ is computed by the next function, using the vehicle's last dead reckoning configuration $q_0 = (x_0, y_0, t_0, k_0)$ where Equations A.9, A.10, A.11, and A.12 are used to compute the new robot configuration.

$$x_1 = x_0 + \Delta s\,(\cos\theta + \frac{\Delta\theta}{2}) \qquad \text{A.9}$$

$$y_1 = y_0 + \Delta s\,(\sin\theta + \frac{\Delta\theta}{2}) \qquad \text{A.10}$$

$$\theta_1 = \theta_0 + \Delta\theta \quad \text{A.11}$$

$$\kappa = kappa \quad \text{A.12}$$

Vehicle Location Trace

The vehicle's current DR configuration can be stored in the robot's on board memory using the location trace feature of MML. This is an user controlled option. If the location trace flag is switched on by the user, the vehicle's current odometry configuration is written to the robot's memory every *n* vehicle control cycles. The *n* value is user selectable. The *location_trace_dump* function allows the user to send the stored robot trajectory data back to the host computer at any point in the program [Sherfey 91].

Vehicle Commanded Velocity

For translation motion, the commanded velocity for the left and right wheels is determined next. First the vehicle's current DR configuration and current path element are compared. A vehicle image configuration is calculated by projecting the vehicle configuration onto the current path element [Alexander 93][Abresch 92]. Next the commanded instantaneous vehicle path curvature (κ) is computed using the image, vehicle configuration, and current path element. Finally, the vehicle's commanded rotational velocity (ω_c) is computed by the Equation A.13.

$$\omega_c = \kappa \times v_c \quad \text{A.13}$$

The commanded left and right wheel velocities are computed by the Equations A.14 and A.15 respectively.

$$v_L = v_c - \frac{W}{2} \quad \text{A.14}$$

$$v_R = v_c + \frac{W}{2} \qquad \text{A.15}$$

Determination of Pulse Width Modulation (pwm) Values

The required pulse width modulation (pwm) values for the left and right wheels are determined based upon the commanded wheel velocities according to the Equations A.16 and A.17.

$$pwm_{left} = pwmlookup\,(v_L) + kpwb\,(v_L - v_{CL}) \qquad \text{A.16}$$

$$pwm_{right} = pwmlookup\,(v_R) + kpwb\,(v_R - v_{CR}) \qquad \text{A.17}$$

The *pwm_lookup* function returns the empirically derived *pwm* value for the corresponding commanded wheel velocity. The motor control word (*mcw*) is determined based upon the vehicle's commanded move direction. Normally, this is forward, but during rotation, sharp turns, and backward motion the *mcw* may change to a negative value.

PATH ELEMENTS

We first define how paths are geometrically described in this method. A configuration q stands for a triple

$q = (p, q, k)$, A.18

where p is a point, q an orientation, and κ a curvature. For an arbitrary configuration q and a point p, either one of the following is said to be an *element*:

line(q), *parabola*(p), *forward_line*(q), *backward_line*(q), *configuration*(q)

A *path* is a sequence $(e_1, e_2, ..., e_n)$, where each e_i is an element. The meaning of each element e is defined as follows (each element means a directed simple path if e is not a *configuration*).

1. *line*(q) means a circle if $\kappa \neq 0$ or a line if $\kappa = 0$ (Figure A.5, Part (a)). This path segment does not have any endpoints.
2. *parabola*(p) means a directed parabola determined by the focus p and the directrix q. (The curvature part k of q is ignored.) (Figure A.5, Part (b)).

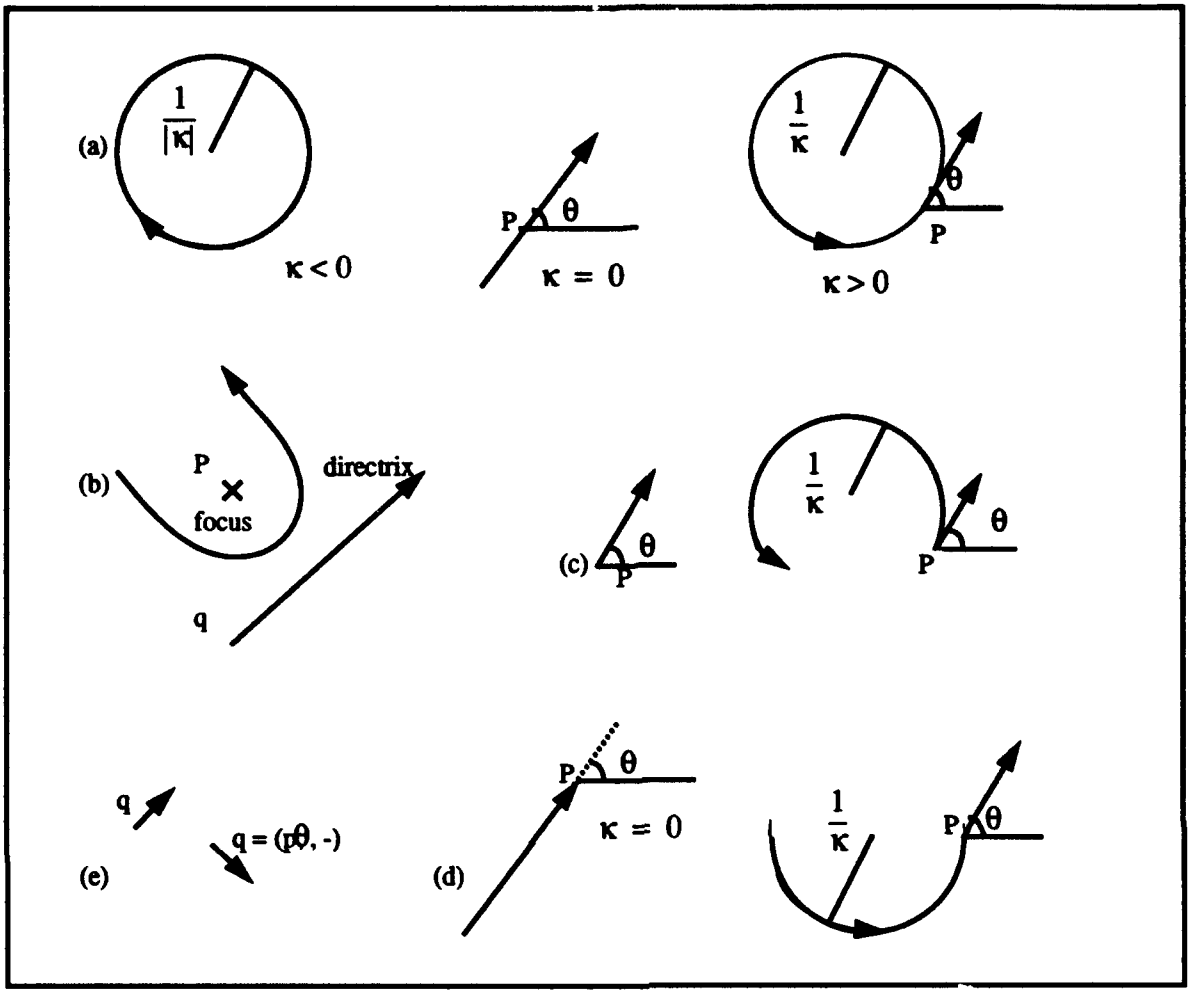


Figure A.5 Directed Path Elements

3. *forward_line*(*q*) means a part of element *line*(*q*). It has a start *p* (Figure A.5, Part (c)).
4. *backward_line*(*q*) means a part of element *line*(*q*). It has an end *p* (Figure A.5, Part (d)).
5. *configuration*(*q*) does not mean a directed path segment by itself. It must have elements which specifies another *configuration* in the previous and the following position in its path. A pair of configurations define a cubic spiral path segment. (Figure A.5, Part (e)) (The curvature part κ is ignored.)

Table A.1 shows permissible combinations for two consecutive elements in a sequence. Each combination is depicted in Figure A.5.

Table A.1 :PERMISSIBLE COMBINATIONS

From / To	line	parabola	backward_line	forward_line	configuration
line	TR (a)	TR (b)	TR (d)	-	-
parabola	TR (c)	-	TR (e)	-	-
forward_line	TR (f)	TR (g)	TR (h)	-	-
backward_line	TRE (i)	TRE (j)	TRE (k)	CS (l)	CS (m)
configuration	-	-	-	CS (n)	CS (o)

The methods of tracking entries in the table are:

TR: normal transition

TRE: transition at the endpoint

CS: cubic spiral

-: not permissible

The finite state machine diagram in Figure A.6 describes the allowable transitions among robot tracking states:

PATH ELEMENT TRANSITIONS

Path elements are the component parts of the robot's intended path. A method for defining how paths are geometrically specified is required. A configuration q as in Equation A.19 is a three element vector where $p = (x, y)$ is a point in the Cartesian plane, θ is the orientation measure counterclockwise with respect to the x-axis, and κ is the path curvature. A parabola is a data structure with a point p which represents the parabola's focus and a configuration q which represents the parabola's directrix as shown in Equation A.20. For an arbitrary configuration q and a parabola r, any one of the following is said to be a *path element: line*(q), *parabola*(r), *forward_line*(q), *backward_line*(q), *cubic*(q).

$$q = (p, \theta, \kappa) \qquad \text{A.19}$$

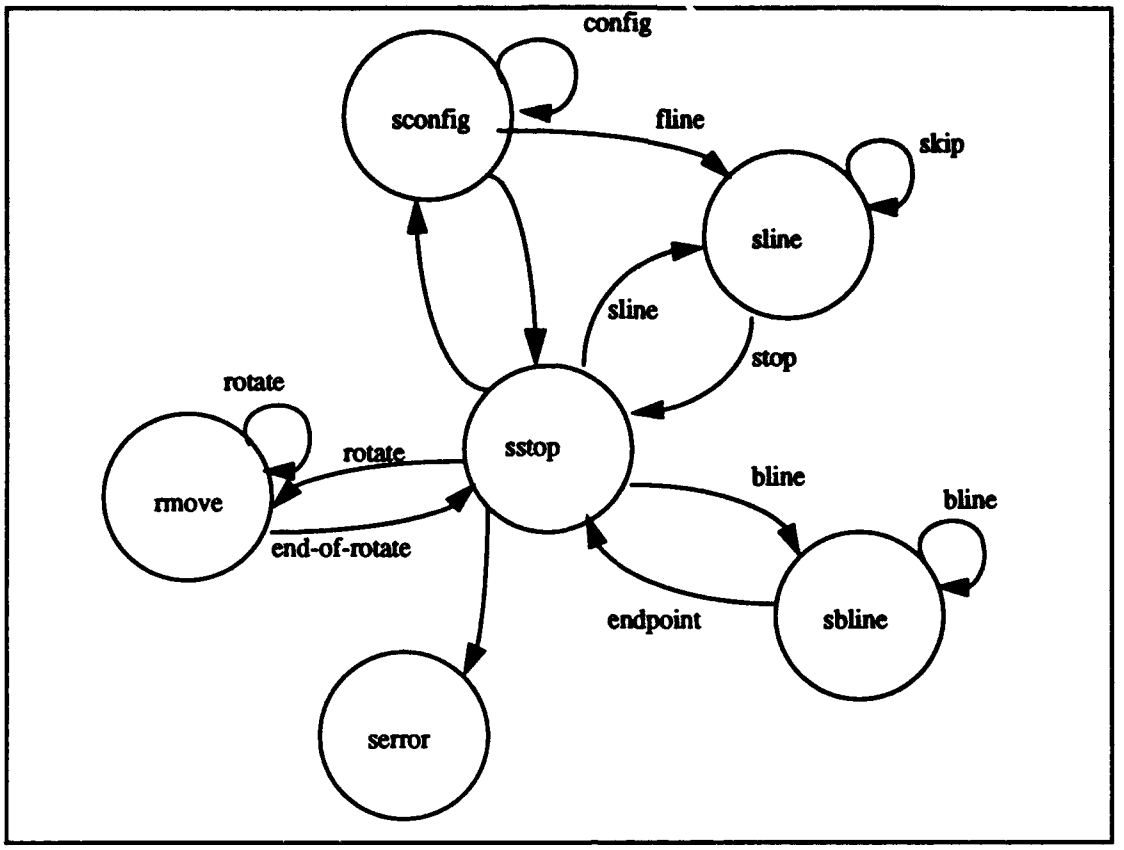


Figure A.6 - Finite State Machine for Robot Status

$$r = (p, q) \qquad \text{A.20}$$

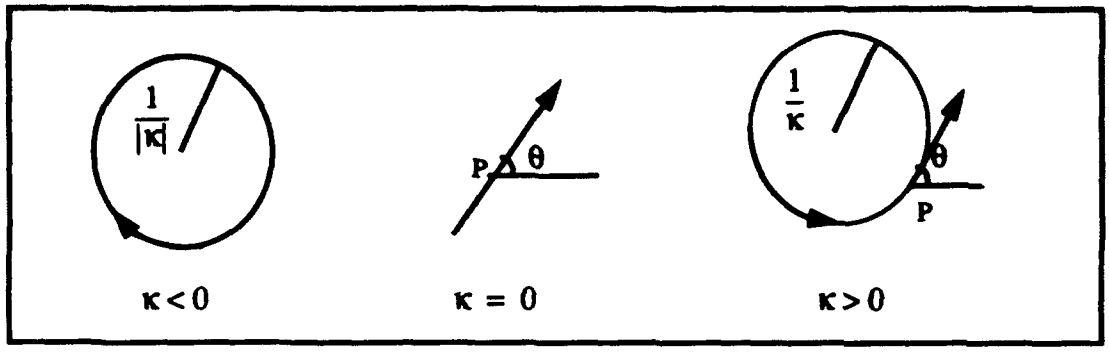


Figure A.7 The Line Function

A *path* is a sequence of *path elements* $(e_1, e_2, ..., e_n)$, where each e_i is an element. The specification for each element *e* is defined as follows: each element means a directed simple path if *e* is not a *configuration.*

The *line*(*q*) function commands the vehicle to track a circular path element if $\kappa \neq 0$ or a straight line path element if $\kappa = 0$ as shown in Figure A.7. This path segment does not have any endpoints. The vehicle tracks the commanded line unless ordered to track some other path element by a subsequent user command. The curvature (κ) of the line path element is constant along the entire path.

The *parabola*(*r*) function commands the vehicle to track a directed parabolic path element as determined by the focus *p* and the directrix *q*. The curvature part (κ) of *q* is ignored. This function is useful for obstacle avoidance, since the world space obstacle can be the focus of the parabola.

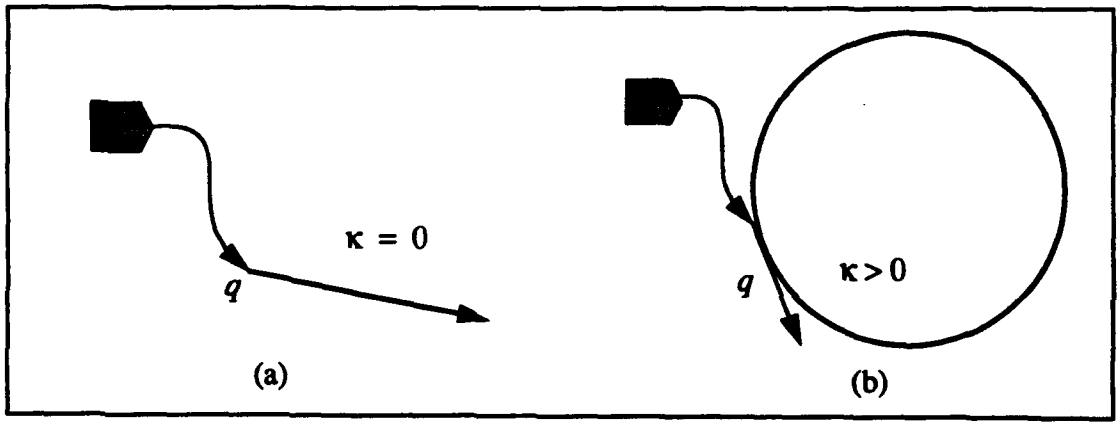


Figure A.8 Forward Line Tracking

The *forward_line*(q) function is a compound command. This command tells the robot to follow a cubic spiral path to the beginning of the directed half-line formed by q. Once the robot reaches the start of the half-line, it tracks this line just as the *line*(q). Figure A.8 (a) illustrates the vehicle automatically using a cubic spiral path to move from its current configuration to the path element q, then the vehicle switches to tracking the straight forward line. Similarly, in Figure A.8 (b), the vehicle uses a cubic spiral to reach the configuration q and then switches to tracking the circular path element specified by q.

The *backward_line*(q) is similar to the forward line function except the configuration q specifies the endpoint of the path element. The vehicle may transition to other path elements after reaching the configuration q if other path element commands follow. If no command follows then the vehicle stops at the end configuration of the backward line. Backward lines may be straight or circular path elements depending on the value of κ in q. Figure A.9 illustrates two backward lines.

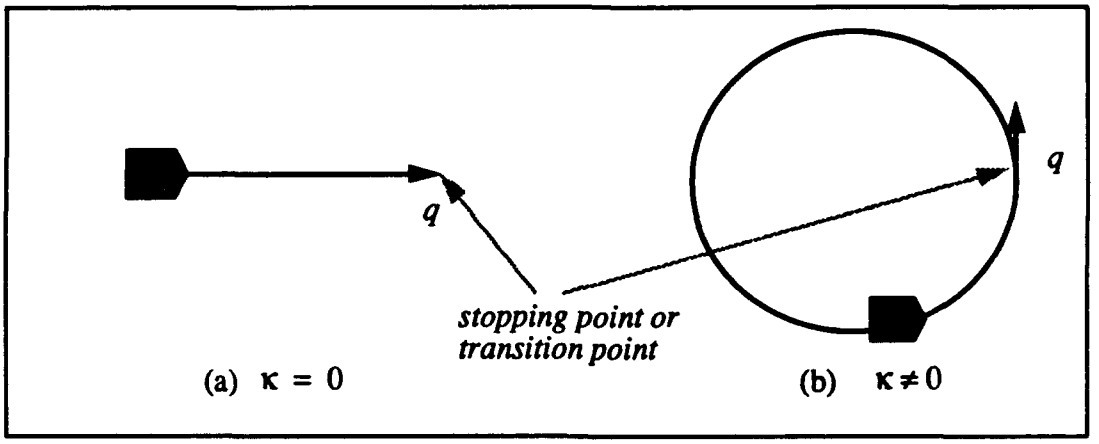


Figure A.9 Backward Line Path Elements

In Figure A.9 (a), the vehicle tracks the straight line specified by q and stops (or transitions to another path element) at the configuration q. In Figure A.9 (b), the robot tracks the circular path q and also stops at the point $(q.x, q.y)$.

The *cubic*(q) function commands the vehicle to move through a specific configuration using a cubic spiral path element. The cubic spiral must have a starting configuration in order to be meaningful. The command uses path elements which specify another *configuration* in the previous and the following position on its path to form the cubic spiral. When *cubic*(q) is the first command, the robot's current configuration is used as the starting configuration for the spiral. In all cases, a pair

of configurations define a cubic spiral path segment. Figure A.10 illustrates the vehicle moving

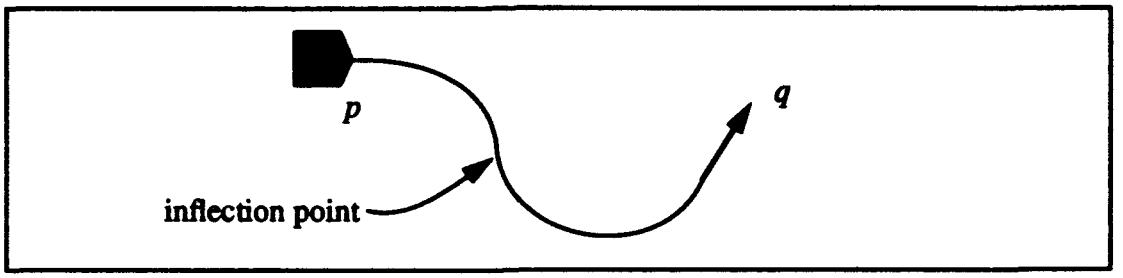


Figure A.10 Cubic Spiral Path Elements

from its current configuration p to q using a cubic spiral path element. The curvature part of the configuration q is not used. Notice that two cubic spirals are actually used with a point of inflection between them.

PATH ELEMENTS TRANSITIONS

This section describes the allowed path element transitions in greater detail. The path element segments are designed to allow an autonomous robotic vehicle to follow a Voronoi path through an obstacle field. To accomplish this, straight lines, circular arcs, and parabolic line segments are required.

Thus far, only path elements that share a common intersection point have been considered. Some applications require vehicle transitions between non-intersecting path elements. One simple example of this is lane changing for obstacle avoidance. Just as automobiles move from one parallel lane to another to avoid slow traffic or an obstacle in the road, the robot "changes lanes" from its current path to a parallel path (either left or right) when necessary to avoid an obstacle. Another application involves motion planning by representing all obstacles as circles as described in [Brutzman 92]. Additionally, line-to-circle, circle-to-line, circle-to-circle, parabola-to-line and line-to-parabola transitions are needed to enhance user program robustness and flexibility. In all of the following transition descriptions, *p1* is the current vehicle path element and *p2* is the next path element.

Straight Line Path Transitions with Intersecting Paths

When two consecutive directed lines, q_1 and q_2 are given, the vehicle leaves the current path element q_1 at a point p_l upstream of the intersection point p_{12}. The optimum *leaving point* P_l must satisfy the condition that the trajectory does not oscillate if it leaves q_1 at p_l and oscillates if it leaves any point closer to p_{12} than P_l ; see Figure A.11. The distance between P_l and p_{12} is called the leaving distance and is proportional to s_o.

For intersecting straight line paths, the intersection point is first determined. Then leaving points on P_1 are selected based upon the intersection point. From the hypothetical leaving points, the robot's path is projected from path element q_1 to path element q_2. A non-oscillating, smooth transition from the first path to the second path is sought. A step value of s_o is used for this process. The first leaving point examined is at a distance of one s_o from the intersection point on q_1. The algorithm steps in s_o increments away from the intersection point along p_1 until the best leaving point is approximated. The best leaving point then is determined to the nearest one eighth of s_o. In Figure A.11, the optimum leaving point is P_l. If the vehicle leaves path q_1 too early (for instance at point P_k), there is less control over the robot's motion. If the robot leaves path q_1 too late, for instance at point P_m, then the robot overshoots the intended path q_2 during the transition. The leaving distance in Figure A.11 is the distance along the path element q_1 between point P_l and point P_{12}. In general, the leaving distance is an increasing function of the difference in the θ values of the two paths at the intersection.

Straight Line Path Transitions with Non-Intersecting Paths

Parallel straight line path elements have no intersection point. A method of leaving the current path and tracking the next path must be specified in this case. When the line command is issued and the vehicle's current path element does not intersect the next path element, the vehicle immediately stops following the current straight line path element and tracks the next parallel element. In other words, the vehicle's image on the current path element is the leaving point to transition to the new path element.

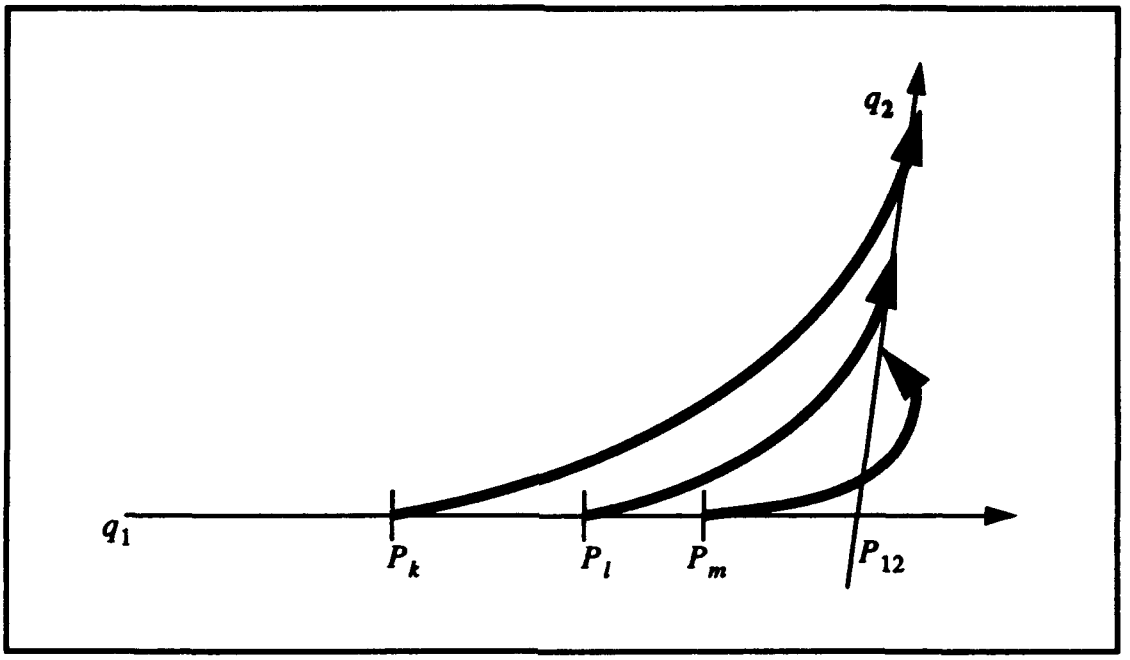


Figure A.11 Transition Between Intersecting Straight Line Paths

In Figure A.12 (a) parallel paths in the same direction give simple lane changing behavior. In this example, the vehicle immediately stops tracking path element p_1 and lane changes to the left to path element p_2 upon receipt of the command *line(&p2)*.

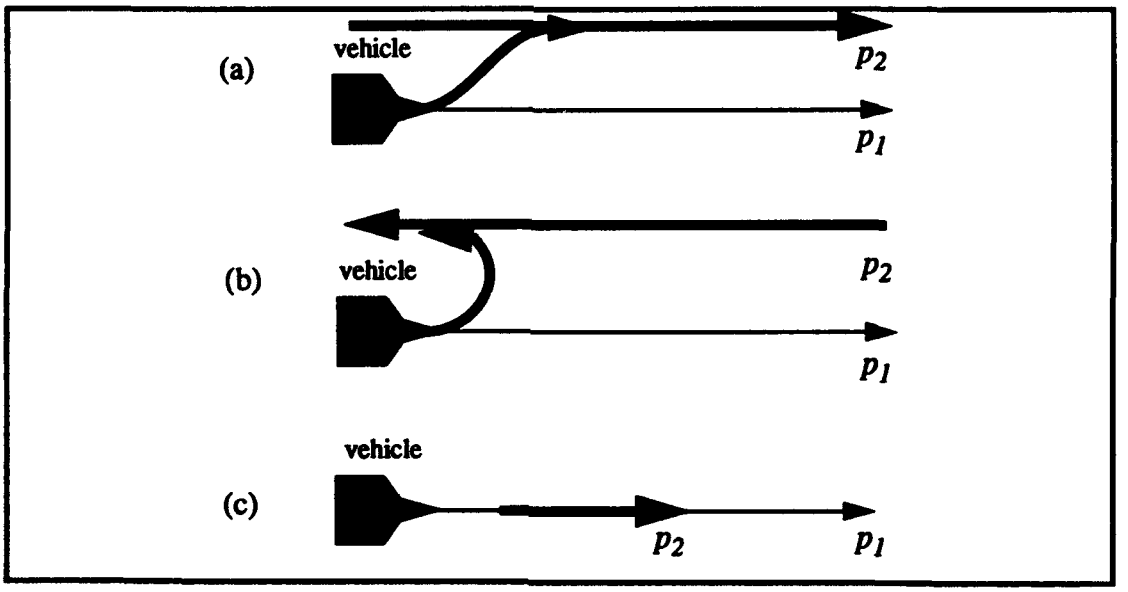


Figure A.12 Transitions Between Parallel Straight Line Paths

In Figure A.12 (b), parallel paths with opposite orientation cause the vehicle to turn around. Initially the vehicle tracks path element p_1. When the command *line(&p2)* is received, the vehicle immediately leaves path element p_1 and tracks path element p_2 since there is no intersection point. Since p_1 and p_2 have opposite orientations, the vehicle turns towards path element p_2 and eventually turns all the way around to track p_2.

In Figure A.12 (c), co-linear paths cause the vehicle motion to be unchanged. Since path elements p_1 and p_2 have the same orientation and are collinear, no net change in vehicle motion occurs when the command *line(&p2)* is received. If p_2 has the opposite orientation of p_1 then the vehicle immediately turns around and follows p_2.

Straight Line to Circular Path Transitions

This section describes transitions between *line* type path elements where one of the two elements is a circular path. The intersection of the two path elements must be considered. To determine how the path elements intersect comparison of the circular path element's radius r and the minimum distance d from the line path element to the center of the circular path element must be made. The value of r is simply $r = 1/\kappa$, where κ is the curvature of the circular path. The value of d is determined by Equation A.21 [Kanayama 91]. For a directed line $L = (a, b, \theta)$ and a point $p = (x, y)$, where p is the center of the circular path element this distance is given by;

$$dist(L, p) = (y + b)\cos\theta - (x - a)\sin\theta = d \qquad \text{A.21}$$

When the line and circle intersect, there are several possibilities. When the circle's radius r is equal to the minimum distance d from the line to the center of the circle, the single point of tangency is called the osculating point. This is the simplest case since the osculating point is defined as the intersection point and is also the leaving point. This rule results in a small amount of vehicle path oscillation during the transition.

When the circular path element's radius r is less than the minimum distance *dist* from the line to the center of the circle, there are two intersection points, the upstream point and the downstream point. The upstream intersection point [Alexander 93] is returned as the intersection point

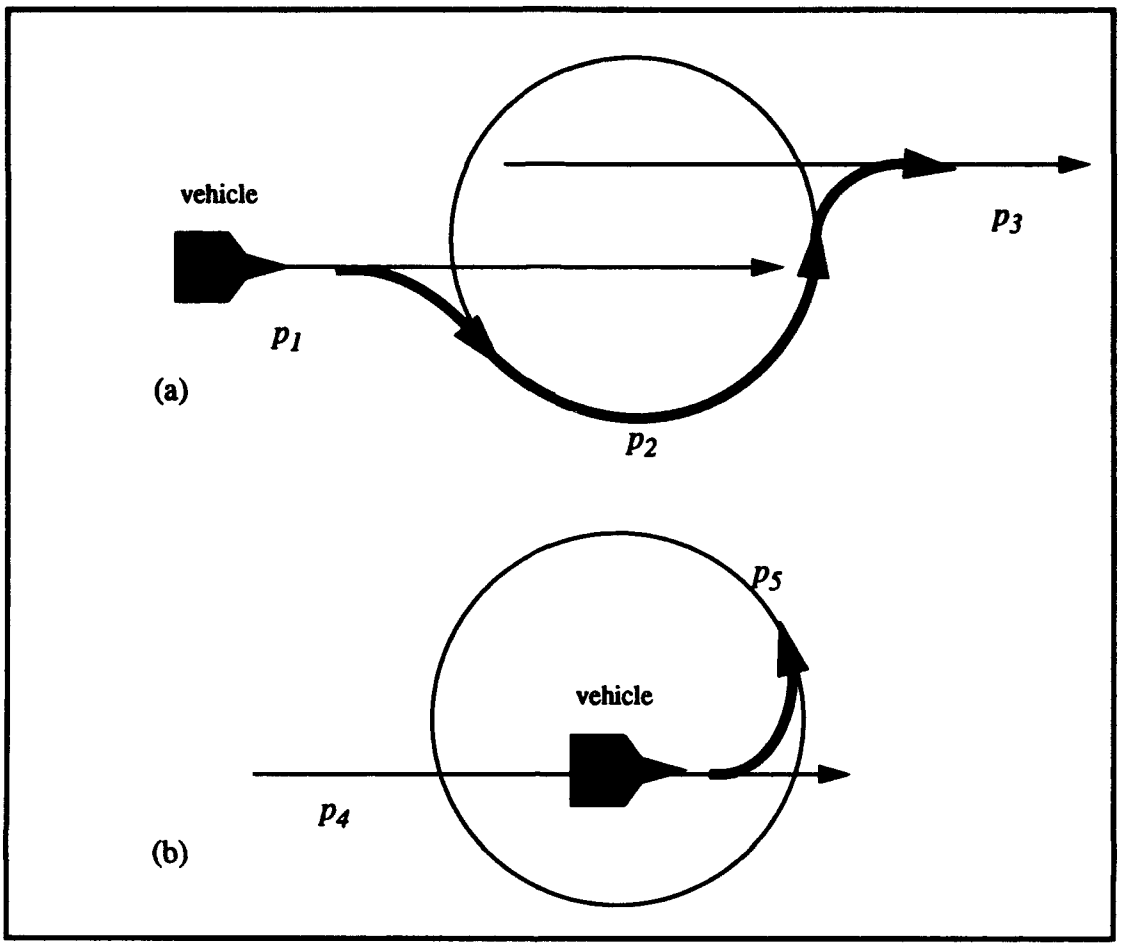


Figure A.13 Transitions Between Intersecting Straight Line Paths and Circular Paths

when the first path p_1 is a line and the vehicle is outside of the circle. See Figure A.13 (a). The leaving point is calculated by the path projection method and lies on the current path element.

When the vehicle is on path element p_4 inside of the circular path element p_5 and the command is read, in order to calculate a straight line to circular path transition, the downstream point is returned as the intersection point. The leaving point is calculated by the path projection method [Alexander 93]. This is shown in Figure A.13 (b). In this case, the leaving point is inside of the circular path element p_5 on path element p_4.

Straight Line to Circular Path Transitions of Non-Intersecting Paths

If d is greater than r, the line and circle path elements are non-intersecting. The mode of the circle is an important consideration when the line and circle are non-intersecting. The transition is only allowed when the circle and the line have the same directionality. Notice in Figure A.14 (a) the circle is counterclockwise (mode +) and the line's direction is left to right. This allows the vehicle to move such that it is not forced to rapidly change direction during the transition from path element p_1 to p_2. A clockwise (mode -) p_2 is not allowed since the vehicle would be forced to make a sharp turn to the left at the leaving point.

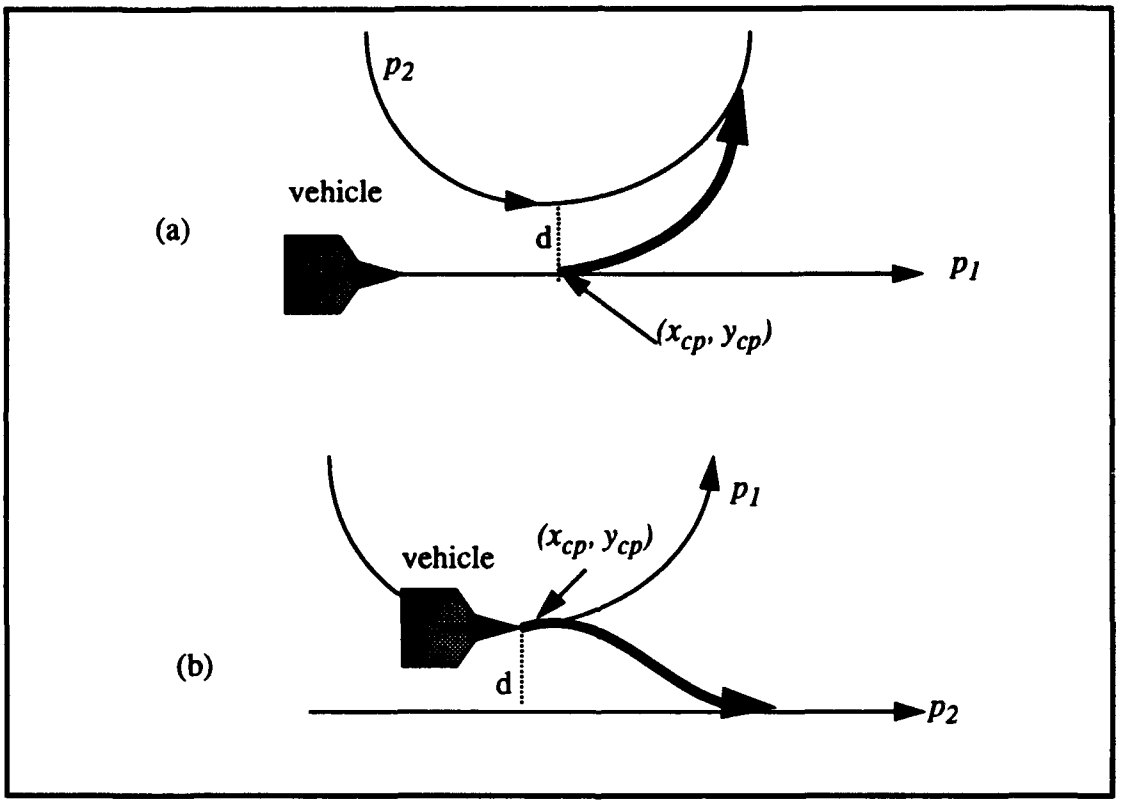


Figure A.14 Transitions Between Straight Line Paths and Circular Paths

Transitions between a straight path element and a non-intersecting circular path element or vice versa require the determination of the closest point (x_{cp}, y_{cp}) between the two path elements. This point is used as the leaving point for non-intersecting elements. In Figure A.14 (a), the vehicle initially tracks straight line path element p_1. The command *line*(&*p2*) specifies a non-in-

tersecting circular path. Path element p_1 is specified by $p_1 = (x_1, y_1, \theta_1, 0)$. Path element $p_2 = (x_2, y_2, \theta_2, \kappa_2)$ is a circle with the center at the point (x_{center}, y_{center}) in accordance with Equations A.22 and A.23, where r is the radius of the circular path element.

$$x_{center} = x_2 - r\sin(\theta_2) \qquad \text{A.22}$$

$$y_{center} = y_2 + r\cos(\theta_2) \qquad \text{A.23}$$

To calculate the closest point, the value of d must be determined. Equations A.21 and A.25 give the closest point (x_{cp}, y_{cp}) on the straight line path element which is the leaving point for the non-intersecting line to circle transition case.

$$x_{cp} = x_{center} + d\cos(\theta_1 - \frac{\pi}{2}) \qquad \text{A.24}$$

$$y_{cp} = y_{center} + d\sin(\theta_1 - \frac{\pi}{2}) \qquad \text{A.25}$$

For a circle to line transition, the closest point on the circle (x_{cp}, y_{cp}) given by Equations A.26 and A.27 is used as the leaving point. When $d > 0$

$$x_{cp} = x_{center} + |r|\cos(\theta_1 - \frac{\pi}{2}) \qquad \text{A.26}$$

$$y_{cp} = y_{center} - |r|\cos(\theta_1 - \frac{\pi}{2}) \qquad \text{A.27}$$

and when $d < 0$

$$x_{cp} = x_{center} - |r| \cos(\theta_1 - \frac{\pi}{2}) \quad \text{A.28}$$

$$y_{cp} = y_{center} + |r| \cos(\theta_1 - \frac{\pi}{2}) \quad \text{A.29}$$

In Figure A.14(b), the vehicle is commanded to move from a circular path element p_1 to a straight line path element p_2. The closest point is used as the leaving point in the same way.

Circle to Circle Path Transitions

Circle to circle path element transitions are designed to provide maximum flexibility in vehicle motion commands. A continuum of proximity exists between two circles with regard to the distance between their centers. The mode of the two circles plays an important part in determining how the transitions between circles should occur. Basically, their are two classes of circle-to-circle transition, same mode and opposite mode. Figure A.15 illustrates four types of transitions between non-intersecting circles. For circles with the same mode, either ++ or - -, the transitions occurs on the exterior of the circles. These transitions are called same mode transitions. For circles with opposite mode, either +- or -+, the transition causes the vehicle to move between circles. These tangents are called opposite mode transitions.

Circle to Circle Path Transitions (Circles with the Same Mode)

To determine if two circles intersect, the sum of the two circle's radii $r_1 + r_2$ is compared to the distance between the centers of the two circles d. If d is greater than $r_1 + r_2$, the two circles are non-intersecting. This case is illustrated in Figure A.16 (a) for circles with the same mode. An external tangent is used as an intermediate vehicle path for this type of transition.

If $d = r_1 + r_2$, then the two circles intersect at an osculating point or point of tangency. This case is illustrated in Figure A.16 (b) for circles with the same mode. Once again an external tangent is used as an intermediate vehicle path element.

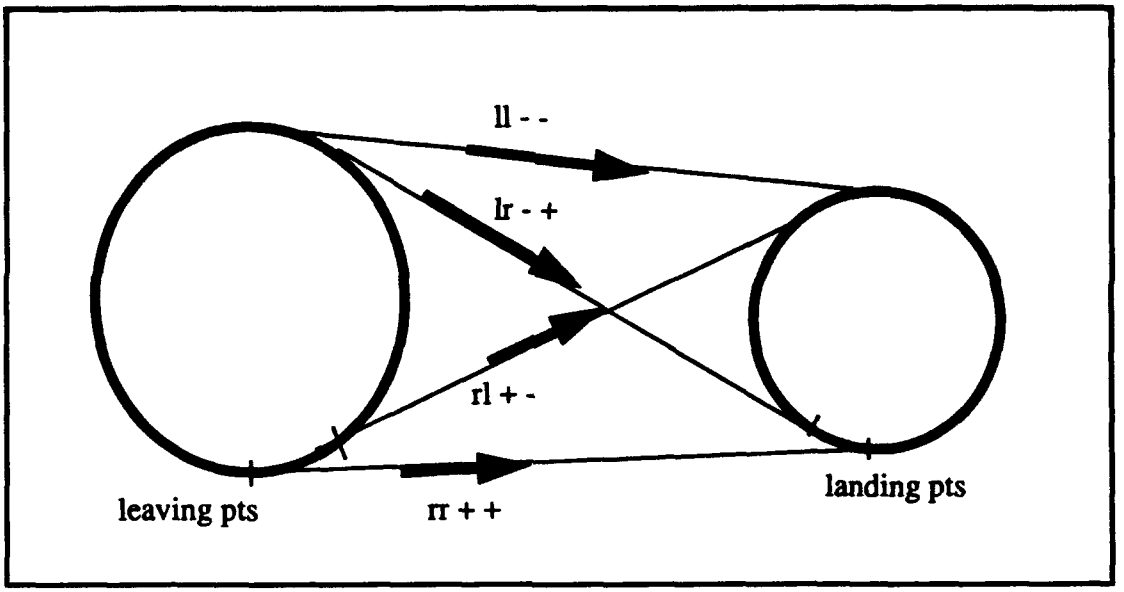


Figure A.15 Tangential Line Segments Between Circular Path Elements

If $|r_1 - r_2| < d < r_1 + r_2$, the circles intersect at two points as shown in Figure A.16 (c). For circles with the same mode, this transition is allowed and an external tangent is used as an intermediate vehicle path element. This transition is not allowed for circles with opposite modes.

If $d = |r_1 - r_2|$, then the two circles intersect at an osculating point with the smaller circle inside of the larger one. This case is shown in Figure A.16 (d). The osculating point is the transition point for circles with the same mode. This type of transition is not allowed for circles with opposite modes.

If $0 < d < |r_1 - r_2|$, then the two circles are non-intersecting with the smaller circle inside of the larger one. Since d is greater than zero, the circles are not concentric. This case is shown in Figure A.16 (e). In this case, for circles with the same mode, the transition point is the *CP* on the current circular path element. This type of transition is not allowed for circles with opposite modes and is handled in section 7.

If $0 = d$, then the two circles are concentric and non-intersecting with the smaller circle inside of the larger one. This case is shown in Figure A.16 (f). In this case, for circles with the same mode, the transition point is the current vehicle image on the current circular path element. This

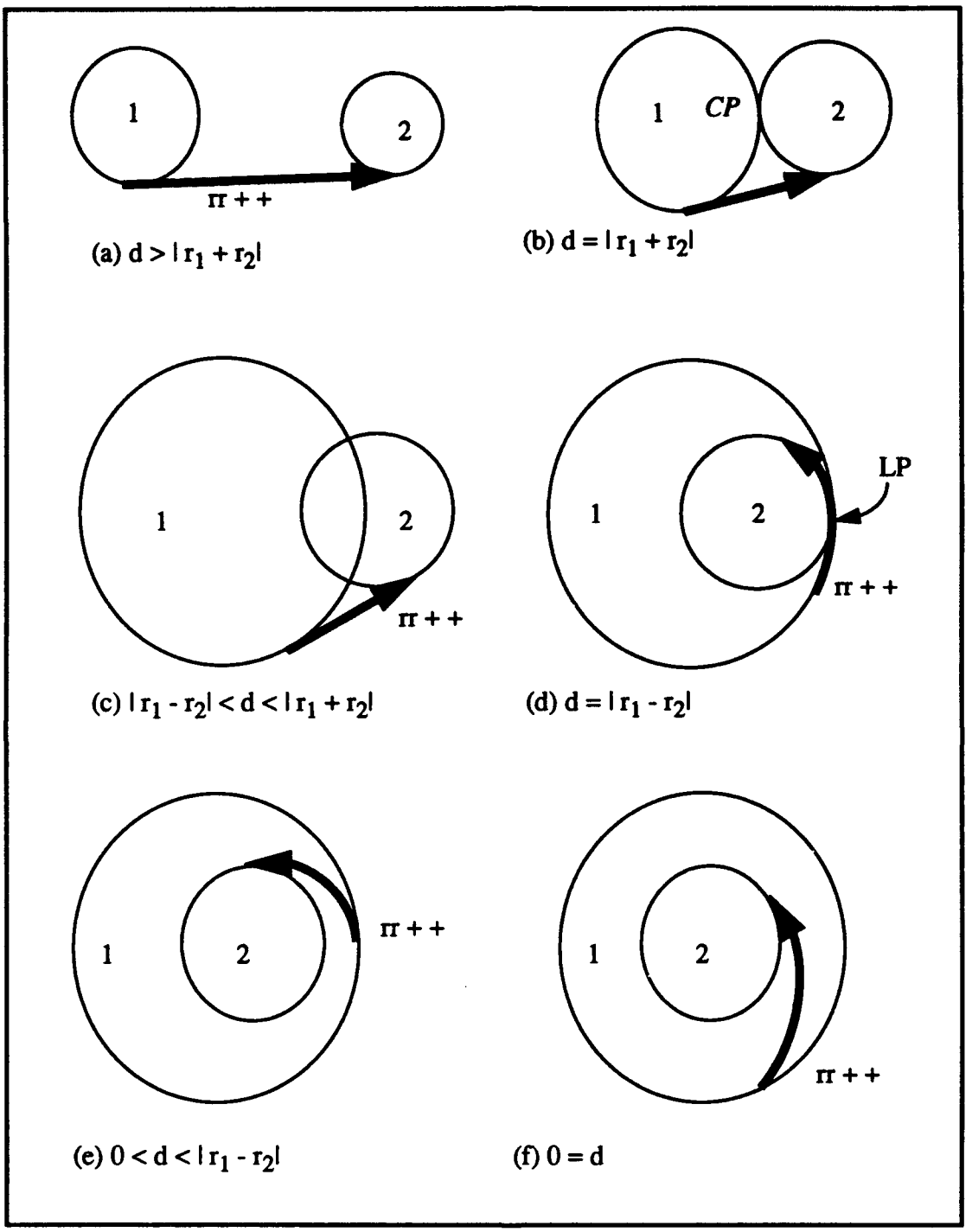


Figure A.16 Circle to Circle Transitions, Same Mode

causes the vehicle to transition immediately from the first path element to the second. Once again, this type of transition is not allowed for circles with opposite modes.

Circle to Circle Path Transitions (Circles with the Opposite Mode)

In Figure A.17 (a) non-intersecting circular path elements with opposite modes are shown. Notice an external tangent is used as an intermediate path element between the two circles.

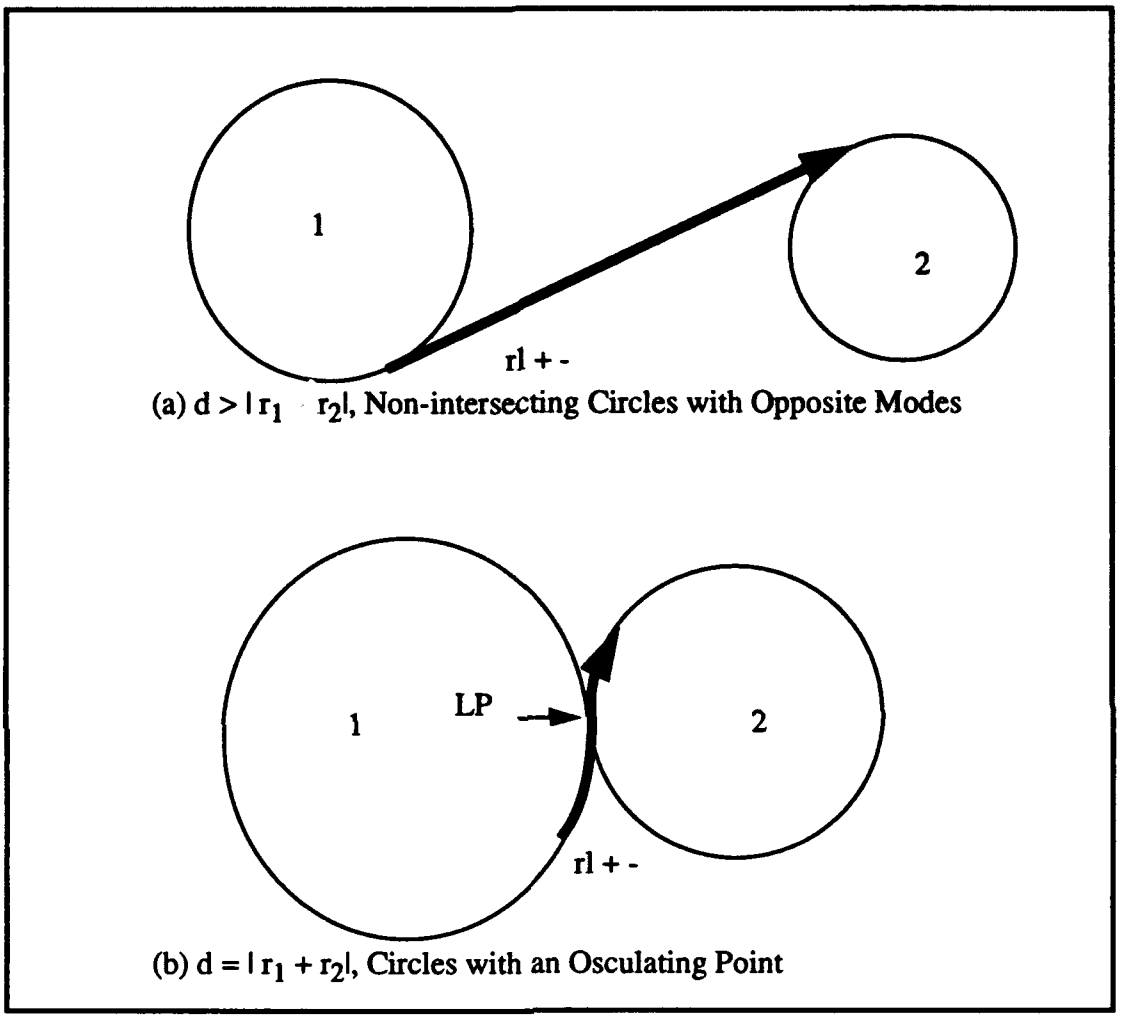


Figure A.17 Transitions Between Circles with Opposite Modes

For intersecting circles with opposite modes, the transition is illustrated in Figure A.17 (b). Notice the vehicle uses the osculating point as the transition point between path elements.

Line to Parabola Path Transitions (Intersecting Paths)

Transitions to and from parabolic path elements are only allowed from straight line path elements due to the complex nature of circle-parabola intersections. Circle to parabola transitions may

have up to four intersection points for some geometries. In Figure A.18 (a), the directed parabolic path element p_1 is specified by a five element vector in accordance with Equation A.30.

$$p_1 = (x_f, y_f, x_d, y_d, \theta_d) \qquad \text{A.30}$$

Where (x_f, y_f) is the focus of the parabola and (x_d, y_d, θ_d) is the parabola's directrix. The straight line path element $p_1 = (x_1, y_1, \theta_1, \kappa_1)$ is specified as before. The vehicle is commanded to follow path p_1 and then p_2, sequentially. The intersection point between p_1 and p_2 is first calculated and then the appropriate leaving point on p_1 is determined [Alexander 93]. In a similiar

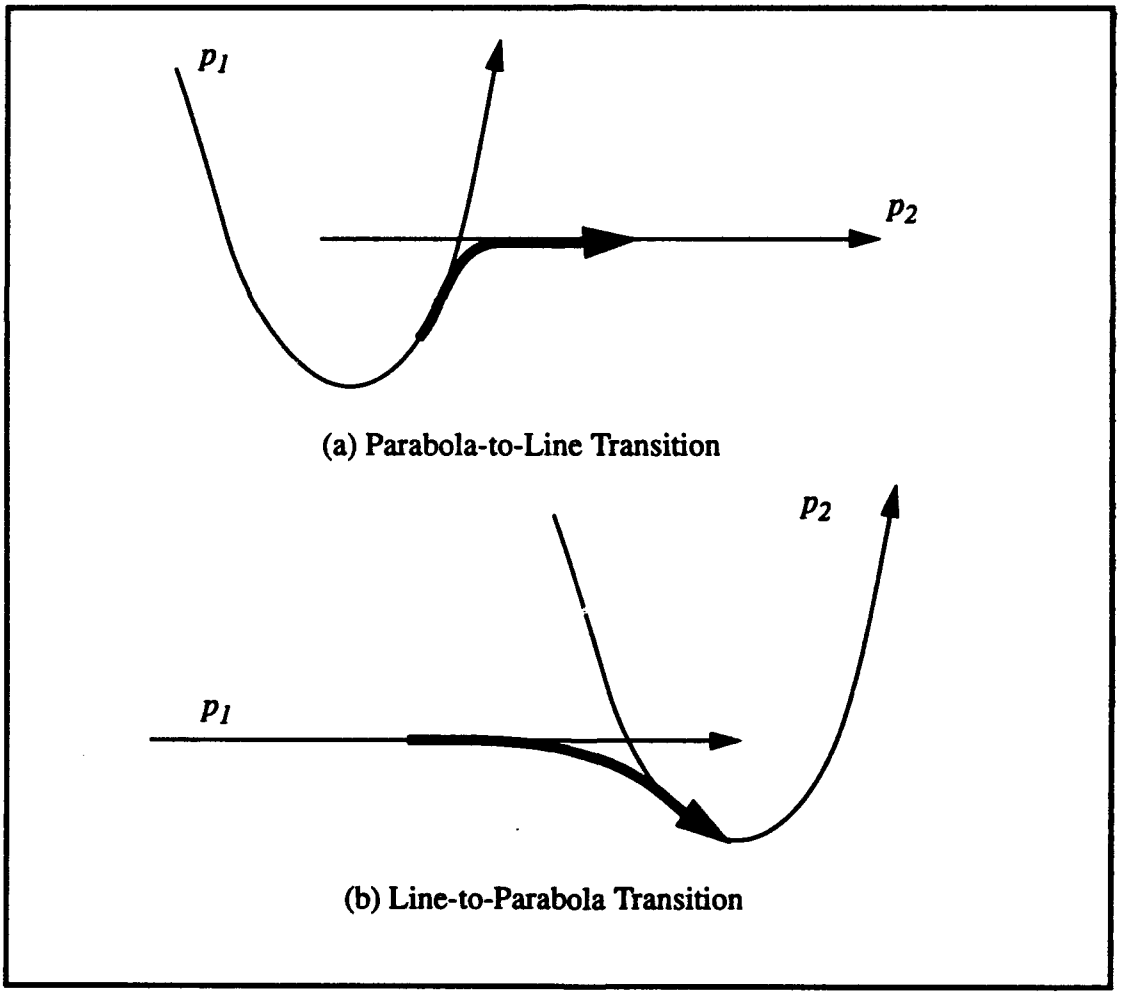


Figure A.18 Transitions between Intersecting Line and Parabolic Path Elements

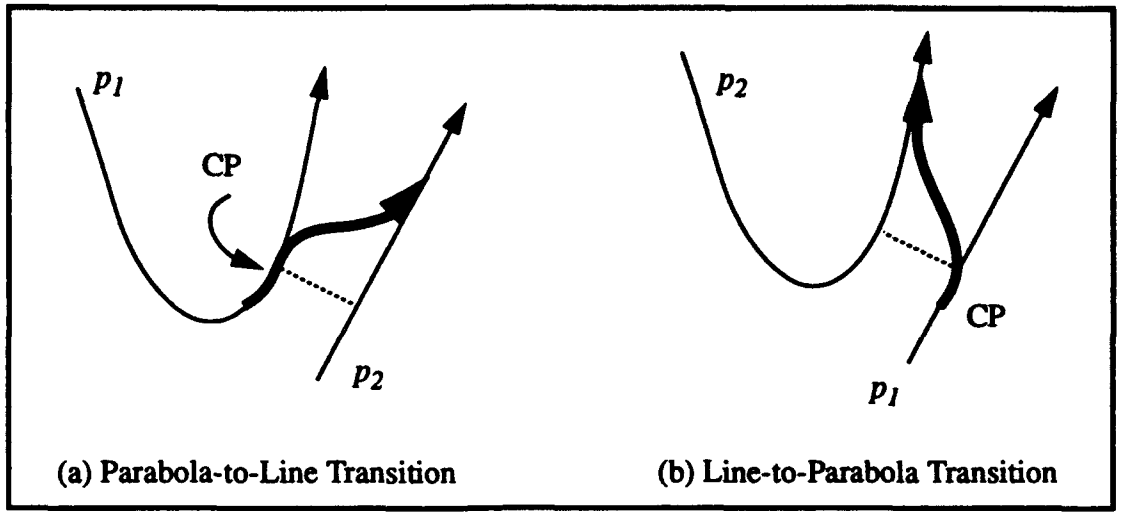


Figure A.19 Transitions between Non-Intersecting Line and Parabolic Path Elements

fashion, the vehicle may be commanded to follow a straight line path p_1 and then transition to a parabolic path p_2 as shown in Figure A.18 (b). The intersection point and the leaving point are calculated in a similiar manner.

Line to Parabola Path Transitions (Non-Intersecting Paths)

For a straight line-to-parabola transition, refer to Figure A.19. If the path elements do not intersect, then the closest point on the straight line path element is used as the leaving point. Similarly, for parabola-to-straight line transitions with no intersection among path elements, the CP on the parabola is used as the leaving point as shown in Figure A.19 (a).

Definitions

Closest Point (CP) - The point on the current path element that has the shortest Euclidean distance to the next sequential path element. This applies to non-intersecting paths only.

Configuration - a four element data structure used to describe a robot position or a path element. The four elements are x, y, θ, and κ.

Immediate Function - Functions that are executed immediately upon the command interpreter reading them. This type of command is not held in a buffer for subsequent execution. Instead, the affected parameters are changed immediately.

Intersection Point - The point or points of intersection between two successive path elements. This is used as a first approximation of the transition point.

Leaving Distance - The distance along the current path element from the leaving point to the intersection point.

Parabola - a five element data structure used to describe a parabolic path element. The five elements are x_f, y_f, x_d, y_d, and q_d. The parabola's focus is represented by the point (x_f, y_f) and the directrix of the parabola is the configuration $(x_d, y_d, q_d, 0.0)$.

Sequential Function - Functions that are executed in a sequential fashion. Each sequential function awaits the logical completion of the previous sequential function.

Transition Point - (Same as leaving point) - The latest hypothetical leaving point that does not result in oscillation in the transition to the next path. At this point the vehicle switches from tracking the current path to the next path.

The Flow of Control

The initialization of variables occurs first. Then control is transferred to user.c. The sequential-type commands are placed into the command buffer and the immediate type commands are executed immediately. The wait_motion and mark_motion commands are used to temporarily halt reading commands into the command buffer.

VEHICLE MOTION COMMANDS

Set Robot Configuration Sequential (set_rob)

Syntax: void **set_rob**(q)
CONFIGURATION q;

Description:

Sets the robot's odometry configuration in a sequential manner. This function is used normally at the start of the MML program to tell the robot where it is initially. Subsequent odometry resets are also made using this function. This function is illustrated in Figure A.20.

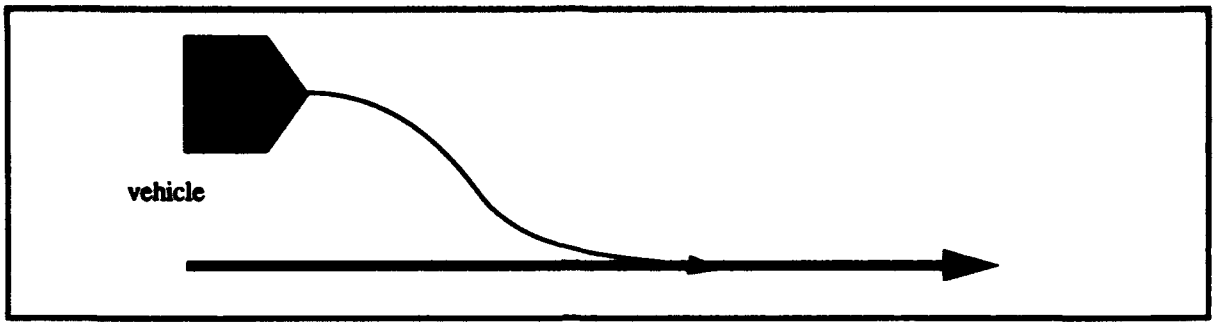


Figure A.20 - The *set_rob* Function

Function Call: **set_rob**(&q)

Location: loco.c

Temporal Type: Sequential Function

Set Robot Configuration Immediate (set_rob0)

Syntax: void **set_rob0**(q)
CONFIGURATION q;

Description:

Resets the robot's odometry configuration such that the robot continues moving in a smooth manner. This resets the x_{od}, y_{od}, and q_{od} components of the robot's configuration only. The kappa of the robot cannot be reset since this would result in an instantaneous change in the robot's curvature. This is not allowed. In Figure A.20, the vehicle is tracking a the desired path. In this case, set_rob0 is used to reset the robot's odometry configuration from the current estimate q_{est} to the actual current configuration q_{act}

Function Call: **set_rob0**(&q)

Location: loco.c

Temporal Type: Immediate Function

Get Robot Configuration Immediate (get_rob0)

Syntax: void **get_rob0**(q)
CONFIGURATION q;

Description:

Retrieves the robot's odometry estimate. Returns a pointer to the location of the robot's current estimate of its configuration.

Function Call: **get_rob0**(&q)

Location: loco.c

Temporal Type: Immediate Function

Move While Tracking a Line (line)

Syntax: void **line**(q)
CONFIGURATION q;

Description:

Command that orders the robot to follow the line specified by the configuration *q*. If the path curvature is zero then *q.kappa* = 0.0. This means that the path represents a straight line passing through the point (*q.x*, *q.y*) with orientation *q.theta*. If the path curvature is nonzero, then the robot follows a circular path. When the value of κ is less than zero then the vehicle's direction of motion on the circle is clockwise, and when κ is greater than zero, then the motion is counterclockwise. These concepts are illustrated in Figure A.21. Speed is automatically reduced to allow the robot to make sharp turns. This is reflected by the dependency between κ and the vehicle speed. In simple terms, the vehicle speed must be reduced to allow it to move safely with larger values of κ.

Function Call: **line**(&q);

Location: loco.c

Temporal Type: Sequential Function

Move While Tracking a Forward Half Line (forward_line)

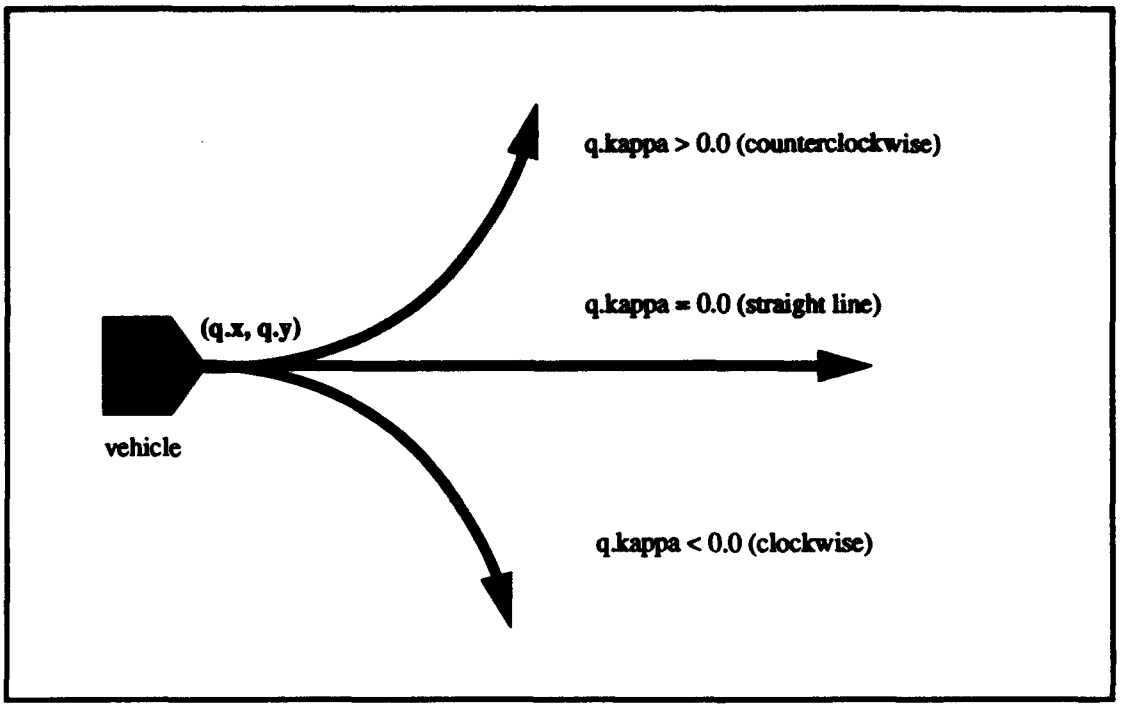


Figure A.21 - The *line* Function

Syntax: void **fline**(q)
CONFIGURATION q;

Description:

Follow the forward_line specified by a configuration *q*. If the vehicle image is on the half line specified by *q*, then the vehicle uses this line as its path. Otherwise, if the vehicle's image does not fall on the half line (i.e. behind the point (*q.x*, *q.y*)) then the vehicle shifts to configuration-to-configuration tracking using a cubic spiral path specification until the line's starting point is reached. See Figure A.22. In case 1 the vehicle's image falls on the half line, in this case the vehicle moves in exactly the same fashion as for the line function. In case 2, the vehicle's image does not fall on the half line. Vehicle motion in case 2 uses point tracking with cubic spirals as the shape of the path. The vehicle tracks to the point (*q.x*, *q.y*) and passes close to this point. The vehicle must pass through the configuration (*q.x*, *q.y*, *q.t*) as it transitions onto the forward half line.

Function Call: **fline**(&q);

Location: loco.c

Temporal Type: Sequential Function

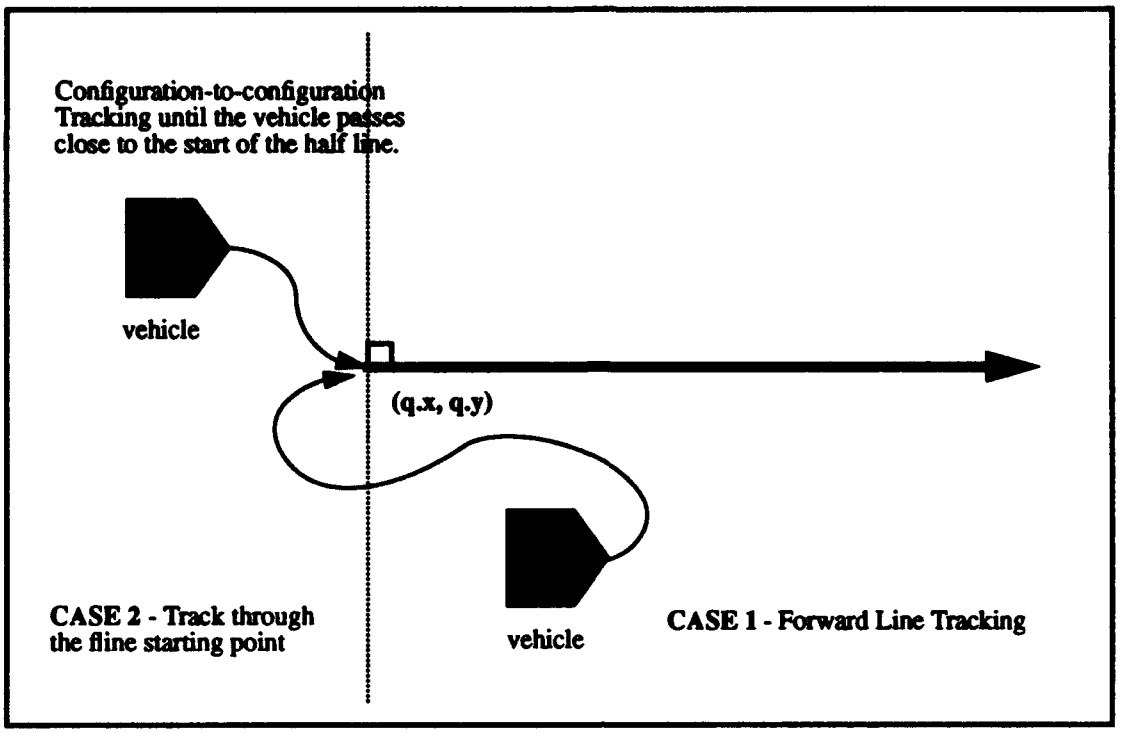


Figure A.22 - The *fline* Function

Move While Tracking a Backward Half Line (backward_line)

Syntax: void **bline**(q)
CONFIGURATION q;

Description:

Follow the backward_line specified by a configuration *q*. See Figure A.23 for an illustration. In case 1, the vehicle image falls on the half-line and the robot tracks as in the line function. In case 2, the vehicle's image does not fall on the half-line, this is an undefined situation that gives an error message or an exception handler. The configuration point (*q.x*, *q.y*) can be used as a vehicle stopping point or as a transition for configuration-to-configuration tracking (see the config(*q*) command).

Function Call: **bline**(&q);

Location: loco.c

Temporal Type: Sequential Function

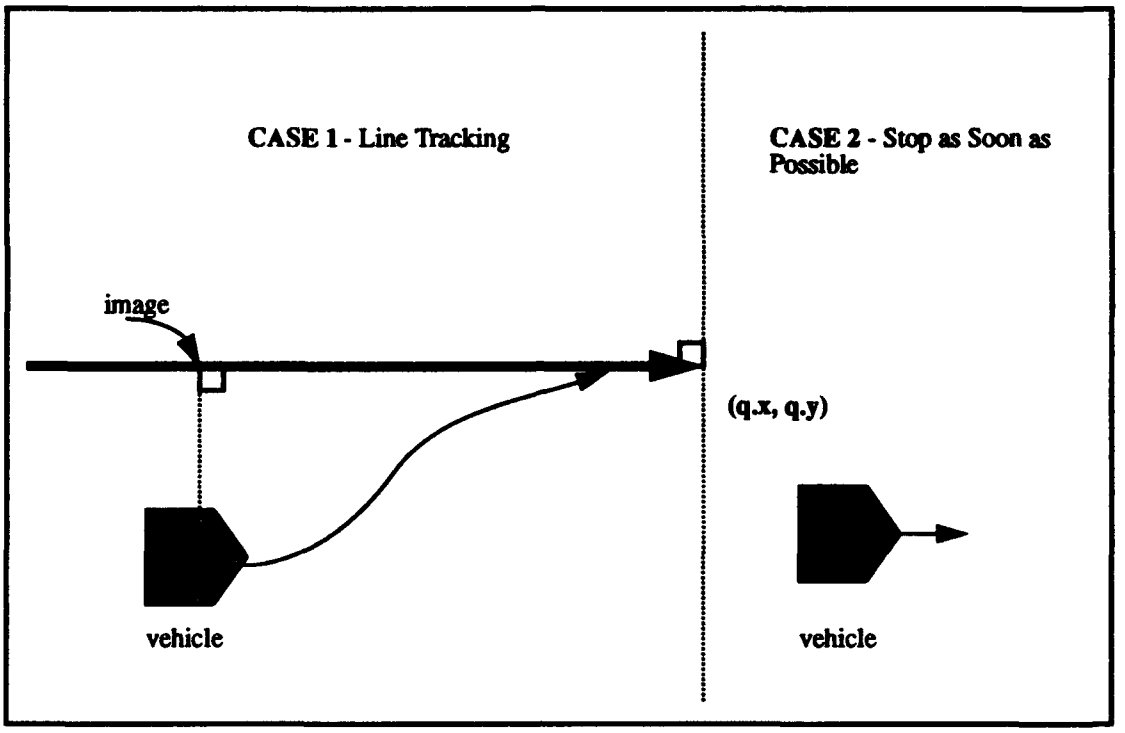


Figure A.23 - The *bline* Function

Define a Robot Configuration Variable (def_configuration)

Syntax: CONFIGURATION **def_configuration**(x, y, t, k, &p)

```
double x;
double y;
double t;
double k;
CONFIGURATION p;
```

Description:

Assigns the four parameters necessary to specify a configuration. The parameters x and *y* define the vehicle's location on the cartesian plane. The parameter t represents the vehicle's orientation and κ represents κ, the curvature of the vehicle's current motion.

Function Call: **def_configuration**(x, y, t, k, &p);

Location: geom.c

Temporal Type: Immediate Function

Define a Parabolic Path Variable (def_parabola)

Syntax: PARA **def_parabola**(xf, yf, xd, yd, td, &p)

```
double xf;
double yf;
double xd;
double yd;
double td;
PARA p;
```

Description:

Assigns the five parameters necessary to specify a parabola. The point (x_f, y_f) is the focus of the parabola and the configuration (x_d, y_d, t_d, 0.0) is the directrix of the desired parabola as shown in Figure A.24. Notice the directrix is always a straight line, therefore the directrix has k = 0 by default.

Function Call: **def_parabola**(xf, yf, xd, yd, td, &p);

Location: geom.c

Temporal Type: Immediate Function

Move While Tracking a Parabola (parabola)

Syntax: void **parabola**(p)

```
PARA p;
```

Description:

Follow the parabola specified by a focus (x_f, y_f) and a directrix specified by a configuration (x_d, y_d, t_d, 0.0), see Figure A.24. The parabola function is used primarily as a means of obstacle avoidance. Figure A.24 illustrates the robot following a straight line path. When an obstacle is encountered

on the robot's intended path, a shift is made to temporary parabolic path tracking. This allows the robot to smoothly maneuver around the detected obstacle and return to the intended straight line path.

Function Call: **parabola**(&p);

Location: loco.c

Temporal Type: Sequential Function

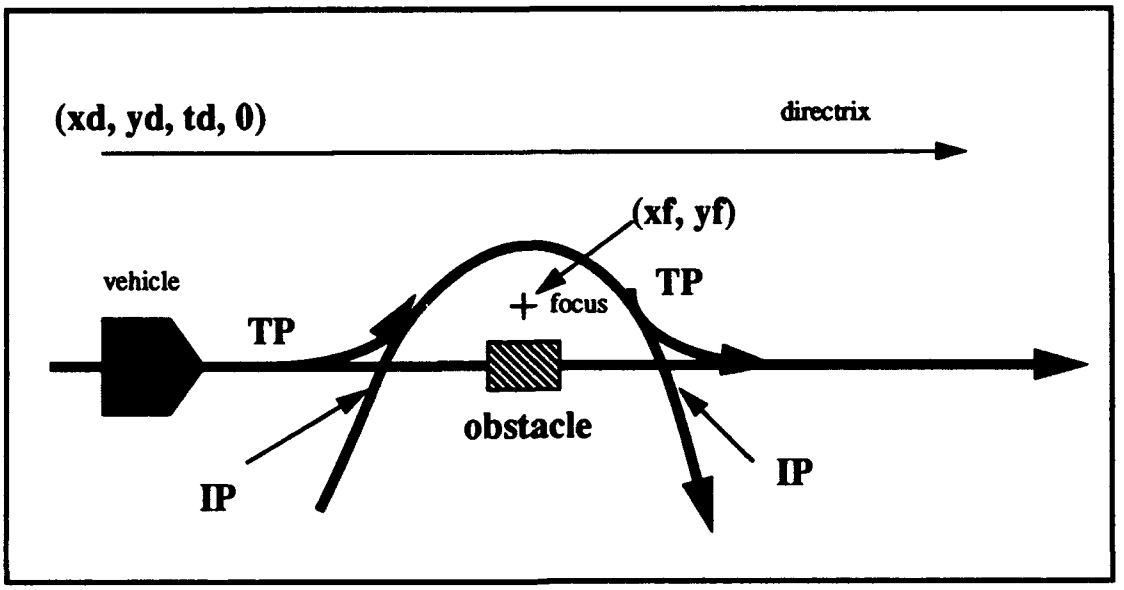


Figure A.24 - The *para* function

Move to a Configuration (config)

Syntax: void **config**(q)

CONFIGURATION q;

Description:

This function specifies a configuration as a destination and uses the configuration from the previous motion commands to reach the destination using one or two cubic spirals. The kappa value of the configuration is ignored since cubic spirals start and end with zero curvature. This is the move command used in previous versions of MML. In Figure A.25 the vehicle is commanded to move through three successive configurations using a series of config commands, namely config(&q1), config(&q2), and config(&q3). The robot automatically plans a smooth cubic spiral path be-

tween successive configurations. Note that not all config-to-config transitions are allowed due to the nature of cubic spirals. An error message or an exception handler is required to recover when prohibited pairs of configurations are specified.

Function Call: **config**(&q);

Location: loco.c

Temporal Type: Sequential Function

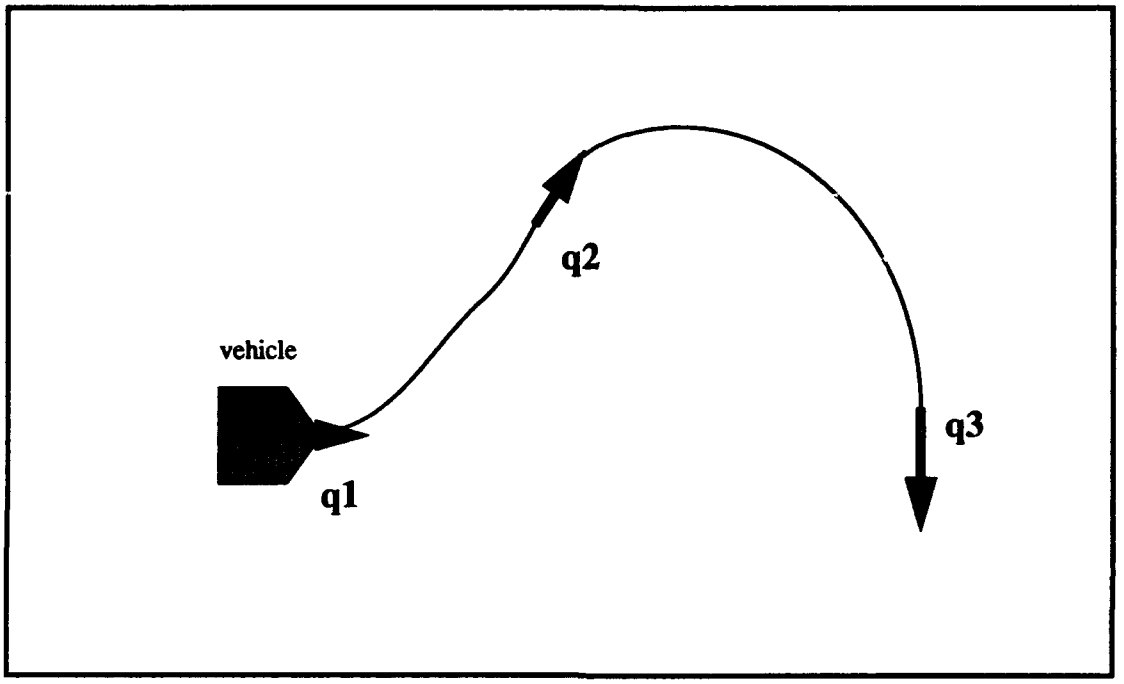


Figure A.25 - The *config* function

Immediate Stop (stop)

Syntax: void **stop0**()

Description:

This function make robot stop dynamically near the point this function was issued. This is not a sequential function, but an immediate one. The sequential functions that have been issued are cancelled. The vehicle accelerates at negative of the set acceleration until fully stopped. All current in-

structions are flushed from the instruction buffer. This function is related to the halt0() and resume0() functions.

Function Call: **stop0**();

Location: loco.c

Temporal Type: Immediate Function

Halt Robot (halt0)

Syntax: void **halt0**()

Description:

This function make robot stop dynamically near the point this function was issued. This is not a sequential function, but an immediate one. The vehicle accelerates at negative of the set acceleration until fully stopped. It does not modify the instruction buffer. This function is related to the stop0() function and is cancelled by a subsequent resume0() function. The robot motion can continue when the resume0() function is issued.

Function Call: **halt0**();

Location: loco.c

Temporal Type: Immediate Function

Location: loco.c

Temporal Type: Immediate Function

Resume Robot Motion (resume0)

Syntax: void **resume0**()

Description:

This function allows the robot to move once again after a halt0() function has been issued. This function does not modify the instruction buffer in any way. The robot motion cannot be resumed until the resume0() function is issued. This function is related to the halt0() function.

should be stored so they do not have to be recomputed

Function Call: **size_const0**(s);

Location: loco.c

Temporal Type: Immediate Function

Set Size Constant Sequential (size_const)

Syntax: void **size_const**(s)
double s;

Description:

This function sequentially updates the size constant s_0 for the tracking algorithm. The new value of s_0 is stored in the instruction buffer and s_0 is changed when the sequential command is executed.

Function Call: **size_const**(s);

Location: loco.c

Temporal Type: Sequential Function

Set Vehicle Speed Immediate (speed0)

Syntax: void **speed0**(s)
double s;

Description:

This function immediately resets the vehicle speed. The nominal vehicle speed is 30 cm/sec. The vehicle moves at the nominal speed until a new speed is set or reset. The vehicle smoothly accelerates to the new speed using the value of the current vehicle acceleration value as the rate.

Function Call: **speed0**(s);

Location: loco.c

Temporal Type: Immediate Function

Set Vehicle Speed Sequential (speed)

Syntax: void **speed**(s)
double s;

Description:

This function sequentially resets the vehicle speed. The nominal vehicle speed is 30 cm/sec. The vehicle moves at the nominal speed until a new speed is set or reset. The vehicle smoothly accelerates to the new speed using the current vehicle acceleration value as the rate.

Function Call: **speed**(s);

Location: loco.c

Temporal Type: Sequential Function

Set Vehicle Acceleration Sequential (acc)

Syntax: void **acc**(a)
double a;

Description:

This function sequentially resets the vehicle acceleration. The nominal vehicle acceleration is 20 cm/sec^2. The vehicle accelerates at the nominal acceleration until a new acceleration is set or reset.

Function Call: **acc**(a);

Location: loco.c

Temporal Type: Sequential Function

Set Vehicle Acceleration Sequential (acc0)

Syntax: void **acc0**(a)
double a;

Description:

This function immediately resets the vehicle acceleration. The nominal vehicle acceleration is 0.5 cm/sec^2. The vehicle accelerates at the nominal acceleration until a new acceleration is set or reset.

Function Call: **acc0**(a);

Location: loco.c

Temporal Type: Immediate Function

Set Vehicle Rotational Speed Immediate (r_speed0)

Syntax: void **r_speed0**(s)
double s;

Description:

This function immediately resets the vehicle's rotational speed. The nominal vehicle rotational speed is 0.5 rad/sec. The vehicle moves at the nominal rotational speed until a new speed is set or reset. The vehicle smoothly accelerates to the new speed using the value of the current vehicle rotational acceleration value as the rate.

Function Call: **r_speed0**(s);

Location: loco.c

Temporal Type: Immediate Function

Set Vehicle Speed Sequential (r_speed)

Syntax: void **r_speed**(s)
double s;

Description:

This function sequentially resets the vehicle's rotational speed. The nominal vehicle speed is 0.5 rad/sec. The vehicle moves at the nominal rotational speed until a new rotational speed is set or reset. The vehicle

smoothly accelerates to the new speed using the current vehicle rotational acceleration value as the rate.

Function Call: **r_speed**(*s*);

Location: loco.c

Temporal Type: Sequential Function

Set Vehicle Rotational Acceleration Sequential (r_acc)

Syntax: void **r_acc**(a)
double a;

Description:

This function sequentially resets the vehicle's rotational acceleration. The nominal vehicle acceleration is 0.5 rad/sec^2. The vehicle accelerates at the nominal acceleration until a new acceleration is set or reset.

Function Call: **r_acc**(a);

Location: loco.c

Temporal Type: Sequential Function

Set Vehicle Rotational Acceleration Immediate (r_acc0)

Syntax: void **r_acc0**(a)
double a;

Description:

This function immediately resets the vehicle's rotational acceleration. The nominal vehicle acceleration is 20 *rad/*sec^2. The vehicle accelerates at the nominal rotational acceleration until a new acceleration is set or reset.

Function Call: **r_acc0**(a);

Location: loco.c

Temporal Type: Immediate Function

Get Total Distance Traveled (path_length)

Syntax: double **path_length**()

Description:

Returns the robot distance traveled since the start of the current program. This distance is stored as the parameter *ss* and it is updated every 10 msec by the odometry function. (This is the control function in the file control.c) This function is a critical part of the current robot odometry correction.

Function Call: distance = **path_length**();

Location: loco.c

Temporal Type: Immediate Function

Wait for a Point (wait_point)

Syntax: void **wait_point**(p)
POINT p;

Description:

Busy waits in task level 0 until the vehicle's image passes a certain point. This function delays the stepwise reading of the "user.c" file until the distance from the vehicle's image to a specified point reaches a local minimum.

Function Call: **wait_point**(p);

Location: loco.c

Temporal Type: Sequential Function

Leave the Current Path Element (skip)

Syntax: void **skip**()

Description:

Causes the robot to immediately leave the current path element. The robot will immediately start tracking the next sequential path element if one exists, otherwise the robot stops moving near its current position. Cannot be used with a rotate-to-cubic spiral transition or any transition-at-endpoint (TRE) command sequence.

Function Call: **skip**();

Location: loco.c

Temporal Type: Immediate Function

Get the Current Path Element (get_line)

Syntax: PATH_ELEMENT **get_line**()

Description:

Returns the path element that the robot is currently tracking. The current path element is returned in the form of a PATH_ELEMENT record. This record consists of four fields; *pc* (type CONFIGURATION), *pp* (type POINT), *tp* (type POINT), and *type* (type int). In the case of an fline function, the path element returned is a cubic spiral or an sline depending on the state of the compound function.

Function Call: element = **get_line**();

Location: loco.c

Temporal Type: Immediate Function

MML IMPLEMENTATION DETAILS

MML System Software Architecture

The path tracking MML system architecture is shown in Figure A.26. This is a partial representation of the entire system. This diagram focuses on the vehicle locomotion. When the program starts to run, initialization occurs first. All global variables are given an initial value and the con-

stants are defined. After the initialization, control is transferred to the user.c code. This is basically the user's commands for the robot. Each command calls a specific MML function. Each MML function is either sequential or immediate.

Table A.2 : MML SYSTEM TASK PRIORITY

Interrupt Level	Interrupt Source	Function	Interrupt Type	Vector	Duration (microsec)
7	stop button	reset	asynchronous	-	-
6	-	not used	-	-	-
5	-	not used	-	-	-
4	Serial Board 1	locomotion	synchronous	64	2500
3	Serial Board 0	laptop	asynchronous	65	variable
2	Sonar Board	sonar	synchronous	66	240
1	Serial Board 0	debugger	-	67	-
0	-	user's instruct	none	-	-

In the level 0 task process, sequential MML functions load the necessary path element information into the command buffer. For a path element function such as *line(&p)*, the path element configuration *p* is loaded into the command buffer and the path intersection point and leaving point are calculated when two or more paths are pending. Since the intersection point and leaving point functions currently run in the foreground, there is some delay in reading sequential user commands. In a later version, the intersection point and leaving point calculations will be tasks run at an interrupt level above the foreground.

Immediate MML functions change one or more global variables, but do not load information into the command buffer. Immediate functions change robot parameters immediately. One example is the *speed0(sp)* function. This function sets the vehicle parameter *vel_c* equal to *sp*, which immediately changes the current vehicle speed. Upon receipt of this command, the vehicle smoothly accelerates to the new commanded speed.

Some explanation of the multitasking processes is required. The Motorola 68020 CPU has eight interrupt levels [Motorola 85]. Some of these interrupts are used to run vehicle tasks at various priority levels in the single CPU, multitasking system. Table A.2 illustrates these vehicle tasks.

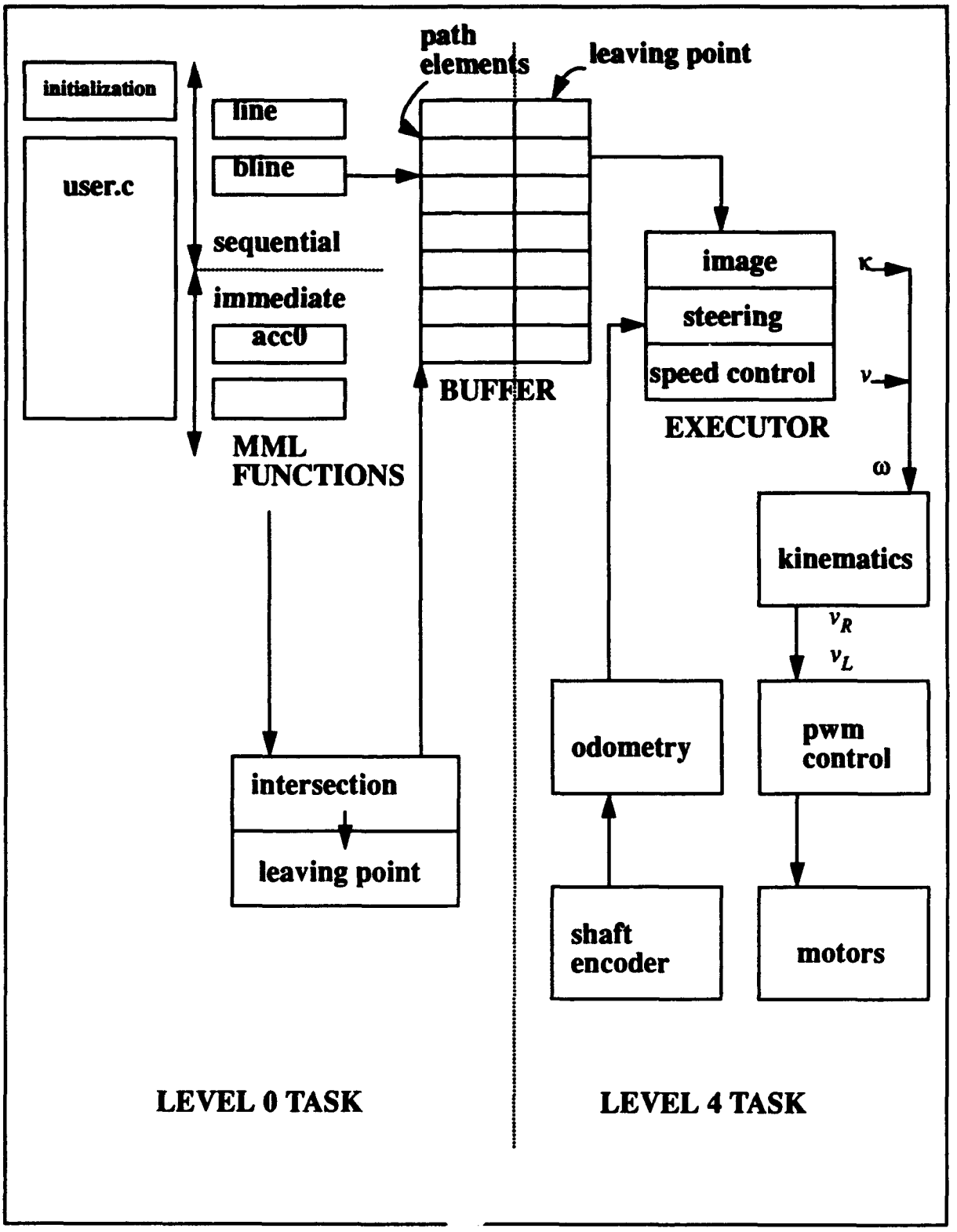


Figure A.26 - MML System Architecture

The higher the interrupt level, the higher the priority of the associated task. At the highest level is the robot's reset button, this tasks overrides all other tasks, stops the robot and resets the CPU. Levels five and six are currently not used. Interrupt level four is the highest priority task that runs during robot operations. This important task is responsible for steering the vehicle. Every 10 msec, the locomotion task interrupts all other lower priority running tasks and runs for about 2500 micro-seconds. This task first reads the shaft encoders and computes the vehicle's odometry configuration estimate. This is a dead reckoning technique since only internal devices are read. Next the most recent odometry configuration is used to calculate the proper k and velocity for the vehicle. These parameters are used to determine the desired vehicle rotational velocity ω. A kinematic function calculates the left and right wheel velocities v_L and v_R. This information is used to determine the necessary pwm command to be sent to the left and right wheel drive motors.

At interrupt level 3, the vehicle's user interface input/output task runs. This task is responsible for printing information to the vehicle on board monitor and reading input from the user entered on the console's keyboard. Also, file transfer from the robot to the host computer is controlled by this task.

At interrupt level 2, the vehicle sensor functions run. This interrupt is triggered by range information that is placed in the sonar board register. When one or more vehicle sonars are enabled, this task runs about every 30 msec. When none of the robot's twelve sonars are enabled, this tasks does not run at all [Sherfey 91].

Interrupt level 1 is the msbn() task which is currently not used. Eventually, the intersection point and leaving point tasks will be transferred to this level.

Interrupt level 0 is the user's instruction interpretation task. Initialization of all variables and interpretation of the user's commands run at this level. All other higher priority tasks can interrupt the level 0 task. This task fills the command buffer based on the user's sequential commands and modifies system parameters based upon immediate commands. The sonar sensors are enabled and disabled at this level. All robot navigation functions run at this level also.

The Command Buffer Data Structure

1. The position of the transition point in the instruction buffer is such that the first path element is written into the instruction buffer with no transition point. The second path element is written

into the instruction buffer only after the transition point between the first and second path elements is determined. All writes to the instruction buffer are atomic. The following is an example:

2. Pointers are maintained to positions in the buffer as follows:

a. *current_path_element_ptr* - the points to the path element that the robot is currently tracking.

b. *current_inst_ptr* - points to the current instruction in the instruction buffer. This instruction is not always a path element.

c. *next_path_element_ptr* - points to the next path element type instruction. This pointer is used in conjunction with the *current_inst_ptr* to calculate the transition point. It is also used by the transition point test routine to determine if the robot has reached the transition point.

d. *new_inst_point* - Points to the next sequential empty portion of the instruction buffer.

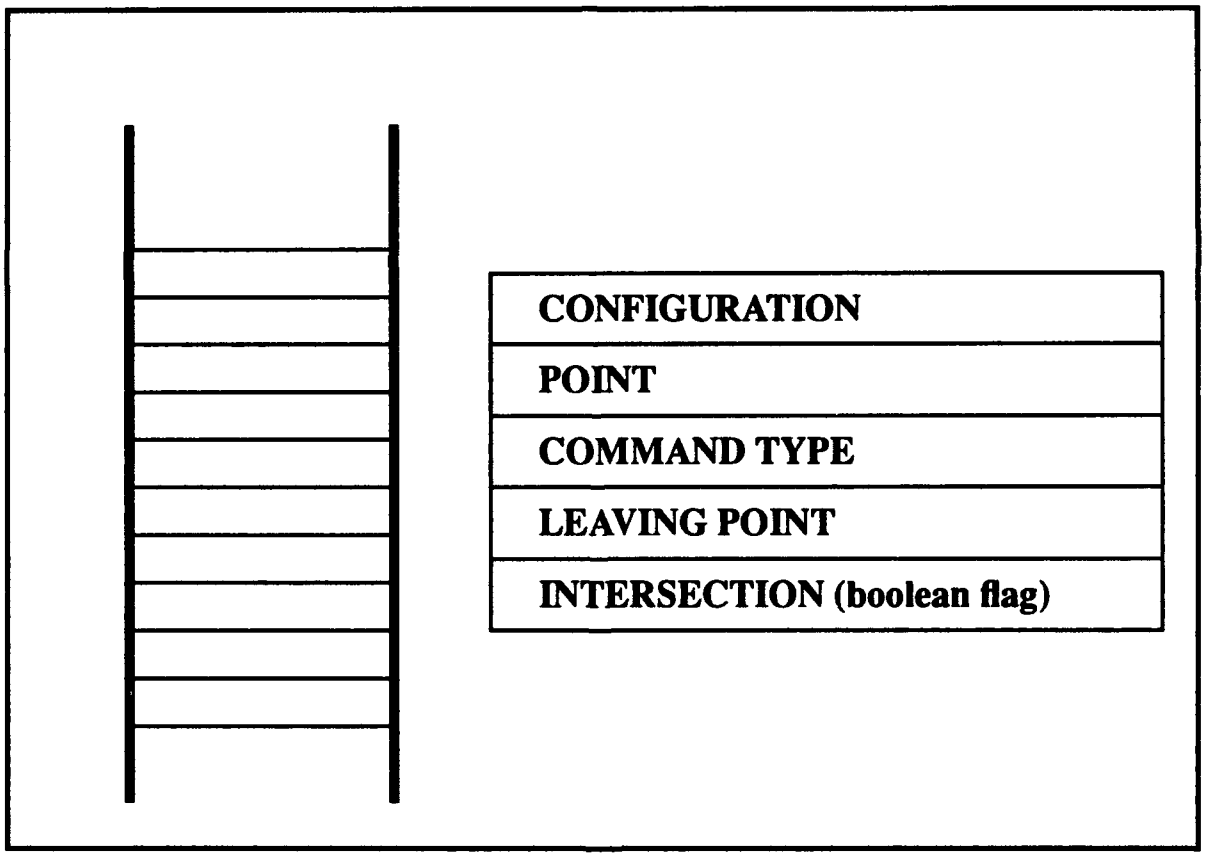


Figure A.27 - Data Structure for the Command Buffer

3. The control function is reconfigured to remove all functions previous done by stepper(). Level 4 tasks run every 10 msec in the following order:

a. Odometry (reads wheel encoders)

b. v, omega update;

1. CASE SSTOP: cur_v = 0; cur_w = 0;

2. CASE SMOVE: current_image = update_image(vehicle, path);

test_TP(current_image, trans_pt);

kappa = update_kappa(vehicle, path);

vel_c = update_vel();

cur_w = update_w(); (= kappa * vel_c)

3. CASE RMOVE: cur_v = 0; cur_w = update_w() /* Trapezoidal control of rotational velocity */

c. update_pwm d. location trace (conditional)

The command interpreter uses an array structure to store path element records. The interpreter reads the *user.c* file one instruction at a time. The procedure *set_instr* reads each path element command, loads the path element data into a path element record and then places the record into the command buffer. The overall system structure is illustrated in Figure A.27. Access to this structure is First-In-First-Out, so commands are stored in the same order as they appear in *user.c*.

Individual path element records consist of the following fields; configuration, point, command type, leaving point and a stop flag. Not all fields are used for all types of path elements, for instance, for a *line(&p)* command, the point field is not used since the line's configuration is stored in the configuration field and the point portion is not necessary. The leaving point part of the path element record is the point where the vehicle should stop following the current path element and start following the next path element. The leaving point is calculated for the first and second elements in the command buffer only. All other leaving point calculations are held pending until the vehicle is actually following the first path element.

The parabola type path element is one command type that uses the point part of this record. This portion of the record is used as the *focus* of the parabolic path element. The path the vehicle is currently following is pointed to by the current_path pointer. This is a pointer to a path element record that represents the vehicle's current path. This path element record is removed from the

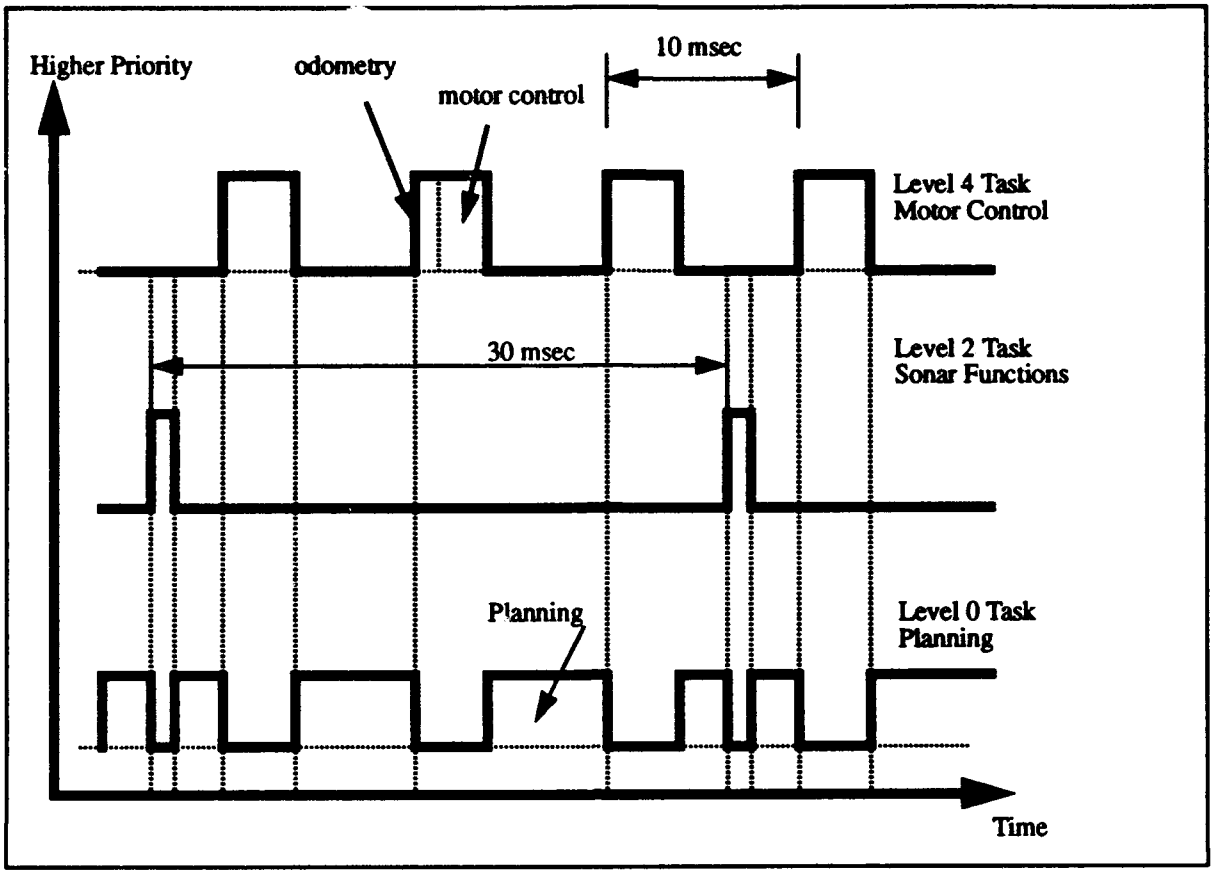


Figure A.28 - System Control Timing

command buffer when the vehicle starts to follow the new path element. The next planned path then moves to the head of the buffer. Leaving point calculation is initiated every time an element is removed from the buffer such that the first and second path elements in the buffer have their leaving points calculated.

b. Sonar Functions - located in the file sonar.c

Procedure: sonar(n)

Description: returns the distance (in centimeters) sensed by the n^{th} ultrasonic sensor. If no echo is received, then a -1 is returned. If the distance is less than 10 cm, then a 0 is returned.

Procedure: enable_sonar(n)

Description: enables the sonar group that contains sonar n, which causes all the sonars in that group to echo-range and write data to the data registers on the sonar control board. Marks the n'th position of the enabled_sonars array to track which sonars are enabled.

Procedure: disable_sonar(n)

Description: removes the sonar n from the enabled_sonars list. If sonar n is the only enabled sonar from its group, then the group is disabled as well and will stop echo ranging. This has benefit of shortening the ping interval for groups that remain enabled.

Procedure: wait_sonar(n)

Description: Busy waits at level 0 until new data is available for sonar n. Then returns the range value for sonar n.

Procedure: global(n)

Description: returns a structure of type **POSIT** containing the global x and y coordinates of the position of the last sonar return.

Procedure: enable_linear_fitting(n)

Description: causes the background system to gather data points from sonar n and form them into line segments as governed by the linear fitting algorithm. Increments service_flag.

Procedure: disable_linear_fitting(n)

Description: causes background system to cease forming line segments for sonar n. Decrements the service_flag. Will also disable the calculation of global coordinates for that sonar if data logging of global data is not enabled.

Procedure: enable_data_logging(n,filetype,filenumber)

Description: causes the background system to log data for sonar (n) to a file (filenumber). The data to be logged is specified by an integer flag (filetype). A value of 0 for filetype will cause raw sonar data to be saved, 1 will save global x and y, and 2 will save line segments. The filenumber may range between 0 and 3 for each of the three types, providing up to 12 data files. Example:

```
enable_data_logging(4,1,0);
```

will cause raw data from sonar #4 to be saved to file 0, while:

```
enable_data_logging(7,2,0);
```

will cause segments for sonar #7 to be saved to file 0.
Function increments the service_flag.

Procedure: disable_data_logging(n,filetype)

Description: causes the background system to cease logging data of a given filetype for a sonar *n*. Decrements the service_flag.

Procedure:serve_sonar(x, y, t, ovfl, data1, data2, data3, data4, group)

Description: this procedure is the "central command" for the control of all sonar related functions. It is linked with the *ih_sonar* routine and loads sonar data to the *sonar_table* from there. It then examines the various control flags in the sonar_table to determine which activities the user wishes to take place, and calls the appropriate functions. This procedure is invoked approximately every thirty milliseconds by an interrupt from the sonar control board.

Procedure: get_segment(n)

Description: returns a pointer to the oldest segment on the linked list of segments for sonar n; i.e. the record at the head of the linked list. It is destructive, thus subsequent calls will return subsequent segments until the list is empty. This is accomplished by first copying the contents of the head record into a temporary record called segstruct and then freeing the allocated memory for the head record. The pointer returned is actually a pointer to this temporary storage. If *get_segment* is called on an empty list a null pointer is returned.

Procedure: get_current_segment(n)

Description: returns a pointer to the segment currently under construction if there is one, otherwise returns null pointer. This is accomplished by calling end_segment, copying the data into segstruct and then returning a pointer to segstruct. The memory allocated by end_segment is then freed.

Procedure: set_parameters(c1,c2,c3)

Description: allows the user to adjust constants which control the linear fitting algorithm. C1 is a multiplier for standard deviation and C2 is an absolute value; both are used to determine if an individual data point is usable for the algorithm. C3 is a value for ellipse thinness; it is used to determine the end of a segment. Default values are set in main.c to 3.0, 5.0, and 0.1 respectively.

Procedure: enable_interrupt_operation()

Description: places sonar control board in interrupt driven mode.

Procedure: disable_interrupt_operation()

Description: stops interrupt generation by the sonar control board. A flag is set in the status register when data is ready, and it is the user's responsibility to poll the sonar system for the flag.

Procedure: calculate_global(n)

Description: this procedure calculates the global x and y coordinates for the range value and robot configuration in the sonar table. The results are stored in the sonar table.

Procedure: linear_fitting(n)

Description: this procedure controls the fitting of range data to straight line segments. First it collects three data points and establishes a line segment with it's interim data values. After the segment is established, the procedure tests each subsequent data point to determine if it falls within acceptable bounds before calling the least squares routine to include the data point in the line segment. After inclusion of the data point the segment is again tested to ensure the entire set of data points are sufficiently linear. If any of the tests fail, the line segment is ended and a new one started. The completed line segment is stored in a data structure called segment, and segments are linked together in a linked list.

Procedure: start_segment(n)

Description: this procedure establishes a new line segment with the three data points contained in segment_data[n].init(x and y). It writes the appropriate data to the interim values in segment_data[n].

Procedure: add_to_line(n, x, y)

Description: this procedure calculates new interim data for the line segment and stores it in segment_data[n]. It also changes the end point values to the point being added.

Procedure: end_segment(n)

Description: this procedure allocates memory for the segment data structure, loads the correct values into it and returns a pointer to the structure.

Procedure: reset_accumulators(n);

Description: resets the accumulative values in segment_data[n] (sgmx, sgmy, sgmx2, sgmy2, sgmxy) to zero.

Procedure: build_list(ptr, n);

Description: this function accepts a pointer to a segment data structure and a sonar number, and appends the segment structure to the tail of a linked list of structures for that sonar.

Procedure: log_data(n, type, filenumber,i)

Description: this procedure causes data to be written to a file. The filenumber designates which "column" (0,1,2, or 3) of a two dimensional array for that type of data is used. The data array and a counter for each column forms the data structure for each type. The value of i is used to index the seg_list array for storing line segments.

Procedure: set_log_interval(n, d)

Description: this procedure allows the user to set how often the sonar system writes data to the raw data or global data files. The interval d is stored at sonar_table[n], and one data point will be recorded for every d data points sensed by the sonar. Default value for interval d is 13, which for a speed of 30 cm/sec and sonar sampling time of 25 msec should record a data point every 10 cm.

Procedure: wait_until(variable, relation, value)

Description: this procedure will delay it's completion (and thus the continuance of the program it's embedded in) until the variable achieves the relation with the value specified. For example, presume the robot is traveling along the x axis. If the user wants the robot to begin recording sonar data when the x position of the robot exceeds 500 cm., he would insert this command after the move command:

```
wait_until(X,GT,500.0);
enable_sonar(sonar number);
```

The variable are predefined as X, Y, A and D0 through D11, and correspond to the robot's current x position, y position, theta, and range from sonars 0 through 11. Relations are predefined as GT, LT and EQ corresponding to greater than, less than and equal to. Value may be any number expressed as a double or the predefined values PI, HPI, PI34, PI4, or DPI.

Procedure: xfer_raw_to_host(filenumber, filename)

Description: this function allocates memory for a buffer and then converts a raw data log file to a string format stored in the buffer. It then calls *host_xfer* to send the string to the host. When that transfer is complete, it frees the memory it allocated for the buffer. Filename must be entered in double quotes
("dumpraw" for example).

Procedure: xfer_global_to_host(filenumber, filename)

Description: this function performs the same function as *xfer_raw_to_host*, except it transfers global data vice raw data.

Procedure: xfer_segment_to_host(filenumber, filename)

Description: this function performs the same function as xfer_raw_to_host, except it transfers segment data vice raw data.

Procedure: finish_segments(n)

Description: this function completes segments at the end of a data run. Necessary because the linear fitting function only terminates a segment based on the data - it has no way of knowing that the user has stopped collecting data.

b. Programming Examples

These user.c file is provided as simple examples of robot programs written in MML
Example: The first example is a simple racetrack. The user.c file follows

```
#include "mml.h"

user()
{
 CONFIGURATION pstart;
 CONFIGURATION first_path;
 CONFIGURATION second_path;
 CONFIGURATION third_path;
 CONFIGURATION fourth_path;
 CONFIGURATION fifth_path;
 double s = 10.0;
 int laps;
 int lap_count = 0;

  buffer_loc=index_loc=malloc(300000);
  bufloc=indxloc=(double *)malloc(60000);
  loc_tron(2,0x3f,30);

  def_configuration(1200.0, 65.0, 0.0, 0.0, &pstart);
  def_configuration(1100.0, 65.1, 0.0, 0.0, &first_path);
  def_configuration(1500.0, 65.0, 0.0, 0.02, &second_path);
  def_configuration(1700.0, 164.9, PI, 0.0, &third_path);
  def_configuration(1200.0, 165.0, PI, 0.02, &fourth_path);

  set_rob(&pstart);
  size_const(s);
  speed(15.0);
  r_printf("\12 Enter desired number of laps. ");
  laps=getint(CONSOLE);
```

```
while (lap_count < laps)
    {
    line(&first_path);
    line(&second_path);
    line(&third_path);
    line(&fourth_path);
    ++lap_count;
    }  /* end while loop */
  line(&first_path);
  wait_until(X, GT, 1400.0);
  loc_troff();
  halt();
  motor_on = NO;
  loc_trdump("loc_dump.8Dec92");
}  /* end user.c */
```

A plot of the robot's motion is shown in Figure A.29. The user first declares five configurations and other variables needed for the program. Next the location trace function is enabled. Then the configurations necessary to allow the robot to move are assigned. The starting configuration is

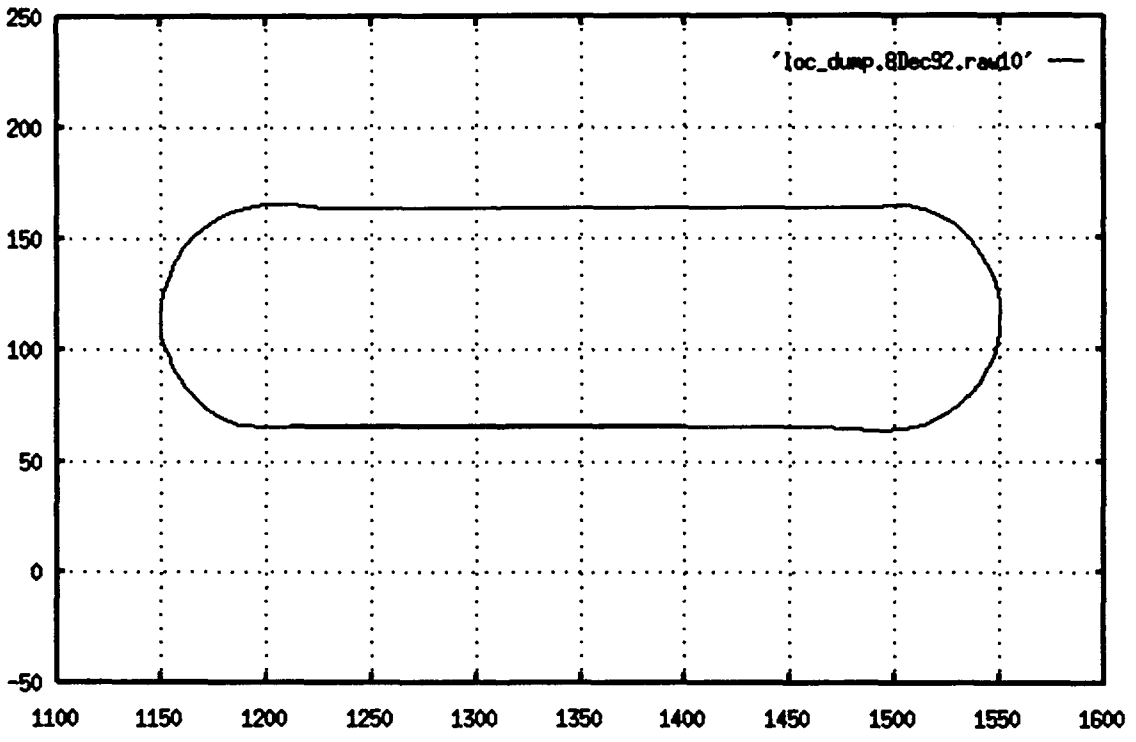


Figure A.29 - Yamabico's Trajectory for Example Program

set to x=65.0, y=1200.0 and theta = 0.0 using the `set_rob(&pstart)` function. The six constant is set to the value of s by the function `size_const(s)` and the speed is set to 30.0 cm/sec by the command `speed(15.0)`. The next two lines of code get the number of times the user desires the robot to drive around the racetrack.

The robot first drives a straight line path (`first_path`), the automatically transitions to the next path. The robot drives around the racetrack controlled by the while loop.

The function `wait_until(X, GT, 1400.0)` tells the robot to wait at level 0 until the robot's odometry value of x exceeds 1400.0. Then the location trace function is turned off by the function `loc_troff()` and the robot is stopped using the `halt()` function. The wheel motors are turned off so the robot can be pushed by the command `motor_on = NO`. Then the robot's location trace date file is transferred back to the host computer using the command `loc_trdump("loc_dump.8Dec92")`, where `"loc_dump.8Dec92"` is the using file.

Definitions

The following definitions are provided to assist the reader in understanding the terms used to explain the path tracking method of vehicle control.

Closest Point (CP) - The point on the current path element that has the shortest Euclidean distance to the next sequential path element or to some point in the Cartesian plane. This applies to non-intersecting paths for determining the appropriate leaving point.

Image - The projection of the robot's current configuration onto the robot's current path element. See Figure A.30. The image is represented as a configuration with the x and y components representing the closest point on the path element from the robot's current odometry estimate. The values of θ and κ are the same as those of the path element at the image point. This information is expressed as a configuration.

Immediate Function - Functions that are executed immediately when the command interpreter reads them. This type of command is not held in a buffer for subsequent execution, instead, the affected parameters are changed immediately.

Instruction Buffer - A first-in-first-out (FIFO) queue for temporary storage of pending robot sequential commands. MML sequential instructions are interpreted and executed using a sim-

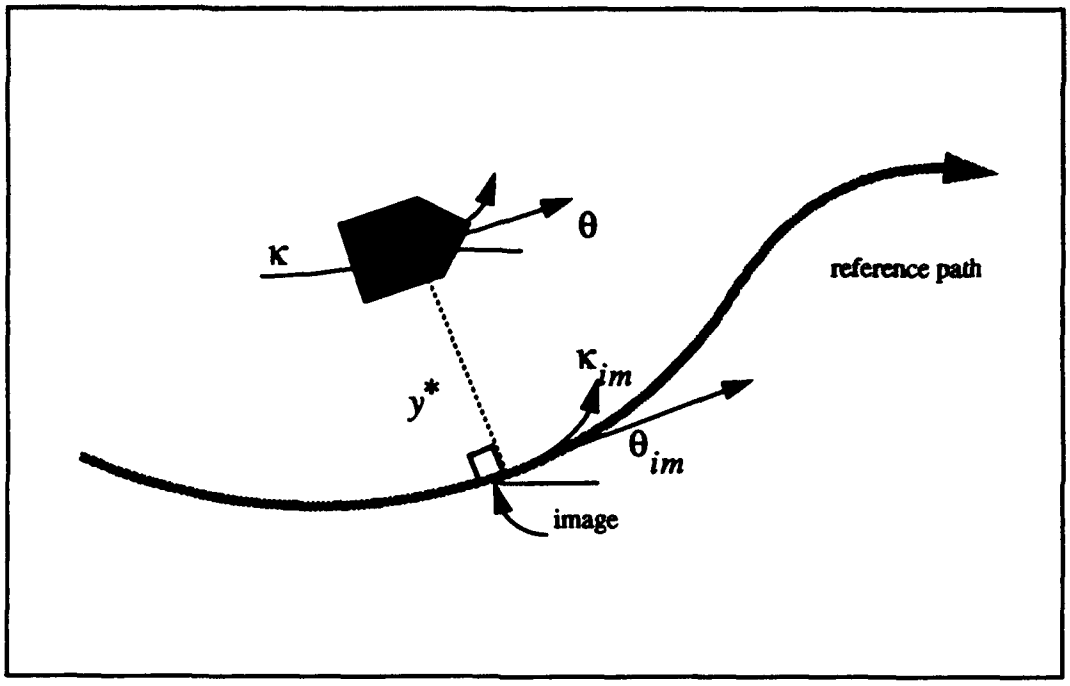


Figure A.30 Path Tracking Control

ple producer and consumer paradigm. A producer task places the user instructions in the instruction buffer. A consumer task removes these instructions and executes them sequentially in the order they are removed from the instruction buffer.

Intersection Point - The point or points of intersection in the Cartesian plane between two successive path elements. This is used as a first approximation of the transition point. When two path elements intersect at more than one point, the intersection points are involved; they are labeled upstream and downstream based on the intended direction of robot motion.

Leaving Distance - The distance along the current path element from the leaving point to the intersection point.

Leaving Point - (same as transition point) - The image on the current path element closest to the intersection point that does not result in oscillation in the transition to the next path. At this point the vehicle switches from tracking the current path to tracking the next path.

Mode - The direction of vehicle motion on a circular path element. Counterclockwise motion is a positive mode and clockwise is a negative mode.

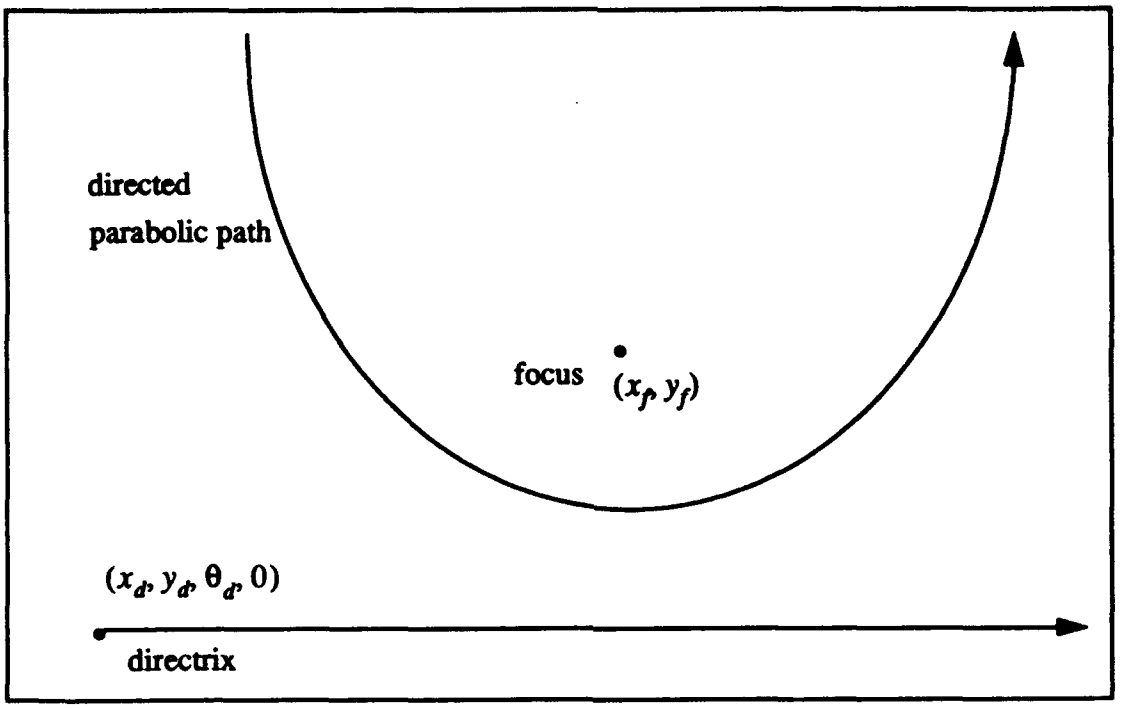


Figure A.31 The Parabola Specification

Parabola - A five element data structure used to describe a parabolic path element. The five elements are x_f, y_f, x_d, y_d, and θ_d. The parabola's focus is represented by the point (x_f, y_f) and the directrix of the parabola is the configuration $(x_d, y_d, \theta_d, 0)$. Figure A.31 is an illustration of a parabolic path element specified by a focus and a directrix.

Sequential Function - Sequential functions are robot control functions that are executed in the order received. Each sequential command must run to completion before the next one can start. Sequential functions are placed into the instruction buffer to await execution. Each sequential function awaits the logical completion of the previous sequential function.

Transition Point - (Same as leaving point) - The image on the current path element closest to the intersection point that does not result in oscillation in the transition to the next path. At this point the vehicle switches from tracking the current path to the next path.

Configuration - the physical location and orientation of a robot represented by p = (x, y, theta, kappa) where (x, y) is a point and the orientation theta is taken clockwise from the x-axis and kappa is the robot's path curvature.

Navigator - an intermediate level on the Intelligence Module which receives the milestones of the future path from the Planner, and performs a more thorough search within the preferable stripe of this level, the results of this search are done at a higher resolution, and they are submitted to the Pilot as a guideline for the lowest level of IM.

Pilot - the lowest level of the Intelligence Module which performs the synthesis of actual motion trajectory. In other words the Pilot translates the output from the Navigator into the trajectory of the mechanical motion in accordance with the accepted set of elementary maneuvers.

Planner - the upper level of the Intelligence Module which performs the search for an optimum path at a lowest resolution. Planner determines the preferable stripe determined by the milestones, which is submitted to the Navigator for the subsequent stage of the planning.

Sonar Fix - A determination of the robot's current location or previous location using input from the sonar system and the world model

Sonar Model - An abstract geometical model of a known world. This model is implemented in software and basically simulates the sonar returns that would be obtained by a robot at a given posture. The robot may utilize actual sonar returns, its dead reckoned posture and output from the sonar model, to fix its position in a given world. Additionally, this allows the robot to discriminate between known and unknown obstacles.

Sonar Model

The sonar model consists of line segments line0 through line11 representing the twelve Yamabico sonars. Each line segment has a length of 4 meters, which is the ideal maximum range of the Yamabico sonar sensors.

The world is modeled as a number of line segments. A function, segment_crossing_test(lineA, lineB) returns true if the line segments lineA and lineB cross at any point. The twelve Yamabico sonars are modeled mathematically as 4 meter long line segments. Each of the twelve sonar line segments is tested for crossing with the world line segments. If a sonar line segments crosses a

world line segment, the intersection r of the two line segments is determined. The distance from r to the robot's current position is returned as the expected sonar range to the known obstacle.

Index

APPENDIX B. LOCOMOTION SOURCE CODE

```
/*****************************************************************************
FILENAME: control.c
PURPOSE: path tracking functions for MML
CONTAINS:
 control()
 pwm_lookup(vel)
AUTHOR: Dave MacPherson
DATE: 1 Feb 93
COMMENTS: Needs more work.
*****************************************************************************/

#include "mml.h"

/*****************************************************************************
FUNCTION: control
PARAMETERS: none
PURPOSE: Reads robot encoders to update odometry
every 10 msec (INTVL) and then sends
commands to the motors that drive the
wheels.
RETURNS: pwm commands to drive the left and right drive
 wheels.
CALLED BY: motor (assembly language code)
CALLS:
COMMENTS: 26 April 93 - Dave MacPherson
*****************************************************************************/
control()
{
        register int lpwm, rpwm, bufpwm;
        register double v_l, v_r, cur_vl, cur_vr;
        register double dist_inc;
        register double dtheta;
        register double cur_v, cur_w;
        double delta_theta, delta_dist1;

        double pwm_lookup();
        void store_loc_trace_data();
        void next();
        double read_left_wheel_encoder();
        double read_right_wheel_encoder();
        void get_velocity();

#ifndef SIM
        /* calculate the required linear and rotational velocities */
        get_velocity(&uv, &uw, vehicle);
```

```
/* get encoder information returns how far the left and right
 wheels have moved forward in one step */


        if (status != RMOVE)
        tread = TREAD;/* Narrower trend width for forward motion */
        else
        tread = TREAD_R;
        /* If robot is in the rotate mode use wider trend width */
        tread2 = 0.5 * tread;
        dist_inc = (read_right_wheel_encoder()
         + read_left_wheel_encoder()) / 2.0;
        ss += dist_inc;
        dtheta = (read_right_wheel_encoder() - read_left_wheel_encoder()) / tread;
        cur_v = dist_inc / INTVL;
        cur_w = dtheta / INTVL;
        cur_vl = cur_v - tread2 * cur_w;
        cur_vr = cur_v + tread2 * cur_w;
        /* get left and right wheel velocity */

/* update current configuration */
        next(&vehicle, dist_inc, dtheta);
        cur_x = vehicle.x;
        cur_y = vehicle.y;
        cur_t = vehicle.t;
        vehicle.k = kappa;

/* trace loc */
        if (ltrace_f != 0)
        store_loc_trace_data(vehicle.t, vehicle.k, uv, uw);

        /* if the vehicle's motors are not on the return */

        if (!motor_on)
        return;

#endif


        if (setting_configuration)/* for set_c function temporal exec */
        {
        setting_configuration = NO;
        vehicle.x = set_P.x;
        vehicle.y = set_P.y;
        vehicle.t = norm(set_P.t);
        }
        /* calculate the required linear and rotational velocities */
/* get_velocity(&uv, &uw); */

        if (emg_stp != 0)
```

```
        uv = uw = 0.0;

        /* compute commanded left and right wheel velocities */
        v_l = uv - tread2 * uw;
        v_r = uv + tread2 * uw;

#ifdef SIM
        /*
         * For the simulator compute vehicle's configuration based on left
         * and right commanded wheel speed this is required in lieu of real
         * odometry
         */

        delta_theta = uw * INTVL;
        delta_dist1 = uv * INTVL;
        vehicle.x += (cos(vehicle.t + delta_theta / 2.0)
         * delta_dist1);
        vehicle.y += (sin(vehicle.t + delta_theta / 2.0)
         * delta_dist1);
        vehicle.t = vehicle.t + delta_theta;
        vehicle.k = kappa;
#endif

/* adjust pwm's based upon the difference between the
 calculated wheel velocity and the odometry wheel
 velocity */
        /* left wheel */
        lpwm = pwm_lookup(v_l) + kpw_b * (v_l - cur_vl);

        /* right wheel */
        rpwm = pwm_lookup(v_r) + kpw_b * (v_r - cur_vr);

#ifndef SIM

/* set up motor control word (mcw) and threshold pwm values */

        if (mv_direction < 0.0)
        {
        bufpwm = lpwm;
        lpwm = -rpwm;
        rpwm = -bufpwm;
        }
        mcw = (mcw & 0xf0f0) | ((lpwm) > 0 ? 1 : 2) | ((rpwm) > 0 ? 0x0100 : 0x0200);
        if (lpwm > 127)
        lpwm = 127;
        else if (lpwm < -127)
        lpwm = -127;
        if (rpwm > 127)
        rpwm = 127;
        else if (rpwm < -127)
        rpwm = -127;
```

```
#endif


        return (lpwm << 16 | rpwm & 0xff);
}       /* end control */


/*****************************************************************************
FUNCTION: get_velocity()
PARAMETERS: uv, uw
PURPOSE: Determines the robot velocity and rotational
velocity based upon vel_c and kappa.
RETURNS: *uv, *uw
CALLED BY: control()
CALLS: update_vel(), update_kappa(), transition_point_test()
COMMENTS: 23 Apr 93 - Dave MacPherson
TASK: Level 4 interrupt
*****************************************************************************/
void get_velocity(uv, uw)
double *uv, *uw;
{
        PATH_ELEMENT path;

        switch (status)
        {
        case SSTOP:
        length_s = 0.0;
        vel_c = 0.0;
        if (wait_cnt == 0)
        {
        (*uv) = vel_c;
        (*uw) = 0.0;
        read_inst();
        } else
        wait_cnt--;
        break;
/*
        case SPWAY:
        (* uv) = vel_c = update_vel();
        if (sonar(4) == -1);
        kappa = 0.0;
        else if (sonar(4) > 50.0)
        kappa = -0.01;
        else if (sonar(4) <= 50.0)
        kappa = 0.01;
        (* uw) = kappa * vel_c;
        break;
*/
        case SLINE:
```

```
(*uv) = vel_c = update_vel();/* commanded velocity */
kappa = update_kappa();
(*uw) = kappa * vel_c;/* commanded omega */

if (get_inst != put_inst)
{
if (transition_point_test(current_image, get_inst->tp))
{
--no_o_paths;
current_robot_path.pc = get_inst->c;
current_robot_path.type = get_inst->class;

read_inst();
}/* end if */
}
if (skip_flag_control)
{
current_robot_path.pc = get_inst->c;
read_inst();
skip_flag_control = 0;
}
break;

case SBLINE:
(*uv) = vel_c = update_vel();/* commanded velocity */
kappa = update_kappa();
(*uw) = kappa * vel_c;/* commanded omega */
/*
 * if (no_o_paths > 1) if
 * (transition_point_test(current_image, path,
 * get_inst->tp)) { --no_o_paths;
 * current_robot_path.pc = get_inst->c; read_inst();
 * }
 */
if (vel_c < 0.5 && EU_DIS(current_image.x, current_image.y,
 current_robot_path.pc.x, current_robot_path.pc.y) < 1.0)
{
status = SSTOP;
read_inst();
}
if (skip_flag_control)
{
current_robot_path.pc = get_inst->c;
read_inst();
skip_flag_control = 0;
}
break;

case SCONFIG:

(*uv) = vel_c = update_vel();/* commanded velocity */
```

```
/*
 * if (first_time) { first_time = FALSE;
 *
 * } else call update_cubic_image to advance image
 */


current_image = update_cubic_image(vehicle, current_robot_path);


/*
 * Now update the global kappa that is used to
 * control the robot's actual motion
 */

kappa = update_cubic_kappa(vehicle, current_image);

(*uw) = kappa * vel_c;/* commanded omega */

/*
 * There are many tests to see if the end of the
 * spiral has been reached, easiest is to compare
 * image_s to precomputed length of spiral, stored in
 * pp.x0
 */

if (image_s > current_robot_path.pp.x0)
{
read_inst();
}/* end if */
break;

case SPARABOLA:
break;
case RMOVE:
(*uv) = 0.0;
(*uw) = rvel_c = get_rotational_vel();
break;
case SERROR:
vel_c = update_vel();
if (vel_c <= VEL1)
{
vel_c = 0.0;
motor_on = ON;
}
break;
default:
(*uv) = 0.0;
(*uw) = 0.0;
break;
}/* end switch */
```

```
}       /* end get_velocity() */

/*****************************************************************************
FUNCTION: read_left_wheel_encoder
PARAMETERS: none
PURPOSE: Determines the distance moved by the left
wheel by reading the optical encoder. Filters the
data using a recursive digital filter.
RETURNS: dist_l ( The distance moved by the right wheel
in the current vehicle control cycle ).
CALLED BY: control()
CALLS: none
COMMENTS: 20 Apr 93 - Dave MacPherson
TASK: Level 4 interrupt
*****************************************************************************/
double read_left_wheel_encoder()
{
        double a = 0.7;/* filter constant */
        double dist_l;

        if (mv_direction > 0.0)
        dist_l = mv_direction * dlenc * ENC2DIST;
        else
        dist_l = mv_direction * drenc * ENC2DIST;

        /* Recursive Digital Filter */
        dist_l = a * dist_l + (1 - a) * last_dist_l;
        last_dist_l = dist_l;
        return dist_l;
}       /* end read_left_wheel_encoder */


/*****************************************************************************
FUNCTION: read_right_wheel_encoder
PARAMETERS: none
PURPOSE: Determines the distance moved by the right
wheel by reading the optical encoder. Filters the
data using a recursive digital filter.
RETURNS: dist_r ( The distance moved by the right wheel
in the current vehicle control cycle ).
CALLED BY: control()
CALLS: none
COMMENTS: 20 Apr 93 - Dave MacPherson
TASK: Level 4 interrupt
*****************************************************************************/
double read_right_wheel_encoder()
{
        double a = 0.7;/* filter constant */
        double dist_r;

        if (mv_direction > 0.0)
```

```
        dist_r = mv_direction * drenc * ENC2DIST;
        else
        dist_r = mv_direction * dlenc * ENC2DIST;

        /* Recursive Digital Filter */

        dist_r = a * dist_r + (1 - a) * last_dist_r;
        last_dist_r = dist_r;
        return dist_r;
}       /* read_right_wheel_encoder */


/*****************************************************************************
FUNCTION: next
PARAMETERS: q, delta_s, delta_theta
PURPOSE: Updates the robot's current configuration based upon
the input values of delta_s and delta_theta.
RETURNS: *q ( a pointer to a configuration ).
CALLED BY: control()
CALLS: none
COMMENTS: 19 Apr 93 - Dave MacPherson
TASK: Level 4 interrupt
*****************************************************************************/
void next(q, delta_s, delta_theta)
CONFIGURATION *q;
double delta_s, delta_theta;
{
        double sinc;
        double dtheta2 = delta_theta / 2.0;

        sinc = delta_s;
        if (delta_theta)
        sinc *= sin(dtheta2) / dtheta2;

        /* Update The vehicle's odometry estimate */
        q->x += sinc * cos(q->t + dtheta2);
        q->y += sinc * sin(q->t + dtheta2);
        q->t = q->t + delta_theta;
}       /* end next */


/*****************************************************************************
FUNCTION: pwm_lookup
PARAMETERS: vel (wheel velocity)
PURPOSE: Determines the estimated pwm ratio given
the desired wheel velocity as an input.
RETURNS: pwm value based upon empirically determined velocity
        vs pwm ratio curve.
CALLED BY: control()
CALLS: none
COMMENTS: 12 Jan 93 - Dave MacPherson
```

```
TASK: Level 4 interrupt
*****************************************************************************/
double pwm_lookup(vel)
double vel;
{
        double v = vel;
        double pwm_value;

        v = fabs(vel);
        if (v >= 0.0 && v < 20.0)
        pwm_value = (0.5 * v + 13.0);
        else if (v >= 20.0 && v < 30.0)
        pwm_value = (1.256 * (v - 20.0) + 23.0);
        else if (v >= 30.0 && v < 40.0)
        pwm_value = (2.413 * (v - 30.0) + 35.56);
        else if (v >= 40.0 && v < 50.0)
        pwm_value = (1.651 * (v - 40.0) + 59.69);
        else if (v >= 50.0 && v < 60.0)
        pwm_value = (2.54 * (v - 50.0) + 76.2);
        else if (v >= 60.0 && v < 65.0)
        pwm_value = (5.08 * (v - 60.0) + 101.6);
        else
        r_printf("Error in pwm lookup function");

        if (vel > 0.0)
        return pwm_value;
        else if (vel < 0.0)
        return -pwm_value;
        else
        return 0.0;
}       /* end pwm_lookup */


/*****************************************************************************
FUNCTION: store_loc_trace_data()
PARAMETERS: arg1, arg2, arg3, arg4
PURPOSE: Records location trace data if enabled.
RETURNS: void
CALLED BY: control()
CALLS: none
COMMENTS: 22 Jan 93 - Dave MacPherson
*****************************************************************************/
void store_loc_trace_data(arg1, arg2, arg3, arg4)
double arg1, arg2, arg3, arg4;
{
        if (lop_tr == 0)
        {
        if (scale_tr == 1)
        {
        *(indxloc++) = time_tr;
        } else
```

```
        {
        *(indxloc++) = vehicle.x;
        }
        if ((pattern_tr & 1) != 0)
        {
        *(indxloc++) = vehicle.x;
        }
        if ((pattern_tr & 2) != 0)
        {
        *(indxloc++) = vehicle.y;
        }
        if ((pattern_tr & 4) != 0)
        {
        *(indxloc++) = arg1;
        }
        if ((pattern_tr & 8) != 0)
        {
        *(indxloc++) = arg2;
        }
        if ((pattern_tr & 16) != 0)
        {
        *(indxloc++) = arg3;

        }
        if ((pattern_tr & 32) != 0)
        {
        *(indxloc++) = arg4;
        }
        trace_cnt++;
        lop_tr = smpl_tr;
        }
        lop_tr--;
        time_tr += 0.01;
}       /* end store_loc_trace_data */

/* control.c */
```

```
/*
*************************************************************************
 immediate.c
        Rev 0 May 15, 1993 by Dave MacPherson
*************************************************************************
*/

/*********** INCLUDED FUNCTIONS ************/

 /******** IMMEDIATE FUNCTIONS *********
        stop0()
        get_rob(p)
        get_rob0(p)
        speed0(s);
        r_speed0(s);
        acc0(a);
        r_acc0(a);
        skip();
        get_line();
        halt()
        resume()
        sync()
        path_length();
        flush();

*************************************************************/

#include "mml.h"
#include "spatial.h"


/*
*************************************************************************

 IMMEDIATE FUNCTIONS

*************************************************************************
*/


/*************************************************************************
FUNCTION: path_length (immediate)
PARAMETERS: none
PURPOSE: get the total path length traveled by the robot
RETURNS: double
CALLED BY: user
CALLS: none
COMMENTS: 7 Jan 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
double path_length()
```

```
double arg_;
{
        racc = arg_; /* set the robot's translational acceleration */
} /* end racc0 */


/*************************************************************************
NAME : get_line0
ARGUMENTS : none
FUNCTION : return a pointer to the path element the robot
is currently tracking.
*************************************************************************/
CONFIGURATION get_line0()
{
        return get_inst->c;
} /* end get_line0 */


/*************************************************************************
NAME : stop0
ARGUMENTS : none
FUNCTION : stops the robot and flushes the instruction buffer.
*************************************************************************/
void stop0()
{
        vel_g = 0.0; /* set robot goal velocity to zero */

        /* flush instruction buffer here */
        head_inst = put_inst = get_inst = &inst_buf[0];
        inst_cnt = 0;
        tail_inst = &inst_buf[INST_MAX-1];
        head_len = put_len = get_len = &length_buf[0];
        tail_len = &length_buf[INST_MAX-1];

        seq_status = SSTOP;
        r_printf("\12 Entered the stop0() function.");
} /* end stop0 */


/*************************************************************************
FUNCTION: flush
PARAMETERS: none
PURPOSE: discards all buffer commands after current_robot_path

RETURNS: void
CALLED BY: user
CALLS: imaskoff()?
COMMENTS: 11 Jun 93 -- Bob Fish
TASK: Level 0
*************************************************************************/
```

```
void flush()
{
 int i;

 /* First step is to reset put_inst to be the same as get_inst.
 That way, the next motion command loaded on the buffer will be
 stored after the path currently being tracked.
 Second step is to reset global last_robot_path_element to be the
 current_robot_path. That way, when a new motion command is issued,
 the transition point calculations will be between the current path and
 the new path. no_o_paths is set to 1, because now there is only one
 path, the current one. */

        i = imaskoff();

 put_inst = get_inst;
 no_o_paths = 1;
 last_robot_path_element = current_robot_path;

 r_printf ("\nLRP.type=> ");
 r_printfi(last_robot_path_element.type);
 r_printf ("\nLRP.x=> ");
 r_printfr(last_robot_path_element.pc.x, 2);
 r_printf (" y=> ");
 r_printfr(last_robot_path_element.pc.y, 2);
 r_printf (" th=> ");
 r_printfr(last_robot_path_element.pc.t, 2);

        imaskon(i);

 return;
 }/* end flush */



size_const0(size)
double size;
{
 double kk;

 DIST_CONSTANT = size;
 kk = 1.0 / DIST_CONSTANT;
 aa = 3.0 * kk;
 bb = aa * kk;
 cc = bb * kk / 3.0;
 } /* end size_const0 */

/*
**********************************************************************
NAME : get_rob0
ARGUMENTS : A pointer to a CONFIGURATION
```

```
FUNCTION : return current odometry estimate from the controller.
*****************************************************************************
*/
CONFIGURATION *get_rob0(p)
CONFIGURATION *p;
{
        int i;

        i = imaskoff();
        p->x = vehicle.x;
        p->y = vehicle.y;
        p->t = vehicle.t;
        imaskon(i);
        return (p);
} /* end get_rob0 */


/*
*****************************************************************************
NAME : set_rob
FUNCTION : Set postures of cur.
*****************************************************************************
*/
void set_rob0(p)
CONFIGURATION *p;
{
        int i;
        i = imaskoff();
        set_P.x = p->x;
        set_P.y = p->y;
        set_P.t = p->t;
        setting_configuration = YES;
        imaskon(i);
} /* end set_rob */

double halt_speed = NEGATIVE_SPEED;

/*
*****************************************************************************
NAME: halt
 This function brings the robot to a smooth stop and places it
 in a dormant state. The robot will not respond to any other
 commands until resume() is called. All motion parameters are
 restored by resume() to their values prior to the call to halt().
*****************************************************************************
*/
void halt()
{
        if (halt_speed > 0.0)
        return;
        halt_speed = vel_g;
```

```
        vel_g = 0.0;
} /* end halt */

/*
*****************************************************************************
NAME: resume
 This function resets the robot's motion which was suspended
 by the call to halt() to the last user values.
*****************************************************************************
*/
resume()
{
        if (halt_speed < 0.0)
        return;
        vel_g = halt_speed;
        halt_speed = NEGATIVE_SPEED;
} /* resume() */

/* end immediate.c */
```

```
/*
*************************************************************************
 sequential.c
        Rev 0 May 15, 1993 by Dave MacPherson
*************************************************************************
*/

/*********** INCLUDED FUNCTIONS ************/

 /******** SEQUENTIAL FUNCTIONS *********
        speed(arg_)
        acc(arg_)
        rotate(thetasp)
        r_speed(arg_)
        r_acc(arg_)
        mark_motion()
        wait_motion()
        config(arg_)
        line(arg_)
        bline(arg_)
        fline(arg_)
        switch_dir()
        set_rob(pst)
        size_const(arg_)

*************************************************************/

#include "mml.h"
#include "spatial.h"


*******************************************************/
void set_error(code)
int code;
{
 PATH_ELEMENT path;

        path.type = ERROR;
        path.mode = code;
 set_inst(path);
} /* end set_error() */


/*************************************************************************
FUNCTION: size_const sequential
PARAMETERS: size
PURPOSE: Sets the parameter DIST_CONSTANT in a sequential
fashion. This determines how sharply Yamabico turns.
RETURNS: void
CALLED BY: user
CALLS: set_inst();
```

```
COMMENTS: 7 Jan 93 - Dave MacPherson
        **** DOES NOT LOAD INST BUFFER AS ADVERTIZED *********
TASK: Level 0
*****************************************************************************/
size_const(size)
double size;
{
        PATH_ELEMENT path;
 double kk;

 DIST_CONSTANT = size;
 kk = 1.0 / DIST_CONSTANT;
 aa = 3.0 * kk;
 bb = aa * kk;
 cc = bb * kk / 3.0;
        path.type = SIZE;
        path.pc.x = size;

        set_inst(path);
} /* end size_const() */


/*****************************************************************************
FUNCTION: speed (sequential)
PARAMETERS: arg_
PURPOSE: to set robot's speed
RETURNS: void
CALLED BY: user
CALLS: set_inst
COMMENTS: 7 Jan 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
speed(arg_)
double arg_;
{
        PATH_ELEMENT path;

        path.type = SPEED;
        path.pc.x = arg_;

        set_inst(path);
} /* end speed() */


/*****************************************************************************
FUNCTION: acc (sequential)
PARAMETERS: arg_
PURPOSE: to set robot's acceleration for speed changes
and stopping.
RETURNS: void
CALLED BY: user
```

```
CALLS: set_inst
COMMENTS: 7 Jan 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void acc(arg_)
double arg_;
{
        PATH_ELEMENT path;

        path.type = ACC;
        path.pc.x = arg_;

        set_inst(path);
} /* end acc() */

/*****************************************************************************
FUNCTION: rotate (sequential)
PARAMETERS: arg_
PURPOSE: Rotate the robot by thetasp radians.
Positive is counterclockwise and negative is
clockwise.
RETURNS: void
CALLED BY: user
CALLS: set_inst
COMMENTS: 24 March 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void rotate(thetasp)
double thetasp;
{
        PATH_ELEMENT path;


        if (seq_status != SSTOP && seq_status != SBLINE)
        /* robot must be stopped to shift to rotate */
         {
         set_error(ECODE2);
         return;
         }
        /*thetasp = d2r(thetasp); 04/15/92 */
        if (fabs(thetasp) < 0.0001)return;
        path.type = ROTATE;
        path.pc.x = thetasp;

        set_inst(path);

last_robot_path_element.pc.t += thetasp;
last_robot_path_element.type = SET_ROB;

        nom_p->t += thetasp;
        seq_status = SSTOP;
```

```
} /* end rotate() */

/*****************************************************************************
FUNCTION: rspeed (sequential)
PARAMETERS: arg_
PURPOSE: to set robot's angular speed to be used
when the robot performs a stationary rotation. The parameter
for rotational speed is in radians/second.
RETURNS: void
CALLED BY: user
CALLS: set_inst
COMMENTS: 8 Jan 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void r_speed(arg_)
double arg_;
{
        PATH_ELEMENT path;

        path.type = RSPEED;
        path.pc.x = arg_;

        set_inst(path);
} /* end r_speed() */

/*****************************************************************************
FUNCTION: racc (sequential)
PARAMETERS: arg_
PURPOSE: to set robot's angular acceleration to be used
wheel the robot performs a stationary rotation.
RETURNS: void
CALLED BY: user
CALLS: set_inst
COMMENTS: 8 Jan 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void r_acc(arg_)
double arg_;
{
        PATH_ELEMENT path;

        path.type = RACC;
        path.pc.x = arg_;

        set_inst(path);
} /* end r_acc() */

/*************************************************/
/* designate syncronization to yamabico.*/
/*      Jan. 23 89*/
```

```
/*************************************************/
mark_motion()
{
        msyn_q = 1;
}

/*************************************************/
/* execute syncronization*/
/*      Jan. 23 89*/
/*************************************************/
wait_motion()
{
        wsyn_q = 1;
        while(wsyn_q!=0);
}



/*****************************************************************************
FUNCTION: skip()
PARAMETERS: none
PURPOSE: Causes the robot to skip the next sequential motion
command.
RETURNS: void
CALLED BY: user()
CALLS: none
GLOBALS: skip_flag
COMMENTS: 26 Feb 93 - Dave MacPherson
*****************************************************************************/
void skip()
{
 skip_flag = TRUE;
}


/*****************************************************************************
FUNCTION : config(arg_)
PARAMETERS: configuration arg_
PURPOSE: Implements users command to move to a
 specified configuration using
 one or two cubic spirals.
RETURNS: void
CALLED BY: user.c
CALLS: solve (located in file cubic.c)
GLOBALS: seq_status - set;
COMMENTS: 8 Feb 93 -- Bob Fish
TASK: level 0, foreground job.
*****************************************************************************/
config(arg)
 CONFIGURATION *arg;
{
```

```
CONFIGURATION end_spiral;
CONFIGURATION start_spiral;
int res; /* flag to indicate success or failure of function solve */

        if (seq_status == SLINE)
        {
        set_error(ECODE3);
        r_printf("\n\nLINE to CONFIG Configuration Combination not Allowed.\n\n");
        /* exit(0); */
        }
        else if (seq_status == SPARABOLA)
        {
        set_error(ECODE3);
        r_printf("\n\nPARABOLA to CONFIG Configuration Combination not Al-
lowed.\n\n");
        }
        else if (seq_status == SFLINE)
        {
        set_error(ECODE3);
        r_printf("\n\nFLINE to CONFIG Configuration Combination not Allowed.\n\n");
        }
        else
        {


end_spiral = (*arg);

/* Values of start_spiral are obtained from the last motion instruction
in the buffer:
NOTE NOTE NOTE: this requires that the last motion instruction
be a legal precedent for a cubic spiral */

start_spiral.x = last_robot_path_element.pc.x;
start_spiral.y = last_robot_path_element.pc.y;
start_spiral.t = last_robot_path_element.pc.t;
start_spiral.k = 0.0;

/* Call solve with start_spiral and end_spiral as beginning and end of
the cubic spiral(s). */

res = solve(start_spiral, end_spiral);

/* 'res' can be used to see if the cubic spiral was successful. res may not
be useful for anything else that I can see, unless an error develops */

seq_status = SCONFIG;

/* Update last_robot_path_element to latest path */
last_robot_path_element.pc = end_spiral;
last_robot_path_element.type = SCONFIG;
```

```
return;
        }

} /* end config() */


/*
*****************************************************************************
NAME : line configuration
ARGUMENTS : configuration of path the robot must follow
FUNCTION : to move robot to a specified path
*****************************************************************************
*/
line(arg_)
CONFIGURATION *arg_;
{
        PATH_ELEMENT path;

        path.type = LINE;
        path.pc = (* arg_);
        if (no_o_paths == 0 || skip_flag)
        {
r_printf("\nFirst path, no transition point");
        last_robot_path_element.pc = path.pc;
        last_robot_path_element.type = SLINE;
        no_o_paths = 1;
        }
        else
        {

        path.tp =
        get_transition_point(last_robot_path_element, path);

        r_printf("\nTransition point to line\n x = ");
        r_printfr(path.tp.x0, 2);
        r_printf(" y = ");
        r_printfr(path.tp.y0, 2);
        ++no_o_paths;
        last_robot_path_element.pc = (*arg_);
        last_robot_path_element.type = SLINE;

        }
        set_inst(path);
        seq_status = SLINE;

} /* end line() */

/*
```

```
**************************************************************************
NAME : backward line configuration
ARGUMENTS : configuration of path the robot must follow
FUNCTION : to move robot to a specified path
**************************************************************************
*/
void bline(arg_)
CONFIGURATION *arg_;
{
        PATH_ELEMENT path;

        path.type = BLINE;
        path.pc = (* arg_);
        if (no_o_paths == 0 || skip_flag)
        {
        last_robot_path_element.pc = path.pc;
        last_robot_path_element.type = SBLINE;
        no_o_paths = 1;
        }
        else
        {
        path.tp =
        get_transition_point(last_robot_path_element, path);

        r_printf("\nTransition point to bline\n x = ");
        r_printfr(path.tp.x0, 2);
        r_printf(" y = ");
        r_printfr(path.tp.y0, 2);

        ++no_o_paths;
        last_robot_path_element.pc = (*arg_);
        last_robot_path_element.type = SBLINE;
        }
        /* set the robot's desired path to the value of arg_ */

        set_inst(path);
        seq_status = SBLINE;

} /* end bline() */

/**************************************************************************
FUNCTION: fline (arg_)
PARAMETERS: arg_
PURPOSE: To cause the robot to follow a straight line starting at a specific point.
RETURNS: void
CALLED BY: user()
CALLS: config, line
GLOBALS: none
COMMENTS: 24 Feb 93 - Bob Fish. Since this is a compound command, printouts
 will indicate a cubic spiral followed by a line.
**************************************************************************/
```

```
fline(arg_)
CONFIGURATION *arg_;
{

 if (seq_status == SLINE)
        {
        set_error(ECODE3);
        r_printf("\n\nLINE to FLINE Configuration Combination not Allowed.\n\n");
        }
        else if (seq_status == SPARABOLA)
        {
        set_error(ECODE3);
        r_printf("\n\nPARABOLA to FLINE Configuration Combination not Al-
lowed.\n\n");
        }
        else if (seq_status == SFLINE)
        {
        set_error(ECODE3);
        r_printf("\n\nFLINE to FLINE Configuration Combination not Allowed.\n\n");
        }
        else
        {
 /* This is implemented as a compound command. First call config to
 generate a cubic spiral path to the specified point configuration,
 then call line to track the line that goes through that point,
 with theta and kappa as specified.
 NOTE: This precludes some path reporting capability, since from this
 point on, the concept of fline is lost,
 it is replaced by config and line. */

        config(arg_);
        line(arg_);

         }
}


/*
*************************************************************************
NAME : parabola point directrix
ARGUMENTS : configuration of path the robot must follow
FUNCTION : to move robot to a specified path
*************************************************************************
*/
parabola(focus, directrix)
POINT *focus;
CONFIGURATION *directrix;
{
        PATH_ELEMENT path;

        if (seq_status == SPARABOLA)
```

```
        {
        set_error(ECODE3);
        r_printf("\n\nLINE to PARABOLA Configuration Combination not Al-
lowed.\n\n");
        }
        else if (seq_status == SCONFIG)
        {
        set_error(ECODE3);
        r_printf("\n\nCONFIG to PARABOLA Configuration Combination not Al-
lowed.\n\n");
        }
        else
        {
        path.type = PARABOLA;
        path.pc = (* directrix);
        path.pp = (* focus);

        set_inst(path);

        /* set the robot's desired path to the value of arg_ */
        seq_status = SPARABOLA;
        }
}


/*************************************************************************
FUNCTION: move_hall_follower()
PARAMETERS: arg_ (distance to walls)
PURPOSE: Causes the robot to follow a hallway by
perform odometry corrections in the background.
RETURNS: void
CALLED BY: user()
CALLS: set_inst();
GLOBALS: none
COMMENTS: 8 May 93 - Dave MacPherson
*************************************************************************/
void move_hall_follower(arg_)
double arg_;
{
        PATH_ELEMENT path;

        r_printf("Entered the move_hall_follower function");
        set_inst(SPWAY, arg_);
        seq_status = SPWAY;
} /* end move_hall_follower */



/*
*****************************************************************
```

```
NAME : switch_dir
ARGUMENTS : none
FUNCTION : to reverse the heading direction of the robot
*****************************************************************************
*/
switch_dir()
{
        PATH_ELEMENT path;

        if (seq_status != SSTOP)
         {
         set_error(ECODE2);
         return;
         }
 path.type = SWITCH;
        set_inst(path);
        nom_p->t = norm(nom_p->t + PI);
} /* end switch_dir() */

/*
*****************************************************************************
NAME : sync()
ARGUMENTS : none
FUNCTION : syncronizing to locomotion data update
         sync_loc flag is modified in _ih.loc routine
*****************************************************************************
*/
sync()
{
        sync_loc=0;
        while (sync_loc==0);
        return;
}

/*
*****************************************************************************
NAME : set_rob
ARGUMENTS : Configuration to set robot's location to.
FUNCTION : to add set robot sequence to queue
*****************************************************************************
*/
set_rob(pst)
CONFIGURATION *pst;
{
        PATH_ELEMENT path;

        if(seq_status!=SSTOP)
         {
         set_error(ECODE2);
         return;
         }
```

```
        nom_p->x = pst->x;
        nom_p->y = pst->y;
        nom_p->t = pst->t;
        nom_p->k = pst->k;
        length_norm = 0.0;

        path.type = SET_ROB;
        path.pc = (* pst);
        set_inst(path);

last_robot_path_element.pc = (* pst);
last_robot_path_element.type = SET_ROB;
}

/* end sequential.c */
```

```
/*
*****************************************************************************
 track.c
 last update April 23, 1993 by Dave MacPherson
*****************************************************************************
*/

#include "mml.h"


extern int transition_point_test();


/*****************************************************************************
FUNCTION: read_rotate
PARAMETERS: none
PURPOSE: Reads a rotate instruction. Starts the robot
rotating. The vehicle must be in the stop state in order
top start rotating.
RETURNS: void
CALLED BY:
CALLS: init_rotate()
COMMENTS: 27 December 92 - Dave MacPherson
*****************************************************************************/
void read_rotate()
{
        drvel = racc * INTVL;/* rotation control */
        th_g = vehicle.t + get_inst->c.x;
        goal_pst.t = th_g;
        goal_pst.x = vehicle.x;
        goal_pst.y = vehicle.y;
        if (th_g == vehicle.t)
        {
        change_status(SSTOP);
        return;
        }
        if (th_g > vehicle.t)
        raccdrc = POSITIVE;
        else
        raccdrc = NEGATIVE;
        change_status(RMOVE);
} /* end read_rotate() */

/*****************************************************************************
FUNCTION: cface2()
PARAMETERS: none
PURPOSE: Reverses the current robot direction of travel.
RETURNS: void
CALLED BY:
CALLS: norm
COMMENTS: 27 December 92 - Dave MacPherson
```

```
*************************************************************************/
void cface2()
{
        int i;

        i = imaskoff();

        cur_t = norm(cur_t + PI);
        vehicle.t = norm(vehicle.t + PI);
        mv_direction = - mv_direction;

        imaskon(i);
        return;
}


/*************************************************************************
FUNCTION: limit
PARAMETERS: double u
PURPOSE: limit functon for delta_d input=(delta_d)
output=(limited delta_d).
RETURNS: void
CALLED BY: update_kappa
CALLS: none
COMMENTS: 27 December 92 - Dave MacPherson
*************************************************************************/
double limit(u)
double u;
{
 if(u > 2.0 * DIST_CONSTANT) return(2.0*DIST_CONSTANT);
 if (u < -2.0*DIST_CONSTANT) return(-2.0*DIST_CONSTANT);
 return(u);
} /* end limit */


/*************************************************************************
FUNCTION: update_cubic_kappa
PARAMETERS: vehicle, current_image
PURPOSE: Main steering function for MML when using cubic spirals,
 uses a different ystar than other paths.
RETURNS: void
CALLED BY: stepper
CALLS: limit(),
COMMENTS: 6 Apr 93 - Bob Fish this is different than update_kappa()
 because Dr. K says ystar is different for cubic spirals than
 for lines and circles.
*************************************************************************/
double update_cubic_kappa(config, image)
CONFIGURATION config;
CONFIGURATION image;
{
```

```
register double dkappa1;
double cubic_ystar; /* ystar is different for cubic spiral than for lines&circles */

cubic_ystar = -(config.x-image.x)*sin(image.t) + (config.y-image.y)*cos(image.t);

dkappa1 = -aa * (config.k - image.k)
-bb * (norm( config.t - image.t))
-cc * limit(cubic_ystar);

        return config.k + dkappa1 * INTVL * vel_c;

} /* end update_cubic_kappa */

/
*****************************************************************************
FUNCTION: update_kappa
PARAMETERS: none
PURPOSE: Main steering function for MML.
RETURNS: void
CALLED BY: stepper
CALLS: limit(), update_image()
COMMENTS: 15 Feb 93 - Dave MacPherson
*****************************************************************************/
double update_kappa()
{
register double delta_d;
register double dkappa1;
        double update_delta_d();

current_image = update_image(vehicle, current_robot_path.pc);

delta_d = update_delta_d(vehicle, current_robot_path.pc);

dkappa1 = -aa * (vehicle.k - current_image.k)
-bb * (norm(vehicle.t - current_image.t))
-cc * limit(delta_d);

return vehicle.k + dkappa1 * delta_dist;
} /* update_kappa */

/*****************************************************************************
FUNCTION: update_delta_d()
PARAMETERS: config, path
PURPOSE: calculates the ystar for update_kappa()
RETURNS: double
CALLED BY: update_kappa
CALLS: sin, cos
COMMENTS: 15 Feb 93 - Dave MacPherson
*****************************************************************************/
double update_delta_d(config, path)
CONFIGURATION config;
```

```
CONFIGURATION path;
{
        double delta_d;

 delta_d = (-(config.x - path.x) * (path.k *
 (config.x - path.x) + 2 * sin(path.t)) -
 (config.y - path.y) * (path.k *
 (config.y - path.y) - 2 * cos(path.t))) /
 (1 + sqrt((path.k *(config.x - path.x)+
 sin(path.t))*
 (path.k *(config.x - path.x)+
         sin(path.t))
 + ((path.k * (config.y - path.y) -
         cos(path.t))*
 (path.k * (config.y - path.y) -
        cos(path.t)))));

         return delta_d;
} /* end update_delta_d() */



/*************************************************************************
FUNCTION: transition_point_test
PARAMETERS: image, tp
PURPOSE: Tests to determine if the robot's image
 is at or passed the transition point
RETURNS: int (1 = at or passed the transition point, 0 = otherwise)
CALLED BY:
CALLS:
COMMENTS: 15 Feb 93 - Dave MacPherson
 27 May 93 - Revision 1 Bob Fish, modified to check for already
 past the transition point.
*************************************************************************/
int transition_point_test(image, tp)
CONFIGURATION image;
POINT tp;
{

 /* Note: this needs to be modified, so that distance to the
 transition point is calculated using path distance vice
 EU_DIS. */


 double current_dist;

 current_dist = EU_DIS(image.x, image.y, tp.x0, tp.y0);

 if (fabs(current_robot_path.pc.k) < ZERA)
        {
```

```
/* current path is a straight line, check to see if close to or
past the transition pt. */


if (current_dist < 1.0)
        {
i_am_here = 15;
return 1;
         }

else if (current_dist > last_dist+.05)
{
i_am_here = 19;
         return 1;
         }
         else
{
i_am_here = 13;
         last_dist = current_dist;
         return 0;

         } /* end if */
}

else /* path is a circle, use the transition point as the only test */

         {
         if (current_dist < 1.0)
         return 1;
         else
         return 0;

         } /* end if */

} /* end transition point test */



/*
************************************************
 detect end motion
 then check next instruction
************************************************
*/
end_of_motion()
{
         if ((msyn_m!=0) && (wsyn_q!=0))
         {
         msyn_m = 0;
         wsyn_q = 0;
```

```
        }
}

/*
*************************************************
 set length_stop
*************************************************
*/
set_length_stop(class)
int     class;
{
 if(put_len==get_len)length_stop=INFINITE;
 else
 {
 length_stop=*get_len;
 if(class==STOP)
        {
         if(++get_len>tail_len)get_len=head_len;
        }
 }
}


/*****************************************************************************
FUNCTION: disp_error
PARAMETERS: code
PURPOSE: Reports locomotion errors.
RETURNS: void
CALLED BY:
CALLS:
COMMENTS: 27 December 92 - Dave MacPherson
*****************************************************************************/
void disp_error(code)
int code;
{
        switch (code)
        {
        case ECODE0:
        r_printf("\n postures too close ");
        break;
        case ECODE1:
        r_printf("\n bad cubic spiral specification ");
        break;
        case ECODE2:
        r_printf("\n SSTOP function detected in moving state ");
        break;
        default:
        r_printf("\n undefined error code detected ");
        }
}
```

```
/*****************************************************************************
FUNCTION: change_status
PARAMETERS: new_status
PURPOSE: Reports new locomotion status when the status changes.
RETURNS: void
CALLED BY:
CALLS:
COMMENTS: 27 December 92 - Dave MacPherson
*****************************************************************************/
void change_status(new_status)
int     new_status;
{
 status = new_status;
 if (status == SSTOP) wait_cnt = 100;
        /*changed wait_cnt from 400 to 100 31 May 1992*/

#ifdef SIM
 switch (status)
 {
 case SSTOP:
 r_printf("\nSSTOP\n");
 break;
 case SLINE:
 r_printf("\nSLINE\n");
 break;
 case SBLINE:
 r_printf("\nSBLINE\n");
 break;
 case SFLINE:
 r_printf("\nSFLINE\n");
 break;
 case SCONFIG:
 r_printf("\nSCONFIG\n");
 break;
 case RMOVE:
 r_printf("\nRMOVE\n");
 break;
 case SERROR:
 r_printf("\nSERROR\n");
 break;
 default:
 break;
 }
#endif
} /* end change_status */
/* end track.c */
```

```
/*
*****************************************************************************
 velocity.c
 last update May 24, 1993 by Dave MacPherson
*****************************************************************************
*/

#include "mml.h"


/
*****************************************************************************
FUNCTION: update_vel()
PARAMETERS: none
PURPOSE: Determines the current robot translational velocity.
RETURNS: double
CALLED BY: control()
CALLS: rest_of_path()
COMMENTS: 24 May 93 - Dave MacPherson
TASK: Level 4
*****************************************************************************/
double update_vel()
{
        double vel_gg; /* temporary goal velocity */
        double rest_of_path();

        dvel = tacc * INTVL;

 if (status == SBLINE &&
        2.0 * tacc * rest_of_path(current_robot_path,current_image)
 <= vel_c * vel_c)
        {
        vel_c = max2(vel_c - dvel, 0.0);
        }
        else
        {
        vel_gg = min2(vel_g, WHEEL_MAX / (1 + TREAD / 2 * fabs(kappa)));
        if (vel_gg >= vel_c)
        vel_c = min2(vel_c + dvel, vel_gg);
        else
        vel_c = max2(vel_c - dvel, vel_gg);
        }
 delta_dist = INTVL * vel_c;
        return vel_c;
} /* end update_vel() */


/*****************************************************************************
FUNCTION: rest_of_path()
PARAMETERS: path, image
PURPOSE: Determines the distance remaining on the
```

```
current_robot path.
RETURNS: double
CALLED BY: update_vel()
CALLS: none
COMMENTS: 24 May 93 - Dave MacPherson
TASK: Level 4
*****************************************************************************/
double rest_of_path(path, image)
PATH_ELEMENT path;
CONFIGURATION image;
{
/*
        switch(status)
        {
        case BLINE:
*/
         return ((path.pc.x - image.x)*cos(image.t) +
 (path.pc.y - image.y)*sin(image.t));
/*
        break;
        case CUBIC:
        break;
        }
*/
} /* end rest_of_path */


/*****************************************************************************
FUNCTION: get_rotational_vel()
PARAMETERS: none
PURPOSE: Determines the required rotational robot
velocity when the robot is rotating.
RETURNS: rvel_c
CALLED BY:
CALLS: min2(), max2()
COMMENTS: 22 Apr 93 - Dave MacPherson
 *****************************************************************************/
double get_rotational_vel()
{
        if (2.0 * racc * fabs(th_g - vehicle.t) > rvel_c * rvel_c)
        {
        if (raccdrc == POSITIVE) /* CCW rotation */
        rvel_c = min2(rvel_c + drvel, rvel);
        else /* clockwise rotation */
        rvel_c = max2(rvel_c - drvel, -rvel);
        }
        else /* robot rotational deceleration */
        if (raccdrc == POSITIVE)
        {
        if (vehicle.t < th_g)
        rvel_c = max2(rvel_c - drvel, 0.01);
```

```
        else
        {
        rvel_c = 0.0;
        change_status(SSTOP);
        read_inst();
        }
        }
        else /* CW rotation */
        {
        if (vehicle.t > th_g)
        rvel_c = min2(rvel_c + drvel, -0.01);
        else
        {
        rvel_c = 0.0;
        change_status(SSTOP);
        read_inst();
        }
        }

        return rvel_c;
} /* end get_rotational_vel() */

/* end velocity.c */
```

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C. SONAR SOURCE CODE

```
/* sonar2.c */

/* ultrasonic rangefinder functions */
/* 21 July 92 - Modified to discard sonar returns greater than 4.0 meter
        from the robot when building line segments - line 531 */
#include "mml.h"
#include "cartography.h"

#define print_flex(x,y) y = putstr(" ", putstr(rtoae((double) (x), tmpstr, 4), y))
#define nl_flex(x) x = putstr("\n", x)

/*declaration of functions and return values*/

extern double sonar();
extern void enable_sonar();
extern void disable_sonar();
extern double wait_sonar();
extern posit global ();
extern void enable_linear_fitting();
extern void disable_linear_fitting();
extern void enable_data_logging();
extern void disable_data_logging();
extern void serve_sonar();
extern LINE_SEG *get_segment();
extern LINE_SEG *get_current_segment();
extern void set_parameters();
extern void enable_interrupt_operation();
extern void disable_interrupt_operation();
extern void calculate_global();
extern void linear_fitting();
extern void start_segment();
extern void add_to_line();
extern LINE_SEG *end_segment();
extern void build_list();
extern void log_data();
extern void set_log_interval();
extern void wait_until();
extern void xfer_raw_to_host();
extern void xfer_global_to_host();
extern void xfer_segment_to_host();
extern void xfer_world_to_host();
extern void host_xfer();
extern void finish_segment();

/*****************************************************************************/
```

```
/*
/* Procedure: sonar(n)
/*
/* Description: returns the distance (in centimeters) sensed by the
/* n_th ultrasonic sensor. If no echo is received, then a -1 is
/* returned. If the distance is less than 10 cm, then a 0 is
/* returned.
/*
/*****************************************************************************/

double sonar(n)
int n;
{
        return sonar_table[n].d;
}

/*****************************************************************************/
/*
/* Procedure: enable_sonar(n)
/*
/* Description: enables the sonar group that contains sonar n, which
/* causes all the sonars in that group to echo-range and write data
/* to the data registers on the sonar control board. Marks the n'th
/* position of the enabled_sonars array to track which sonars are
/* enabled.
/*
/*****************************************************************************/

void enable_sonar(n)
int n;
{

        int i;

        i = imaskoff();
        enabled_sonars[n] = 1;
        switch (n)
        {
        case 0:
        case 2:
        case 5:
        case 7:
        enabled = enabled | 0x01;
        break;
        case 1:
        case 3:
        case 4:
        case 6:
        enabled = enabled | 0x02;
        break;
        case 8:
```

```
        case 9:
        case 10:
        case 11:
        enabled = enabled | 0x04;
        break;
        case 12:
        case 13:
        case 14:
        case 15:
        enabled = enabled | 0x08;
        break;
        }
        *command_ptr = enabled;
        imaskon(i);
}

/*************************************************************************/
/*
/* Procedure: disable_sonar(n)
/*
/* Description: removes the sonar n from the enabled_sonars list. If
/* sonar n is the only enabled sonar from it's group, then the
/* group is disabled as well and will stop echo ranging. This has
/* benefit of shortening the ping interval for groups that remain
/* enabled.
/*
/*************************************************************************/

void disable_sonar(n)
int n;
{
        int i, c;
        char mask;

        i = imaskoff();
        enabled_sonars[n] = 0;
        switch (n)
        {
        case 0:
        case 2:
        case 5:
        case 7:
        c = enabled_sonars[0] + enabled_sonars[2] +
        enabled_sonars[5] + enabled_sonars[7];
        if (c == 0)
        enabled = enabled & 0xfe;
        break;
        case 1:
        case 3:
        case 4:
        case 6:
```

```
        c = enabled_sonars[1] + enabled_sonars[3] +
        enabled_sonars[4] + enabled_sonars[6];
        if (c == 0)
        enabled = enabled & 0xfd;
        break;
        case 8:
        case 9:
        case 10:
        case 11:
        c = enabled_sonars[8] + enabled_sonars[9] +
        enabled_sonars[10] + enabled_sonars[11];
        if (c == 0)
        enabled = enabled & 0xfb;
        break;
        case 12:
        case 13:
        case 14:
        case 15:
        c = enabled_sonars[12] + enabled_sonars[13] +
        enabled_sonars[14] + enabled_sonars[15];
        if (c == 0)
        enabled = enabled & 0xf7;
        break;
        }
        *command_ptr = enabled;
        imaskon(i);
}


/*************************************************************************/
/*
/* Procedure: wait_sonar(n)
/*
/* Description: waits in a loop until new data is available for
/* sonar n.
/*
/*************************************************************************/

double wait_sonar(n)
int n;
{
        int a = 0;

        return sonar_table[n].d;
}

/*************************************************************************/
/*
/* Procedure: global(n)
/*
```

```
/* Description: returns a structure of type posit containing the global
/* x and y coordinates of the position of the last sonar return.
/*
/*************************************************************************/

posit global (n)
int n;
{
        posit answer;

        if (sonar_table[n].global == 0)
        calculate_global(n);
        answer.gx = sonar_table[n].gx;
        answer.gy = sonar_table[n].gy;
        answer.psi = sonar_table[n].t + sonar_table[n].axis;
        return answer;
}

/*************************************************************************/
/*
/* Procedure: enable_linear_fitting(n)
/*
/* Description: causes the background system to gather data points
/* from sonar n and form them into line segments as governed by
/* the linear fitting algorithm.
/*
/*************************************************************************/

void enable_linear_fitting(n)
int n;
{
        sonar_table[n].fitting = 1;
        sonar_table[n].global = 1;
}

/*************************************************************************/
/*
/* Procedure: disable_linear_fitting(n)
/*
/* Description: causes background system to cease forming line
/* segments for sonar n.
/* Will also disable the calculation of global coordinates for
/* that sonar if data logging of global data is not enabled.
/*
/*************************************************************************/

void disable_linear_fitting(n)
int n;
{
        sonar_table[n].fitting = 0;
        if (sonar_table[n].filetype[1] == 0)
```

```
        sonar_table[n].global = 0;
}

/*************************************************************************/
/*
/* Procedure: enable_data_logging(n,filetype,filenumber)
/*
/* Description: causes the background system to log data for sonar (n)
/* to a file (filenumber). The data to be logged is specified by an
/* integer flag (filetype). A value of 0 for filetype will cause raw
/* sonar data to be saved, 1 will save global x and y, and 2 will
/* save line segments. The filenumber may range between 0 and 3 for
/* each of the three types, providing up to 12 data files. Example:
/* enable_data_logging(4,1,0);
/* will cause raw data from sonar #4 to be saved to file 0, while:
/* enable_data_logging(7,2,0);
/* will cause segments for sonar #7 to be saved to file 0.
/*
/*************************************************************************/

void enable_data_logging(n, filetype, filenumber)
int n, filetype, filenumber;
{
        if (filetype == 1)
        sonar_table[n].global = 1;

        sonar_table[n].filetype[filetype] = 1;
        sonar_table[n].filenumber[filetype] = filenumber;
}

/*************************************************************************/
/*
/* Procedure: disable_data_logging(n,filetype)
/*
/* Description: causes the background system to cease logging data of a
/* given filetype for a sonar n.
/*
/*************************************************************************/

void disable_data_logging(n, filetype)
int n, filetype;
{
        if ((filetype == 1) && (sonar_table[n].fitting == 0))
        sonar_table[n].global = 0;

        sonar_table[n].filetype[filetype] = 0;
}

/*************************************************************************/
/*
/* Procedure: serve_sonar(x,y,t,ovfl,data1,data2,data3,data4,group)
```

```
/*
/* Description: this procedure is the "central command" for the
/* control of all sonar related functions. It is linked with
/* the ih_sonar routine and loads sonar data to the sonar_table
/* from there. It then examines the various control flags in the
/* sonar_table to determine which activities the user wishes to
/* take place, and calls the appropriate functions. This procedure
/* is invoked approximately every thirty milliseconds by an
/* interrupt from the sonar control board.
/*
/*************************************************************************/
void serve_sonar(x, y, t, ovfl, data4, data3, data2, data1, group)
double x, y, t;
int ovfl, data4, data3, data2, data1, group;
{

        int i, n;
        int data[4];
        int ovfl_mask = 8;

        data[0] = data1;
        data[1] = data2;
        data[2] = data3;
        data[3] = data4;

        for (i = 0; i < 4; i++, ovfl_mask /= 2)
        {
        n = group_array[group][i];/* n = sonar number */
        if (ovfl_mask & ovfl)
        sonar_table[n].d = -1.0;
        else if (data[i] < 100)
        sonar_table[n].d = 0.0;
        else
        sonar_table[n].d = (double) data[i] / 10.0;
        sonar_table[n].x = x;
        sonar_table[n].y = y;
        sonar_table[n].t = t;
        if (sonar_table[n].global == 1)
        calculate_global(n);
        if (sonar_table[n].fitting == 1)
        linear_fitting(n);
        if (sonar_table[n].filetype[0] == 1)
        log_data(n, 1, sonar_table[n].filenumber[0], 0);
        if (sonar_table[n].filetype[1] == 1)
        log_data(n, 2, sonar_table[n].filenumber[1], 0);
        }
}       /* serve_sonar() */

/*************************************************************************/
/*
/* Procedure: get_segment(n)
```

```
/*
/* Description: returns a pointer to the oldest segment on the linked
/* list of segments for sonar n; i.e. the record at the head
/* of the linked list. It is destructive, thus subsequent calls
/* will return subsequent segments until the list is empty. This is
/* accomplished by first copying the contents of the head record
/* into a temporary record called segstruct and then freeing the
/* allocated memory for the head record. The pointer returned is
/* actually a pointer to this temporary storage. If get_segment is
/* called on an empty list a null pointer is returned.
/*
/*************************************************************************/

LINE_SEG *get_segment(n)
int n;
{
        LINE_SEG *ptr;
        int index;

        index = seg_list_head[n];
        if (index == -1)
        ptr = NULL;
        else
        {
        ptr = &seg_list[n][index];
        seg_list_head[n] = (index < 4) ? (index + 1) : 0;
        }
        return ptr;
}

/*************************************************************************/
/*
/* Procedure: get_current_segment(n)
/*
/* Description: returns a pointer to the segment currently under
/* construction if there is one, otherwise returns null pointer.
/* This is accomplished by calling end_segment, copying the data
/* into segstruct and then returning a pointer to segstruct. The
/* memory allocated by end_segment is then freed.
/*
/*************************************************************************/

LINE_SEG *get_current_segment(n)
int n;
{
        LINE_SEG *ptr;

        ptr = end_segment(n);

        return ptr;
}
```

```
/*************************************************************************/
/*
/* Procedure: set_parameters(c1,c2,c3)
/*
/* Description: allows the user to adjust constants which control
/* the linear fitting algorithm. C1 is a multiplier for standard
/* deviation and C2 is an absolute value; both are used to
/* determine if an individual data point is usable for the
/* algorithm. C3 is a value for ellipse thinness; it is used to
/* determine the end of a segment. Default values are set in main.c
/* to 3.0, 5.0, and 0.1 respectively.
/*
/*************************************************************************/

void set_parameters(c1, c2, c3)
double c1, c2, c3;
{
        C1 = c1;
        C2 = c2;
        C3 = c3;
}

/*************************************************************************/
/*
/* Procedure: enable_interrupt_operation()
/*
/* Description: places sonar control board in interrupt driven mode.
/*
/*************************************************************************/

void enable_interrupt_operation()
{
        *BIM_ptr = *BIM_ptr | 0x10;
}

/*************************************************************************/
/*
/* Procedure: disable_interrupt_operation()
/*
/* Description: stops interrupt generation by the sonar control
/* board. A flag is set in the status register when data is ready,
/* and it is the user's responsibility to poll the sonar system
/* for the flag.
/*
/*************************************************************************/

void disable_interrupt_operation()
{
        *BIM_ptr = *BIM_ptr & 0xef;
}
```

```
/*************************************************************************/
/*
/* Procedure: calculate_global(n)
/*
/* Description: this procedure calculates the global x and y coordinates
/* for the range value and robot configuration in the sonar table.
/* The results are stored in the sonar table.
/*
/*************************************************************************/

void calculate_global(n)
int n;
{
        double lx, ly, gt, range, phi, axis, offset;

        gt = sonar_table[n].t;
        range = sonar_table[n].d;
        phi = sonar_table[n].phi;
        axis = sonar_table[n].axis;
        offset = sonar_table[n].offset;

        if (range == -1)
        range = 9999;
        lx = sonar_table[n].x + (cos(gt + phi) * offset);/* global x position of
         * sonar */
        ly = sonar_table[n].y + (sin(gt + phi) * offset);/* global y position of
         * sonar */
        sonar_table[n].gx = lx + (cos(gt + axis) * range);/* global x position of
         * range */
        sonar_table[n].gy = ly + (sin(gt + axis) * range);/* global y position of
         * range */
}

/*************************************************************************/
/*
/* Procedure: linear_fitting(n)
/*
/* Revised by Y. Kanayama, 07-07-93
/*
/* Description: this procedure controls the fitting of point data to straight
/* line segments. First it tests if the new coming point is not far from
/* the fitted line. If the test is passed, the point is added to test
/* if the thinnes test is passed. If it is passed, the addition is
/* finalized.
/* If any of the tests fail, the line segment is ended and a new one
/* started. The completed line segment is stored in a data structure
/* called segment, and segments are linked together in a linked list.
/*
/*************************************************************************/
```

```
void linear_fitting(n)
int n;
{
        double x, y, m00, m10, m01, m20, m11, m02;
        double alpha, r, sigma, delta;
        LINE_SEG *finished_segment;

/*
        if (sonar_table[n].d < 9.3 || sonar_table[n].d > 409.0)
        {
        finish_segment(n);
        start_segment(n);
        return;
        }
        */
        x = sonar_table[n].gx;/* temporary moments */
        y = sonar_table[n].gy;
        m00 = segment_data[n].m00;
        if (m00 < 1.5)
        {
        add_to_line(n, x, y);
        return;
        }
        r = segment_data[n].r;/* m00 >= 2 */
        alpha = segment_data[n].alpha;
        delta = fabs(r - x * cos(alpha) - y * sin(alpha));

/*      sigma = sqrt( segment_data[n].m_major / (m00 - 1.0)); */
        if (delta > C2)
        {
/*       if (m00 > 10.0) */
        finish_segment(n);
        start_segment(n);
        add_to_line(n, x, y);
        return;
        } else
        {
        add_to_line(n, x, y);
        return;
        }
}       /* end linear_fitting */



/*****************************************************************************/
/*
/* Procedure: start_segment(n)
/*
/* Description: this procedure establishes a new line segment with the three
/* data points contained in segment_data[n].init(x and y). It writes
```

```
/* the appropriate data to the interim values in segment_data[n].
/*
/*************************************************************************/

void start_segment(n)
int n;
{
        segment_data[n].m00 = 0.0;
        segment_data[n].m10 = 0.0;
        segment_data[n].m01 = 0.0;
        segment_data[n].m20 = 0.0;
        segment_data[n].m11 = 0.0;
        segment_data[n].m02 = 0.0;
}




/*************************************************************************/
/*
/* Procedure: add_to_line(n, x, y)
/*
/* Description: this procedure calculates new interim data for the line segment
/* and stores it in segment_data[n]. It also changes the end point values to
/* the point being added.
/*
/*************************************************************************/

void add_to_line(n, x, y)
int n;
double x, y;
{
        double m00, m10, m01, m20, m11, m02;
        double m_major, m_minor, d_major, d_minor, alpha, r, rho;
        double mux, muy, mm20, mm11, mm02;

        m00 = segment_data[n].m00 += 1.0;
        m10 = segment_data[n].m10 += x;
        m01 = segment_data[n].m01 += y;
        m20 = segment_data[n].m20 += SQR(x);
        m11 = segment_data[n].m11 += x * y;
        m02 = segment_data[n].m02 += SQR(y);

        if (m00 < 1.5)
        {
        segment_data[n].startx = x;
        segment_data[n].starty = y;
        }
        mux = m10 / m00;
        muy = m01 / m00;
```

```
        mm20 = m20 - SQR(m10) / m00;
        mm11 = m11 - m10 * m01 / m00;
        mm02 = m02 - SQR(m01) / m00;
/*
        segment_data[n].m00 = m00;
        segment_data[n].m10 = m10;
        segment_data[n].m01 = m01;
        segment_data[n].m20 = m20;
        segment_data[n].m11 = m11;
        segment_data[n].m02 = m02;
*/
        if (m00 > 1.5)
        {
        m_major = (mm20 + mm02) / 2.0 - sqrt((mm02 - mm20) * (mm02 - mm20) / 4.0 +
SQR(mm11));
        m_minor = (mm20 + mm02) / 2.0 + sqrt((mm02 - mm20) * (mm02 - mm20) / 4.0
+ SQR(mm11));
        d_major = 4.0 * sqrt(fabs(m_minor / m00));
        d_minor = 4.0 * sqrt(fabs(m_major / m00));
        rho = d_minor / d_major;
        alpha = atan2(-2.0 * mm11, (mm02 - mm20)) / 2.0;
        r = mux * cos(alpha) + muy * sin(alpha);

        segment_data[n].alpha = alpha;
        segment_data[n].r = r;
        segment_data[n].m_major = m_major;
        segment_data[n].m_minor = m_minor;
        segment_data[n].d_major = d_major;
        segment_data[n].d_minor = d_minor;
        segment_data[n].rho = rho;
        segment_data[n].endx = x;
        segment_data[n].endy = y;
        }
}

/*************************************************************************/
/*
/* Procedure: end_segment(n)
/*
/* Description: this procedure allocates memory for the segment data structure,
/* loads the correct values into it and returns a pointer to the structure.
/*
/*************************************************************************/

LINE_SEG *end_segment(n)
int n;
{
        LINE_SEG *seg_ptr;
        double startx, starty, endx, endy, delta, alpha, r, length;

        seg_ptr = &segstruct;
```

```
        startx = segment_data[n].startx;
        starty = segment_data[n].starty;
        endx = segment_data[n].endx;
        endy = segment_data[n].endy;
        alpha = segment_data[n].alpha;
        r = segment_data[n].r;
        delta = startx * cos(alpha) + starty * sin(alpha) - r;
        startx = startx - (delta * cos(alpha));
        starty = starty - (delta * sin(alpha));
        delta = endx * cos(alpha) + endy * sin(alpha) - r;
        endx = endx - (delta * cos(alpha));
        endy = endy - (delta * sin(alpha));
        length = sqrt(SQR(startx - endx) + SQR(starty - endy));

        seg_ptr->headx = startx;
        seg_ptr->heady = starty;
        seg_ptr->tailx = endx;
        seg_ptr->taily = endy;
        seg_ptr->alpha = alpha;
        seg_ptr->r = r;
        seg_ptr->length = length;
        seg_ptr->dmajor = segment_data[n].d_major;
        seg_ptr->dminor = segment_data[n].d_minor;
        seg_ptr->sonar = n;

        return seg_ptr;
}       /* end end_segment */


/*****************************************************************************/
/*
/* Procedure: build_list(ptr, n);
/*
/* Description: this function accepts a pointer to a segment data structure and
/* a sonar number, and appends the segment structure to the tail of a linked
/* list of structures for that sonar.
/*
/*****************************************************************************/

void build_list(ptr, n)
int n;
LINE_SEG *ptr;
{
        int next;

        if (seg_list_tail[n] == -1)
        seg_list_head[n] = 0;
        next = (seg_list_tail[n] < 4) ? ++seg_list_tail[n] : 0;
        if (next == seg_list_head[n])
        seg_list_head[n] = (seg_list_head[n] < 4) ? ++seg_list_head[n] : 0;
```

```
        seg_list[n][next] = *ptr;
        if (sonar_table[n].filetype[2] == 1)
        log_data(n, 3, sonar_table[n].filenumber[2], next);
}

/*************************************************************************/
/*
/* Procedure: log_data(n, type, filenumber,i)
/*
/* Description: this procedure causes data to be written to a file. The filenumber
/* designates which "column" (0,1,2, or 3) of a two dimensional array for
/* that type of data is used. The data array and a counter for each column
/* forms the data structure for each type. The value of i is used to index
/* the seg_list array for storing line segments.
/*
/*************************************************************************/

void log_data(n, filetype, filenumber, i)
int n, filetype, filenumber, i;
{
        int count, interval, next;

        switch (filetype)
        {
        case 1:
        count = raw_data_log[filenumber].count;
        interval = sonar_table[n].interval;
        if ((count < MAXRAW) && !(count % interval))
        {
        next = raw_data_log[filenumber].next;
        raw_data_log[filenumber].darray[next] = sonar_table[n].d;
        raw_data_log[filenumber].xarray[next] = sonar_table[n].x;
        raw_data_log[filenumber].yarray[next] = sonar_table[n].y;
        raw_data_log[filenumber].tarray[next] = sonar_table[n].t;
        raw_data_log[filenumber].next += 1;
        }
        raw_data_log[filenumber].count += 1;
        break;
        case 2:
        count = global_data_log[filenumber].count;
        interval = sonar_table[n].interval;
        if ((count < MAXGLOBAL) && !(count % interval))
        {
        next = global_data_log[filenumber].next;
        global_data_log[filenumber].xarray[next] = sonar_table[n].gx;
        global_data_log[filenumber].yarray[next] = sonar_table[n].gy;
        global_data_log[filenumber].next += 1;
        }
        global_data_log[filenumber].count += 1;
        break;
        case 3:
```

```
        count = segment_data_log[filenumber].count;
        if (count < MAXSEGMENT)
        {
        segment_data_log[filenumber].array[count] = seg_list[n][i];
        }
        segment_data_log[filenumber].count += 1;
        break;
        }
}


/*************************************************************************/
/*
/* Procedure: set_log_interval(n,d)
/*
/* Description: this procedure allows the user to set how often the sonar system
/* writes data to the raw data or global data files. The interval d is stored
/* at sonar_table[n], and one data point will be recorded for every d data
/* points sensed by the sonar. Default value for interval d is 13, which for
/* a speed of 30 cm/sec and sonar sampling time of 25 msec should record a
/* data point every 10 cm.
/*
/*************************************************************************/

void set_log_interval(n, d)
int n, d;
{
        sonar_table[n].interval = d;
}


/*************************************************************************/
/*
/* Procedure: wait_until(variable,relation,value)
/*
/* Description: this procedure will delay it's completion (and thus the continuance
/* of the program it's embedded in) until the variable achieves the relation with
/* the value specified. For example, presume the robot is traveling along the x
/* axis. If the user wants the robot to begin redording sonar data when the x
/* position of the robot exceeds 500 cm., he would insert this command after the
/* move command:
/* wait_until(X,GT,500.0);
/* enable_sonar(sonar number);
/* The variable are predefined as X, Y, A and D0 through D11, and correspond to
/* the robot's current x position, y position, alpha, and range from sonars 0
/* through 11. Relations are predefined as GT, LT and EQ corresponding to greater
/* than, less than and equal to. Value may be any numlber expressed as a double
/* or the predefined values PI, HPI, PI34, PI4, or DPI.
/*
/*************************************************************************/
```

```
void wait_until(variable, relation, value)
int variable, relation;
double value;
{
        double *ptr;
        double theta;
        int test, item;

        if ((variable == 14) && (relation == 17))
        test = (int) (1000.0 * value);
        else if (relation == 17)
        test = (int) (value);

        switch (variable)
        {
        case 0:
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
        case 6:
        case 7:
        case 8:
        case 9:
        case 10:
        case 11:
        ptr = &sonar_table[variable].d;
        break;
        case 12:
        ptr = &vehicle.x;
        break;
        case 13:
        ptr = &vehicle.y;
        break;
        case 14:
        theta = 1000.0 * vehicle.t;
        ptr = &theta;
        break;
        }
        switch (relation)
        {
        case 15:
        do
        {
        item = *ptr;
        }
        while (item <= value);
        break;
        case 16:
        do
```

```
        {
        item = *ptr;
        }
        while (item >= value);
        break;
        case 17:
        do
        {
        item = (int) *ptr;
        }
        while (item != test);
        break;
        }
}

/*************************************************************************/
/*
/* Procedure: xfer_raw_to_host(filenumber,filename)
/*
/* Description: this function allocates memory for a buffer and then converts a raw data
/* log file to a string format stored in the buffer. It then calls host_xfer to send
/* the string to the host. When that transfer is complete, it frees the memory it
/* allocated for the buffer. Filename must be entered in double quotes ( "dumpraw"
/* for example).
/*
/*************************************************************************/

void xfer_raw_to_host(filenumber, filename)
int filenumber;
char *filename;
{
        char *rbuffer;
        char *start;
        int i, c, j;

        i = raw_data_log[filenumber].next;
        c = 20 + (i * 33);
        rbuffer = malloc(c);
        start = rbuffer;
        for (j = 0; j < i; j++)
        {
        print_flex(raw_data_log[filenumber].darray[j], rbuffer);
        print_flex(raw_data_log[filenumber].xarray[j], rbuffer);
        print_flex(raw_data_log[filenumber].yarray[j], rbuffer);
        print_flex(raw_data_log[filenumber].tarray[j], rbuffer);
        nl_flex(rbuffer);
        }
        putb('\0', rbuffer);
        rbuffer = start;
        host_xfer(rbuffer, filename);
        free(rbuffer);
```

```
}

/*************************************************************************/
/*
/* Procedure: xfer_global_to_host(filenumber,filename)
/*
/* Description: this function performs the same function as xfer_raw_to_host, but for
/* global data vice raw data.
/*
/*************************************************************************/

void xfer_global_to_host(filenumber, filename)
int filenumber;
char *filename;
{
        char *gbuffer;
        char *start;
        int i, c, j;

        i = global_data_log[filenumber].next;
        c = 20 + (i * 17);
        gbuffer = malloc(c);
        start = gbuffer;
        for (j = 0; j < i; j++)
        {
        print_flex(global_data_log[filenumber].xarray[j], gbuffer);
        print_flex(global_data_log[filenumber].yarray[j], gbuffer);
        nl_flex(gbuffer);
        }
        putb('\0', gbuffer);
        gbuffer = start;
        host_xfer(gbuffer, filename);
        free(gbuffer);
}

/*************************************************************************/
/*
/* Procedure: xfer_segment_to_host(filenumber,filename)
/*
/* Description: this function performs the
/* same function as xfer_raw_to_host, but for
/* segment data vice raw data.
/*
/*************************************************************************/

void xfer_segment_to_host(filenumber, filename)
int filenumber;
char *filename;
{
        char *segbuffer;
```

```
        char *start;
        int i, c, j;

        i = segment_data_log[filenumber].count;
        c = 20 + (i * 77);
        segbuffer = malloc(c);
        start = segbuffer;
        for (j = 0; j < i; j++)
        {
        print_flex(segment_data_log[filenumber].array[j].headx, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].heady, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].tailx, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].taily, segbuffer);
        nl_flex(segbuffer);
        print_flex(segment_data_log[filenumber].array[j].alpha, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].r, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].length, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].dmajor, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].dminor, segbuffer);
        nl_flex(segbuffer);
        }
        putb('\0', segbuffer);
        segbuffer = start;
        host_xfer(segbuffer, filename);
        free(segbuffer);
}

/
**************************************************************************
******/
/*
/* Procedure: xfer_world_to_host(world)
/*
/* Description: this function transfers the edges of
/* a partial world to the host.
/*
/
**************************************************************************
******/

void xfer_world_to_host(PW, filename)
Map_World *PW;
char *filename;
{
        char *segbuffer;
        char *start;
        int i, c, j, k;
        int l = 0;
        Map_Polygon *current_polygon;
        EDGE *current_edge;
```

```
        i = PW->boundary->degree;

        current_polygon = PW->hole_list;
        for (k = 1; k < PW->degree; k++)
        {
        i += current_polygon->degree;
        current_polygon = current_polygon->next;
        }

        c = 20 + (i * 43);
        segbuffer = malloc(c);
        start = segbuffer;

        /* Put the boundary polygon edges in the buffer */
        current_edge = PW->boundary->edge_list;
        for (j = 0; j < PW->boundary->degree; j++)
        {
        print_flex(current_edge->v1.x, segbuffer);
        print_flex(current_edge->v1.y, segbuffer);
        nl_flex(segbuffer);
        print_flex(current_edge->v2.x, segbuffer);
        print_flex(current_edge->v2.y, segbuffer);
        if (current_edge->type == REAL)
        print_flex(1.0, segbuffer);
        else
        print_flex(0.0, segbuffer);
        nl_flex(segbuffer);
        nl_flex(segbuffer);
        current_edge = current_edge->next;
        }

        /* Put the hole polygon edges in the buffer */
/*
        for (j = 0; j < l; j++)
        {
        print_flex(segment_data_log[filenumber].array[j].headx, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].heady, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].tailx, segbuffer);
        print_flex(segment_data_log[filenumber].array[j].taily, segbuffer);
        nl_flex(segbuffer);
        }
*/
        putb('\0', segbuffer);
        segbuffer = start;
        host_xfer(segbuffer, filename);
        free(segbuffer);
}

/*************************************************************************/
/*
/* Procedure: xfer_real_boundary_edges_to_host(world)
```

```
/*
/* Description: this function transfers the real edges of
/* the boundary polygon of a partial world to the host.
/*
/*****************************************************************************/

void xfer_real_boundary_edges_to_host(PW, filename)
Map_World *PW;
char *filename;
{
        char *edgebuffer;
        char *start;
        int c, j, k;
        int count = 0;
        Map_Polygon *current_polygon;
        EDGE *cur  nt_edge;


        current_edge = PW->boundary->edge_list;
        for (k = 1; k < PW->boundary->degree; k++)
        {
        if (current_edge->type == REAL)
        ++count;
        current_edge = current_edge->next;
        }

        c = 20 + (count * 35);
        edgebuffer = malloc(c);
        start = edgebuffer;

        /* Put the boundary polygon edges in the buffer */
        current_edge = PW->boundary->edge_list;
        for (j = 0; j < PW->boundary->degree; j++)
        {
        if (current_edge->type == REAL)
        {
        print_flex(current_edge->v1.x, edgebuffer);
        print_flex(current_edge->v1.y, edgebuffer);
        nl_flex(edgebuffer);
        print_flex(current_edge->v2.x, edgebuffer);
        print_flex(current_edge->v2.y, edgebuffer);
        nl_flex(edgebuffer);
        nl_flex(edgebuffer);
        current_edge = current_edge->next;
        }
        }

        putb('\0', edgebuffer);
        edgebuffer = start;
        host_xfer(edgebuffer, filename);
        free(edgebuffer);
```

```
}       /* end xfer_real_boundary_edges_to_host() */

/*****************************************************************************/
/*
/* Procedure: xfer_inferred_boundary_edges_to_host(world)
/*
/* Description: this function transfers the inferred edges of
/* the boundary polygon of a partial world to the host.
/*
/*****************************************************************************/

void xfer_inferred_boundary_edges_to_host(PW, filename)
Map_World *PW;
char *filename;
{
        char *edgebuffer;
        char *start;
        int c, j, k;
        int count = 0;
        Map_Polygon *current_polygon;
        EDGE *current_edge;


        current_edge = PW->boundary->edge_list;
        for (k = 1; k < PW->boundary->degree; k++)
        {
        if (current_edge->type == INFERRED)
        ++count;
        current_edge = current_edge->next;
        }

        c = 20 + (count * 35);
        edgebuffer = malloc(c);
        start = edgebuffer;

        /* Put the boundary polygon edges in the buffer */
        current_edge = PW->boundary->edge_list;
        for (j = 0; j < PW->boundary->degree; j++)
        {
        if (current_edge->type == INFERRED)
        {
        print_flex(current_edge->v1.x, edgebuffer);
        print_flex(current_edge->v1.y, edgebuffer);
        nl_flex(edgebuffer);
        print_flex(current_edge->v2.x, edgebuffer);
        print_flex(current_edge->v2.y, edgebuffer);
        nl_flex(edgebuffer);
        nl_flex(edgebuffer);
        current_edge = current_edge->next;
        }
        }
```

```
        putb('\0', edgebuffer);
        edgebuffer = start;
        host_xfer(edgebuffer, filename);
        free(edgebuffer);
}       /* end xfer_real_boundary_edges_to_host() */

/*************************************************************************/
/*
/* Procedure: host_xfer(buffer,filename)
/*
/* Description: this function transfers a data string from the buffer to the host. Not a
/* user function; is called by data conversion functions such as xfer_raw_to_host.
/* User would call the xfer_raw_to_host (or equivalent for global or segment data)
/* to download data from the robot.
/*
/*************************************************************************/

void host_xfer(buffer, filename)
char *buffer;
char *filename;
{
        i_port(HOST, 9600, 0, 0, 0);
        r_printf("\12\15 connect cable and keyin\" \"");
        while (r_getchar() != ' ');
        putstr("\n", HOST);
        i_port(HOST, 9600, 0, 0, 1);
        r_printf("\12\15 ready for dump ");
        while (r_getchar() != 'g');
        putstr("ytof ", HOST);
        putstr(filename, HOST);
        putstr(" w \n", HOST);
        while (r_getchar() != ' ');
        r_printf("dumping ");
        putstr(buffer, HOST);
        putb('\4', HOST);
        putb('\4', HOST);
        r_printf("\7\7\7");
        return;
}


/*************************************************************************/
/*
/* Procedure: finish_segment(n)
/*
/* Description: this function completes segments at the end of a data run. Necessary
/* because the linear fitting function only terminates a segment based on the
/* data - it has no way of knowing that the user has stopped collecting data.
/*
/*************************************************************************/
```

```
void finish_segment(n)
int n;
{
        LINE_SEG *seg_ptr;

        if (segment_data[n].m00 > 10.0)
        {
        seg_ptr = end_segment(n);
        build_list(seg_ptr, n);
        }
}
```

APPENDIX D. ODOMETRY CORRECTION SOURCE CODE

```
/*
* file : nav.c
* purpose : All robot subroutines required for navigation
*
*/

#include "mml.h"

/* declaration of functions and return values */

extern void wait_point();
extern int wait_segment1();
extern int wait_segment();
extern void get_robot_speed();
extern void get_s_zero();
extern void get_initial_position();
extern void report_configuration();
extern CONFIGURATION get_sonar_config();
extern void correct_odometry_error();
extern void enable_display_status();
extern void displaystatus();

/*****************************************************************************
 FUNCTION: wait_point(pt)
 PARAMETERS: POINT pt
 PURPOSE: Busy wait until the the closest point of approach to
 parameter pt, then return.
 RETURNS: void
 CALLED BY: main
 CALLS: get_rob0;
 COMMENTS: 16 November 92 - Dave MacPherson
*****************************************************************************/
#define FLT_MAX 3.40282347e+38
void wait_point(pt)
POINT *pt;
{
double dist = FLT_MAX;
CONFIGURATION now;

get_rob0(&now);
while (dist > DIST(now.x, now.y, pt->x0, pt->y0))
{
dist = DIST(now.x, now.y, pt->x0, pt->y0);
get_rob0(&now);
}
```

```
}/* wait_point(pt) */

/
*****************************************************************************
*
 FUNCTION: wait_segment1()
 PARAMETERS: none
 PURPOSE: busy wait until the currentline segment being built
 is completed or the robot travels a distance greater
than the parameter length.
 RETURNS: integer value equal to the segment count
 CALLED BY: main
 CALLS: path_length();
 COMMENTS: 18 November 92 - Dave MacPherson
*****************************************************************************
*/
int wait_segment1()
{
int seg_count;
double current_pos;
double length = 100.0;

current_pos = path_length();
seg_count = segment_data_log[0].count;
r_printf("\12 seg_count => ");
r_printfi(seg_count);
while (1)
{
if ((path_length() - current_pos) > length)
{
seg_count = -1;
break;
}
if (segment_data_log[0].count > seg_count)
break;
}
return (seg_count);
}


/
*****************************************************************************
*
 FUNCTION: wait_segment()
 PARAMETERS:
 PURPOSE:
 RETURNS:
 CALLED BY:
 CALLS: NONE
 COMMENTS: 11 September 92 - Dave MacPherson
*****************************************************************************/
```

```
/*
int wait_segment()
 {
 int seg_count;

 seg_count = segment_data_log[0].count;
 r_printf("\12 seg_count => ");
 r_printfi(seg_count);
 while (segment_data_log[0].count == seg_count);
 return(seg_count);
 }
*/


/*************************************************************************
FUNCTION: get_robot_speed()
PARAMETERS: none
PURPOSE: sets the robot's speed for the entire mission
 based upon user input
RETURNS: void
CALLED BY: user()
CALLS: NONE
COMMENTS: 12 September 92 - Dave MacPherson
*************************************************************************/
void get_robot_speed()
{
double sp;

r_printf("\12 Enter desired robot speed. ");
sp = getreal(CONSOLE);
speed(sp);
}

/*************************************************************************
FUNCTION: get_s_zero()
PARAMETERS: none
PURPOSE: sets the robot's s_zero for the entire mission
 based upon user input
RETURNS: void
CALLED BY: user()
CALLS: NONE
COMMENTS: 12 September 92 - Dave MacPherson
*************************************************************************/
void get_s_zero()
{
double s_zero;

r_printf("\12 Enter desired s_zero ");
s_zero = getreal(CONSOLE);
size_const(s_zero);
}
```

```
/*************************************************************************
FUNCTION: get_initial_position()
PARAMETERS: none
PURPOSE: get the initial robot configuration
 based upon user input
RETURNS: void
CALLED BY: user()
CALLS: NONE
COMMENTS: 29 Oct 92 - Dave MacPherson
*************************************************************************/
void get_initial_position()
{
double x;
double y;
double t;
double k;
CONFIGURATION p;

r_printf("\12 Enter the starting x position: ");
x = getreal(CONSOLE);
r_printf("\12 Enter the starting y position: ");
y = getreal(CONSOLE);
r_printf("\12 Enter the starting orientation: ");
t = getreal(CONSOLE);
r_printf("\12 Enter the starting kappa: ");
k = getreal(CONSOLE);
set_rob(def_configuration(x, y, t, k, &p));
}

/*************************************************************************
FUNCTION: report_configuration()
PARAMETERS: none
PURPOSE: gets the current robot configuration
 and then displays it to the screen
RETURNS: void
CALLED BY: user()
CALLS: NONE
COMMENTS: 29 Oct 92 - Dave MacPherson
*************************************************************************/
void report_configuration()
{
r_printf("\12 Current Robot Config: x =>");
r_printfr(vehicle.x, 2);
r_printf(" y =>");
r_printfr(vehicle.y, 2);
r_printf(" theta =>");
r_printfr(r2d(vehicle.t), 2);
r_printf("\12");
}
```

```
/*************************************************************************
FUNCTION: report_path()
PARAMETERS: none
PURPOSE: gets the current robot path
 and then displays it to the screen
RETURNS: void
CALLED BY: user()
CALLS: NONE
COMMENTS: 17 May 93 - Dave MacPherson
*************************************************************************/
void report_path()
{
r_printf("\12 Robot current path: x =>");
r_printfr(current_robot_path.pc.x, 2);
r_printf(" y =>");
r_printfr(current_robot_path.pc.y, 2);
r_printf(" theta =>");
r_printfr(r2d(current_robot_path.pc.t), 2);
r_printf(" kappa =>");
r_printfr(current_robot_path.pc.k, 2);
r_printf("\12");
}


/*************************************************************************
FUNCTION: get_sonar_config()
PARAMETERS:
PURPOSE:
RETURNS:
CALLED BY:
CALLS: NONE
COMMENTS: 11 September 92 - Dave MacPherson
*************************************************************************/
CONFIGURATION get_sonar_config(seg_count)
int seg_count;
{
CONFIGURATION Qsonar;

Qsonar.x = segment_data_log[0].array[seg_count].tailx;
Qsonar.y = segment_data_log[0].array[seg_count].taily;
Qsonar.t = atan2(segment_data_log[0].array[seg_count].heady -
 segment_data_log[0].array[seg_count].taily,
 segment_data_log[0].array[seg_count].headx -
 segment_data_log[0].array[seg_count].tailx);
Qsonar.k = 0.0;
return Qsonar;
}


/*************************************************************************
FUNCTION: correct_odometry_error(Qsonar, Qmodel)
```

```
PARAMETERS:
PURPOSE:
RETURNS:
CALLED BY:
CALLS: NONE
COMMENTS: 11 September 92 - Dave MacPherson
*****************************************************************************/
void correct_odometry_error(Qsonar, Qmodel)
CONFIGURATION Qsonar, Qmodel;
{
CONFIGURATION Qodo_inv;
CONFIGURATION E, Qact, Qodo;
CONFIGURATION Qact_inv, X1, X1_inv;

get_rob0(&Qodo);
inverse(&Qodo, &Qodo_inv);
comp(&Qodo_inv, &Qsonar, &X1);
inverse(&X1, &X1_inv);
comp(&Qmodel, &X1_inv, &Qact);
set_rob(&Qact);

inverse(&Qact, &Qact_inv);
comp(&Qodo, &Qact_inv, &E);
wait_timer(100);
r_printf("\12 E => ");
r_printfr(E.x, 3);
r_printf(" ");
r_printfr(E.y, 3);
r_printf(" ");
r_printfr(E.t, 3);

wait_timer(100);
r_printf("\12 Qsonar => ");
r_printfr(Qsonar.x, 3);
r_printf(" ");
r_printfr(Qsonar.y, 3);
r_printf(" ");
r_printfr(Qsonar.t, 3);

wait_timer(100);
r_printf("\12 Qodo => ");
r_printfr(Qodo.x, 3);
r_printf(" ");
r_printfr(Qodo.y, 3);
r_printf(" ");
r_printfr(Qodo.t, 3);

wait_timer(100);
r_printf("\12 Qact => ");
r_printfr(Qact.x, 3);
r_printf(" ");
```

```
r_printfr(Qact.y, 3);
r_printf(" ");
r_printfr(Qact.t, 3);

}


/*****************************************************************************/
/* Procedure: displaystatus
/*
/* Description: called every 10 ms, this routine provides an update of
/* the current status to the lap-top as an aid in debugging
/* level 4 problems.
/*
/*****************************************************************************/

void displaystatus()
{

if (status != cur_display_status)
{
r_printf("\n\nCurrent status is ");
switch (status)
{
case SSTOP:
r_printf("SSTOP\n");
break;
case SLINE:
r_printf("SLINE\n");
break;
case SBLINE:
r_printf("SBLINE\n");
break;
case SFLINE:
r_printf("SFLINE\n");
break;
case SCONFIG:
r_printf("SCONFIG\n");
break;
case RMOVE:
r_printf("RMOVE\n");
break;
case SERROR:
r_printf("SERROR\n");
break;
default:
r_printf("UNKNOWN\n");
break;
}/* end switch */
}/* end if */
```

```
cur_display_status = status;
/*
 if (i_am_here != last_i_am_here)
 {
 r_printf("\nI_am_here => ");
 r_printfi(i_am_here);
 last_i_am_here = i_am_here;
 }
*/

}


/*****************************************************************************/
/* Procedure: enable_display_status()
/*
/* Description: Lowers interrupt mask to allow level 1 interrupts
/*
/*****************************************************************************/

void enable_display_status()
{

i_imaskdisplaystatus();

}
```

APPENDIX E. CARTOGRAPHY SOURCE CODE

```
**************************************************************************

        FILENAME: map_world.c
        PURPOSE: test file for simulating automated cartography
        CONTAINS:
        LAST UPDATE:10 July 93

**************************************************************************/
#include "mml.h"
#include "cartography.h"
#include "spatial.h"


/**************************************************************************
        FUNCTION: add_hole_to_world()
        PURPOSE: Adds a hole polygon to an existing
        Map World.
        RETURNS: void
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: The hole polygon can only be added after the
        boundary polygon has been added to the Map World.
**************************************************************************/
void add_hole_to_world(H, W)
Map_Polygon *H;
Map_World *W;
{
        Map_Polygon *current_polygon;
        int i;

        if (W->boundary == NULL)
        {
        r_printf("Error: the boundary polygon must be added first.");
        /* exit(0); */
        }
        if (W->hole_list == NULL)
        {
        W->hole_list = H;
        r_printf("\12The first hole was added to the partial map.");
        }
        else
        {
```

```
        if((w = (Map_World *)malloc(sizeof(Map_World))) == NULL) {
        /* fatal("create_world: malloc\n"); */
        /* exit(0); */
        }

        /* initialize fields */
        w->boundary = NULL;
        w->hole_list = NULL;
        w->degree = 0;

        r_printf("\n Created a new partial world.");
        return(w);
} /* create_map_world */

/*****************************************************************************
        FUNCTION: make_edge()
        PURPOSE: creates a new edge
        RETURNS: EDGE
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: This function builds a new edge.
*****************************************************************************/
EDGE *make_edge(x1, y1, x2, y2, type)
double x1, y1, x2, y2;
int type;
{
        EDGE *e1;

        if ((e1 = (EDGE *)malloc(sizeof(EDGE))) == NULL)
        {
        r_printf("Error make_edge: malloc.\n");
        /* exit(0); */
        }
        e1->v1.x = x1;
        e1->v1.y = y1;
        e1->v2.x = x2;
        e1->v2.y = y2;
        e1->type = type;

        return e1;
} /* end make edge */

/
/*****************************************************************************
        FUNCTION: complete()
        PURPOSE: Evaluates a partial world to see if it is
        complete.
        RETURNS: int 0 = FALSE, 1 = TRUE.
        CALLED BY: ANYBODY
        CALLS: poly_complete
        COMMENTS: Uses the poly_complete function to evaluate the
```

```
        completeness of each component polygon in the partial world.
**************************************************************************
*****/
int complete(w)
Map_World *w;
{
        Map_Polygon *current_polygon;
        int i;
        int count = 0;

        if(w->boundary == NULL && w->hole_list == NULL &&
        w->degree == 0)
        {
        r_printf("The world is not complete.\n");
        return 0;
        }
        else if(w->degree == 1 && poly_complete(w->boundary))
        {
        r_printf("The world is complete.\n");
        count = 1;
        }
        else if(w->degree > 1 && poly_complete(w->boundary))
        {
        current_polygon = w->hole_list;
        if (poly_complete(current_polygon))
        {
        count = 2;
        r_printf("The boundary of the world is complete.\n");
        for (i=2; i < w->degree; i++)
        {
        current_polygon = current_polygon->next;
        if (poly_complete(current_polygon))
        {
        r_printf("The hole is complete.\n");
        ++count;
        }
        }
        }
        }

        if (count == w->degree)
        return 1;
        else
        return 0;


} /* complete() */

/**************************************************************************
        FUNCTION: ploy_complete()
        PURPOSE: Evaluates a map polygon to see if it is
```

```
        complete.
        RETURNS: int 0 = FALSE, 1 = TRUE.
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS:
        }
        else
        {
        r_printf("Boundary polygon.\n");
        print_polygon(W->boundary);
        current_polygon = W->hole_list;
        r_printf("Hole Polygon\n");
        print_polygon(current_polygon);
        for (i=2; i < W->degree; i++)
        {
        current_polygon = current_polygon->next;
        r_printf("Hole Polygon\n");
        print_polygon(current_polygon);
        }
        }
} /* end print world */

/*************************************************************************
        FUNCTION: print_polygon()
        PURPOSE: Print all edges of a map polygon
        RETURNS: void
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: this function prints a polygon to the screen
*************************************************************************/
void print_polygon(p)
Map_Polygon *p;
{
        EDGE *current_edge;
        int i;

        current_edge = p->edge_list;
        for (i = 0; i < p->degree; i++)
        {
        r_printf("\nEdge =>");
         r_printfr(current_edge->v1.x, 3);
         r_printfr(current_edge->v1.y, 3);
        r_printfr(current_edge->v2.x, 3);
        r_printfr(current_edge->v2.y, 3);
        if (current_edge->type == REAL)
        r_printf(" REAL\n");
        else
        r_printf(" INFERRED\n");
        current_edge = current_edge->next;
        }
} /* end print polygon */
```

```
/*************************************************************************
        FUNCTION: plot_world()
        PURPOSE: Plots all edges of a map polygon
        RETURNS: void
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: Plots a partial world to the screen using gnuplot.
*************************************************************************/
void plot_world(w)
Map_World *w;
{
        EDGE *current_edge;
        Map_Polygon *current_polygon;
        int i, j;
        FILE *realedges, *inferrededges;
        char command[160];
        int count = 0;

/*
        realedges = fopen("real","w");
        inferrededges = fopen("inferred","w");
*/

        for (i=1; i <= w->degree; i++)
        {
        if (i == 1)
        current_polygon = w->boundary;
        else if (i == 2)
        current_polygon = w->hole_list;
        else if (i == 3)
        current_polygon = w->hole_list->next;

        current_edge = current_polygon->edge_list;
        for (j = 0; j < current_polygon->degree; j++)
        {
        if (current_edge->type == REAL)
        {
        /*
        fprintf(realedges, "%7.2f%7.2f\n%7.2f%7.2f\n\n",
        current_edge->v1.x, current_edge->v1.y,
        current_edge->v2.x, current_edge->v2.y);
        */
        }
        else if (current_edge->type == INFERRED)
        {
        /*
        fprintf(inferrededges, "%7.2f%7.2f\n%7.2f%7.2f\n\n",
        current_edge->v1.x, current_edge->v1.y,
        current_edge->v2.x, current_edge->v2.y);
        */
```

```
        }
        current_edge = current_edge->next;
        }
        /*
        fprintf(realedges, "\n");
        fprintf(inferrededges, "\n");
        */
        }

/*
        fclose(realedges);
        fclose(inferrededges);
        sprintf(command, "gnuplot %s", "world_plot.cmd");
        system(command);
*/
} /* end plot_world */


/*************************************************************************
        FUNCTION: plot_polygon()
        PURPOSE: Plots all edges of a map polygon
        RETURNS: void
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: Plots a polygon to the screen using gnuplot.
*************************************************************************/
void plot_polygon(p)
Map_Polygon *p;
{
        EDGE *current_edge;
        int i;
        FILE *realedges, *inferrededges;
        char command[160];

/*
        realedges = fopen("real","w");
        inferrededges = fopen("inferred","w");
*/

        current_edge = p->edge_list;
        for (i = 0; i < p->degree; i++)
        {
        if (current_edge->type == REAL)
        {
        /*
        fprintf(realedges, "%7.2f%7.2f\n%7.2f%7.2f\n",
        current_edge->v1.x, current_edge->v1.y,
        current_edge->v2.x, current_edge->v2.y);
        */
        }
        else if (current_edge->type == INFERRED)
```

```
        {
        /*
        fprintf(inferrededges, "%7.2f%7.2f\n%7.2f%7.2f\n",
        current_edge->v1.x, current_edge->v1.y,
        current_edge->v2.x, current_edge->v2.y);
        */
        }
        current_edge = current_edge->next;
        }

/*
        fclose(realedges);
        fclose(inferrededges);
        sprintf(command, "gnuplot %s", "polygon_plot.cmd");
        system(command);
        */
} /* end plot_polygon */


/*****************************************************************************
        FUNCTION: add_edge_to_polygon()
        PURPOSE: Adds a new edge to a map polygon
        RETURNS: Map Polygon *
        CALLED BY: ANYBODY
        CALLS:
        COMMENTS: this function allocates space for an edge and
 adds it to a polygon.
*****************************************************************************/
void add_edge_to_polygon(new_edge, p)
EDGE *new_edge;
Map_Polygon *p;
{
        EDGE *current_edge;
        int i;

        if (p->degree == 0)
        {
        p->edge_list = new_edge;
        }
        else
        {
        current_edge = p->edge_list;
        for (i = 1; i < p->degree; i++)
        {
        current_edge = current_edge->next;
        }
        current_edge->next = new_edge;
        new_edge->previous = current_edge;
        new_edge->next = p->edge_list; /* circularly linked list */
        p->edge_list->previous = new_edge;
        }
```

```
        ++p->degree;
} /* end add_edge_to_polygon */


/*************************************************************************
        FUNCTION: create_map_polygon()
        PURPOSE: create instance of a map polygon
        RETURNS: Map Polygon *
        CALLED BY: ANYBODY
        CALLS: fatal() <utilities.c>
        COMMENTS: this function allocates space for a map_polygon and
        returns a pointer to it.
*************************************************************************/
Map_Polygon *create_map_polygon()
{
        Map_Polygon *p;

        /* allocate memory for a polygon */
        if ((p = (Map_Polygon *) malloc(sizeof(Map_Polygon))) == NULL)
        {

        /*
         * fatal("create_polygon: malloc\n"); exit(FAILURE);
         */
        r_printf("malloc failed for create_map_polygon\n");
        /* exit(0); */
        }
        /* initialize fields */
        p->edge_list = NULL;
        p->previous = NULL;
        p->next = NULL;
        p->degree = 0;

        r_printf("\n Created a map polygon.\n");
        return (p);
}       /* end create_map_polygon() */

/*************************************************************************
        FUNCTION: next_scan_config()
        PURPOSE: determines the path to the closest inferred
        for the next translational scan.
        RETURNS: void
        CALLED BY: ANYBODY
        CALLS: fatal() <utilities.c>
        COMMENTS:
*************************************************************************/
void next_scan_config(w, C)
Map_World *w;
CONFIGURATION *C;
{
        void analyze_closest_edge();
```

```
Map_Polygon *current_polygon;
EDGE *current_edge;
int i, j;
double centerx, centery;
double edge_dist;
double closest_edge_dist = 1000.0;
EDGE *closest_edge;

r_printf("Entered the function next_scan_config.\n");
for (i = 0; i < w-> degree; i++)
{
if (i == 0)
current_polygon = w->boundary;
else if (i == 1)
current_polygon = w-> hole_list;
else
current_polygon = current_polygon->next;

current_edge = current_polygon->edge_list;
for (j = 0; j < current_polygon->degree; j++)
{
if (current_edge->type == INFERRED)
{
r_printf("\nEdge =>");
r_printfr(current_edge->v1.x, 3);
r_printfr(current_edge->v1.y, 3);
r_printfr(current_edge->v2.x, 3);
r_printfr(current_edge->v2.y, 3);
centerx = (current_edge->v1.x + current_edge->v2.x)/2.0;
centery = (current_edge->v1.y + current_edge->v2.y)/2.0;
/*
printf("\nEdge Center => %7.2f%7.2f",
centerx, centery);
*/
edge_dist = sqrt((centerx - C->x)*(centerx - C->x)
+ (centery - C->y)*(centery - C->y));

r_printf("\nedge_dist => ");
r_printfr(edge_dist, 3);

if (edge_dist < closest_edge_dist)
{
closest_edge = current_edge;
closest_edge_dist = edge_dist;
r_printf("\nclosest_edge_dist =>");
r_printfr(closest_edge_dist, 3);
}
}
current_edge = current_edge->next;
}
```

```
        } /* end for loop */
        analyze_closest_edge(closest_edge, C);
}

/*****************************************************************************
        FUNCTION: analyze_closest_edge()
        PURPOSE: determines the path to the closest inferred
        for the next translational scan.
        RETURNS: path list
        CALLED BY: next_scan_config
        CALLS: fatal() <utilities.c>
        COMMENTS:
*****************************************************************************/
void analyze_closest_edge(closest_edge, C)
EDGE *closest_edge;
CONFIGURATION *C;
{
        CONFIGURATION path1, path2;
        double centerx, centery;

/*
        printf("\nThe closest edge is => %7.2f%7.2f%7.2f%7.2f\n",
        closest_edge->v1.x, closest_edge->v1.y,
        closest_edge->v2.x, closest_edge->v2.y);
        */
        centerx = (closest_edge->v1.x + closest_edge->v2.x)/2.0;
        centery = (closest_edge->v1.y + closest_edge->v2.y)/2.0;

        /* the first backtrack path starts at the
        robot current configuration */
        path1.x = C->x;
        path1.y = C->y;
        path1.k = 0.0;

        /* the back track path ends at the center of the
        closest inferred segment */
        path2.x = centerx;
        path2.y = centery;
        path2.k = 0.0;

        if (centerx < C->x)
        {
        path1.t = 3.14;
        if (centery < C->y)
        path2.t = -1.57;
        else if (centery > C->y)
        path2.t = 1.57;
        }
        else
        {
        path1.t = 0.0;
```

```
        if (centery < C->y)
        path2.t = -1.57;
        else if (centery > C->y)
        path2.t = 1.57;
        }
        r_printf("\nThe first path element is => ");
        r_printfr(path1.x, 3);
        r_printfr(path1.y, 3);
        r_printfr(path1.t, 3);
        r_printfr(path1.k, 3);
        r_printf("\nThe second path element is => ");
        r_printfr(path2.x, 3);
        r_printfr(path2.y, 3);
        r_printfr(path2.t, 3);
        r_printfr(path2.k, 3);
} /* end analyze_closest_edge */
```

```
/*************************************************************************
FILENAME: mapper8.c
PURPOSE: The Global spatial learning algorithm
using the Free-Space-Model.
CONTAINS: Functions for automated cartography
AUTHOR: Dave MacPherson
DATE: 10 July 1993
*************************************************************************/

#include "mml.h"
#include "cartography.h"

extern LINE_SEG *get_current_segment();

user()
{
        CONFIGURATION C, first, second, third, fourth, fifth, sixth;
        Map_World *PW;
        Map_Polygon *B, *H1, *H2;

        void initialize();
        void find_orthogonal_orientation();
        void follow_hallway();
        void wall_follower();
        void cleanup();
        void translational_scanning();
        void integrate();
        void next_scan_config();
        void turn_right();
        void bline_turn_right();
        void turn_left();
        void turn_around();
        void turn_around1();
        void both_seg_correction();
        void translational_scanning1();

        /* Create a partial world */
        PW = create_map_world();

        /* Create a boundary polygon */
        B = create_map_polygon();

        /* Add the empty boundary polygon to the world */
        add_boundary_to_world(B, PW);


        def_configuration(0.0, 0.0, 0.0, 0.0, &C);
        initialize(&C);
```

```
        find_orthogonal_orientation(&first);
        while (! complete(PW))
        {
        translational_scanning1(C, PW);
        next_scan_config(PW, C);
        }

        cleanup(PW);
}       /* end user */


/*****************************************************************************
FUNCTION: translational_scanning1()
PARAMETERS: C, PW
PURPOSE: Executes a single translational scan for automated
cartography. Builds a boundary polygon from the segments gathered
by the robot.
RETURNS: void
CALLED BY: user
CALLS: report_configuration()
COMMENTS: 11 July 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void translational_scanning1(C, PW)
CONFIGURATION C;
Map_World *PW;
{
        EDGE *e1, *e2, *e3;
        LINE_SEG *right_side_seg;
        LINE_SEG *left_side_seg;
        int i;
        int done = 0;
        int count;

        line(&C);
        while (sonar(FRONTL) < 9.3 || sonar(FRONTL) > 100.0)
        {
        report_configuration();
        wait_timer(100);
        }

        for (i = 0; i < segment_data_log[0].count; i++)
        {
        e2 = make_edge(segment_data_log[0].array[i].headx,
        segment_data_log[0].array[i].heady,
        segment_data_log[0].array[i].tailx,
        segment_data_log[0].array[i].taily,
        REAL);
        add_edge_to_polygon(e2, PW->boundary);
        }
```

```
if (get_current_segment(7) != NULL)
        {
        e1 = PW->boundary->edge_list->previous;
        /* get the last edge added to the boundary polygon */

        right_side_seg = get_current_segment(7);
        e2 = make_edge(right_side_seg->headx, right_side_seg->heady,
        right_side_seg->tailx, right_side_seg->taily, REAL);
        if (e1->v2.x != e2->v1.x)
        {
        e3 = make_edge(e1->v2.x, e1->v2.y, e2->v1.x, e2->v1.y, INFERRED);
        add_edge_to_polygon(e3, PW->boundary);
        }

        add_edge_to_polygon(e2, PW->boundary);
        } /* end if */


        r_printf("\12 The degree of the boundary polygon is.");
        r_printfi(PW->boundary->degree);

} /* end translational_scanning */



/*************************************************************************
FUNCTION: turn_around()
PARAMETERS: none
PURPOSE: Rotates 180 degrees to turn the robot
around in a narrow hallway.
RETURNS: void
CALLED BY: user
CALLS: report_configuration()
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
void turn_around()
{
        r_printf("\12 Entered the turn around part.");
        stop0();
        wait_timer(30);
        rotate(PI);
        while(vehicle.t < 3.1);
        wait_timer(100);
        report_configuration();
        speed0(15.0);
} /* end turn_around() */



/*************************************************************************
```

```
FUNCTION: find_orthogonal_orientation
PARAMETERS: ps
PURPOSE: Rotates 360 degrees to obtain the best
surface for automated cartography
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 May 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void find_orthogonal_orientation(ps)
CONFIGURATION *ps;
{
        void circle_for_segments(); /* This function command the robot to search
        for edges to extract if none are detected during the rotation */

        int i;
        int seg_index;
        int seg_count;
        double seg_alpha;
        double dist;
        double seg_length;
        double seg_dist = 500.0;
        double seg_orientation;
        double headx, heady, tailx, taily;

        CONFIGURATION first;

        report_configuration();
        r_speed(0.3);
        rotate(2*DPI);

        while (vehicle.t < 2*DPI);
        seg_count = segment_data_log[SEG_FILE].count;
        r_printf("\12 Got segments, count= ");
        r_printfi(seg_count);
        if (seg_count == 0)
        circle_for_segments();

        /*
         * Loop through segments found, select alpha from segment that is
         * MIN_SEG_DIST to MAX_SEG_DIST cm away, and has the longest length.
         */


        for (i = 0; i < seg_count; i++)
        {
        dist = segment_data_log[SEG_FILE].array[i].r;
        seg_length = segment_data_log[SEG_FILE].array[i].length;

        /*
```

```
        * Check the constraints for this segment. If it is better
        * than the last one, then remember it with seg_index.
        */

        if (fabs(dist) < MAX_SEG_DIST && fabs(dist) > MIN_SEG_DIST &&
         seg_dist < fabs(dist) && seg_length > MIN_ROT_SEG)
        {
        seg_index = i;
        seg_dist = dist;
        }
        }/* end of for loop */

        /* Print out the segment that was chosen */
        r_printf("\12 The closest segment to use is: ");
        r_printf("\12 hx= ");
        r_printfr(segment_data_log[SEG_FILE].array[seg_index].headx, 3);
        r_printf(" hy = ");
        r_printfr(segment_data_log[SEG_FILE].array[seg_index].heady, 3);
        r_printf(" tx = ");
        r_printfr(segment_data_log[SEG_FILE].array[seg_index].tailx, 3);
        r_printf("\12 ty = ");
        r_printfr(segment_data_log[SEG_FILE].array[seg_index].taily, 3);
        r_printf(" length = ");
        r_printfr(segment_data_log[SEG_FILE].array[seg_index].length, 3);
        r_printf(" Phi = ");
        seg_alpha = (segment_data_log[SEG_FILE].array[seg_index].alpha);
        r_printfr(r2d(seg_alpha), 2);

        headx = segment_data_log[SEG_FILE].array[seg_index].headx;
        heady = segment_data_log[SEG_FILE].array[seg_index].heady;
        tailx = segment_data_log[SEG_FILE].array[seg_index].tailx;
        taily = segment_data_log[SEG_FILE].array[seg_index].taily;
        seg_orientation = atan2(taily - heady, tailx - headx);

        r_printf(" Seg orientation = ");
        r_printfr(r2d(seg_orientation), 2);
        if (seg_orientation < -HPI)
        seg_orientation = PI + seg_orientation;

        r_printf("\12 Rotation Amount = ");
        r_printfr(r2d(seg_orientation - norm(vehicle.t)), 2);

        rotate(seg_orientation - norm(vehicle.t));
        wait_timer(1000);
        /* rotate to a position parallel to the closest segment */

        def_configuration(0.0, 0.0, 0.0, 0.0, &first);
        set_rob0(&first);
        (*ps) = first;
}       /* end find orthogonal orientation */
```

```
/*************************************************************************
FUNCTION: wall_follower
PARAMETERS: path
PURPOSE: follows the right hand wall in a hallway
for automated cartography
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 29 June 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
void circle_for_segments()
{
        CONFIGURATION circle;

        r_printf("\12 No segments detected during rotation.");
        r_printf("\12 Need to circle for segments.");
        def_configuration(vehicle.x, vehicle.y, vehicle.t, 0.01, &circle);
        line(&circle);
} /* end circle_for_segments */


/*************************************************************************
FUNCTION: wall_follower
PARAMETERS: path
PURPOSE: follows the right hand wall in a hallway
for automated cartography
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 29 June 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
void wall_follower(path)
CONFIGURATION path;
{
        LINE_SEG *right_side_seg;
        LINE_SEG *left_side_seg;
        double right_theta;
        double left_theta;
        double theta;
        CONFIGURATION Qodo, Qact;
        CONFIGURATION second, third;
        CONFIGURATION current;
        double right_seg_range;
        double left_seg_range;
        double obstacle_range;
        double new_x, new_y, new_t;
        int count = 0;
```

```
start_segment(RIGHTF);
start_segment(LEFTF);
line(&path);
r_printf("\12 Entering configuration is x => ");
r_printfr(path.x, 3);
r_printf(" y => ");
r_printfr(path.y, 3);
r_printf(" theta => ");
r_printfr(path.t, 3);
r_printf(" kappa => ");
r_printfr(path.k, 3);

/* correct robot path based the right hand wall */
while(count < 4) /* stop after 4 turns */
{
if (sonar(FRONTL) < 120.0 && sonar(FRONTL) > 9.3)
{
r_printf("\12 Entered the left turn part.");
obstacle_range = sonar(FRONTL);

def_configuration(vehicle.x + (obstacle_range - 50.0) * cos(vehicle.t),
vehicle.y + (obstacle_range - 50.0) * sin(vehicle.t),
path.t + HPI,
0.0, &path);
line(&path);
start_segment(RIGHTF);
while (fabs(path.t - vehicle.t) > 0.01);
++count;
}
else if (get_current_segment(RIGHTF) != NULL
&& get_current_segment(RIGHTF)->length > MIN_WALL_SEG
&& sonar(RIGHTF) > 9.3)
{
right_side_seg = get_current_segment(RIGHTF);
right_seg_range = sonar(RIGHTF);

right_side_seg = get_current_segment(RIGHTF);
r_printf("\12 Right side line segment length =");
r_printfr(right_side_seg->length, 2);
right_seg_range = sonar(RIGHTF);
r_printf("\12 Right side line segment range =");
r_printfr(right_seg_range, 2);

right_theta = atan2(right_side_seg->taily -
right_side_seg->heady,
right_side_seg->tailx - right_side_seg->headx);
r_printf("\12 Right side line segment orientation =");
r_printfr(right_theta, 2);

get_rob0(&Qodo);
if(fabs(norm(path.t) - 0.0) < 0.1)
```

```
{
new_x = Qodo.x;
new_y = Qodo.y + (right_seg_range - WALL_DISTANCE);
new_t = Qodo.t - right_theta;
r_printf("\12 theta = 0 Correction.");
}
else if(fabs(norm(path.t) - HPI) < 0.1)
{
new_x = Qodo.x - (right_seg_range - WALL_DISTANCE);
new_y = Qodo.y;
new_t = Qodo.t - norm(right_theta - Qodo.t);
r_printf("\12 theta = HPI Correction.");
}
else if(fabs(norm(path.t) + HPI) < 0.1)
{
new_x = Qodo.x + (right_seg_range - WALL_DISTANCE);
new_y = Qodo.y;
new_t = path.t - norm(right_theta - path.t);
r_printf("\12 theta = minus HPI Correction.");
}
else if(fabs(norm(path.t) + PI) < 0.1 ||
fabs(norm(path.t) - PI) < 0.1)
{
new_x = Qodo.x;
new_y = Qodo.y - (right_seg_range - WALL_DISTANCE);
if (right_theta < 0.0)
new_t = -right_theta;
else
new_t = Qodo.t;
r_printf("\12 theta = PI Correction.");
}

def_configuration(new_x, new_y, new_t, 0.0, &Qact);

set_rob0(&Qact);
r_printf("\12 Right Wall Correction => x = ");
r_printfr(new_x, 3);
r_printf(" y = ");
r_printfr(new_y, 3);
r_printf(" t = ");
r_printfr(new_t, 3);
if(fabs(norm(path.t) - 0.0) < 0.1 ||
fabs(norm(path.t) + PI) < 0.1 || fabs(norm(path.t) - PI) < 0.1)
{
while(fabs(path.y - vehicle.y) > 0.5);
wait_timer(100);
}
else
wait_timer(WAIT);
report_configuration();
} /* end else if */
```

```
        else if (sonar(RIGHTF) < 0.0)
        {
        start_segment(RIGHTF);
        finish_segment(RIGHTF);
        }
        report_configuration();
        wait_timer(100);
        } /* end while */
} /* wall_follower */



/*************************************************************************
FUNCTION: follow_hallway
PARAMETERS: ps
PURPOSE: follows the Sp fifth floor hallway
for automated cartography
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 May 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
void follow_hallway(ps)
CONFIGURATION *ps;
{
        LINE_SEG *right_side_seg;
        LINE_SEG *left_side_seg;
        double right_theta;
        double left_theta;
        double theta;
        CONFIGURATION second, third;
        double right_seg_range;
        double left_seg_range;
/*      void both_seg_correction(); */

        start_segment(RIGHTF);
        start_segment(LEFTF);
        line(&ps);
        r_printf("\12 Entering configuration is x => ");
        r_printfr(ps->x, 3);
        r_printf(" y => ");
        r_printfr(ps->y, 3);
        r_printf(" theta => ");
        r_printfr(ps->t, 3);
        r_printf(" kappa => ");
        r_printfr(ps->k, 3);

        /* correct robot path once based upon hallway walls */
/*
        while((sonar(FRONTL) < 9.3 || sonar(FRONTL) > 100.0) &&
```

```
        (sonar(FRONTR) < 9.3 || sonar(FRONTR) > 100.0))
*/
        while(sonar(FRONTL) < 9.3 || sonar(FRONTL) > 150.0)
        {
        if (get_current_segment(RIGHTF) != NULL
        && get_current_segment(LEFTF) != NULL)
        {
        right_side_seg = get_current_segment(RIGHTF);
        right_seg_range = sonar(RIGHTF);
        left_side_seg = get_current_segment(LEFTF);
        left_seg_range = sonar(LEFTF);

        if (sonar(RIGHTF) > 9.3 && sonar(LEFTF) > 9.3 &&
        right_side_seg->length > MIN_WALL_SEG &&
        left_side_seg->length > MIN_WALL_SEG)
        {
        both_seg_correction(right_side_seg, right_seg_range,
        left_side_seg, left_seg_range);
        } /* end if */
        else if (get_current_segment(RIGHTF) != NULL &&
        right_side_seg->length > MIN_WALL_SEG)
        {
        right_side_seg = get_current_segment(RIGHTF);
        r_printf("\12 Right side line segment length =");
        r_printfr(right_side_seg->length, 2);
        right_seg_range = sonar(RIGHTF);
        r_printf("\12 Right side line segment range =");
        r_printfr(right_seg_range, 2);

        if (sonar(RIGHTF) > 9.3)
        {
        right_theta = atan2(right_side_seg->taily -
        right_side_seg->heady,
        right_side_seg->tailx - right_side_seg->headx);
        r_printf("\12 Right side line segment orientation =");
        r_printfr(right_theta, 2);

        get_rob0(&second);
        def_configuration(second.x,
        right_seg_range - 100.0,
        -right_theta, 0.0 , &third);

        set_rob0(&third);
        r_printf("\12 Right Wall Correction => ");
        r_printfr(right_theta, 3);
        wait_timer(WAIT);
        report_configuration();
        } /* end inner if */
        else
        {
        start_segment(RIGHTF);
```

```
        }
        } /* end else if */
/*
        else if (get_current_segment(LEFTF) != NULL &&
        left_side_seg->length > MIN_WALL_SEG)
        {
        left_side_seg = get_current_segment(LEFTF);
        r_printf("\12 Left side line segment length =");
        r_printfr(left_side_seg->length, 2);
        left_seg_range = sonar(LEFT);
        r_printf("\12 Left side line segment range =");
        r_printfr(left_seg_range, 2);

        if (sonar(LEFTF) > 9.3)
        {
        get_rob0(&p2);
        left_theta = atan2(left_side_seg->taily - left_side_seg->heady,
        left_side_seg->tailx - left_side_seg->headx);
        r_printf("\12 Left side line segment orientation =");
        r_printfr(left_theta, 2);

        def_configuration(p2.x,
        223.0 - left_seg_range,
        -left_theta, 0.0 , &p2);

        set_rob0(&p2);
        r_printf("\12 Left Wall Correction => ");
        r_printfr(left_theta, 3);
        wait_timer(WAIT);
        report_configuration();
        }
        else
        {
        start_segment(LEFTF);
        }
        }
*/
        }
        report_configuration();
        wait_timer(100);
        } /* end while */
        r_printf("\12 Forward Looking Sonar. =");
        r_printfr(sonar(FRONTL), 2);
} /* end follow_hallway */


/*************************************************************************
FUNCTION: both_seg_correction();
PARAMETERS: right_side_seg, right_seg_range
left_side_seg, left_seg_range
```

```
PURPOSE: corrects robot configuration to align to
the center of the hallway
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 May 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void both_seg_correction(right_side_seg, right_seg_range,
        left_side_seg, left_seg_range)
LINE_SEG *right_side_seg;
double right_seg_range;
LINE_SEG *left_side_seg;
double left_seg_range;
{
        double right_theta;
        double left_theta;
        double theta;
        CONFIGURATION p2;

        r_printf("\12 Use both segments for Correction.");
        right_theta = atan2(right_side_seg->taily -
        right_side_seg->heady,
        right_side_seg->tailx - right_side_seg->headx);
        r_printf("\12 Right side line segment orientation =");
        r_printfr(right_theta, 2);

        left_theta = atan2(left_side_seg->taily -
        left_side_seg->heady,
        left_side_seg->tailx - left_side_seg->headx);
        r_printf("\12 Left side line segment orientation =");
        r_printfr(left_theta, 2);

        theta = (right_theta + left_theta) / 2.0;
        get_rob0(&p2);

        def_configuration(p2.x,
        right_seg_range - 100.0,
        vehicle.t - theta, 0.0 , &p2);

        set_rob0(&p2);
        r_printf("\12 Both Wall Correction => ");
        r_printfr(theta, 3);
        while(fabs(vehicle.y) > 1.0);
/*      wait_timer(WAIT); */
        report_configuration();
} /* both_seg_correction() */


/*****************************************************************************
FUNCTION: translational_scanning
```

```
PARAMETERS: ps
PURPOSE: Scans the straight line wall segment
surface for automated cartography
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 May 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void translational_scanning(ps)
CONFIGURATION ps;
{
        LINE_SEG *right_side_seg;
        LINE_SEG *left_side_seg;
        CONFIGURATION p2;
        int n = 7;
        int m = 4;


        line(&ps);
        while (sonar(0) > 100.0 || sonar(0) < MIN_SONAR_RANGE)
        {
        /* code to steer robot down the hallway */
        wait_timer(500);
        r_printf("\12 Use right side line segment for steering");
        right_side_seg = get_current_segment(7);
        r_printf("\12 Right side line segment orientation =");
        r_printfr(r2d(right_side_seg->alpha + HPI), 2);
        r_printf("\12 Right side line segment range =");
        r_printfr(sonar(7), 2);
        report_configuration();
        if (fabs(right_side_seg->alpha + HPI - ps.t) < 0.2)
        {
        get_rob0(&p2);
        def_configuration(p2.x, p2.y, right_side_seg->alpha + HPI, 0.0, &p2);
        skip();
        line(&p2);
        r_printf("\12 Applying a correction using right wall.");
        wait_timer(500);
        }
/*
 r_printf("\12 Left side line segment orientation =");
 left_side_seg = get_current_segment(4);
 r_printfr(r2d(left_side_seg->alpha), 2);
*/
        }


        r_printf("\12Detected obstactle less than 100 cm ahead");
        report_configuration();
```

```
        if (sonar(m) < MIN_SONAR_RANGE && sonar(n) < MIN_SONAR_RANGE)
        {
        r_printf("\12 Under range on both side sensors");
        turn_right();
        } else if (sonar(m) < MIN_SONAR_RANGE && sonar(n) > MIN_SONAR_-
RANGE)
        turn_left();
        else if (sonar(m) > MIN_SONAR_RANGE && sonar(n) > MIN_SONAR_-
RANGE)
        {
        if (sonar(m) > sonar(n))
        turn_left();
        else
        turn_right();
        }
        if (sonar(n) < MIN_SONAR_RANGE && sonar(m) > MIN_SONAR_RANGE)
        turn_right();
}       /* end scan */


/*************************************************************************
FUNCTION: turn_right()
PARAMETERS: none
PURPOSE: Turns the robot right 50.0 cm from its current
configuration.
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*************************************************************************/
void turn_right()
{
        CONFIGURATION second;

        r_printf("\12 Entered the turn right function");
        get_rob0(&second);
        def_configuration(second.x + 50.0 * cos(second.t),
         second.y + 50.0 * sin(second.t),
         second.t - HPI, 0.0, &second);
        line(&second);
        while (vehicle.t > second.t);
}       /* end turn_right() */


/*************************************************************************
FUNCTION: bline_turn_right()
PARAMETERS: none
PURPOSE: Turns the robot right using a bline function.
RETURNS: void
CALLED BY: user
```

```
CALLS: none
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void bline_turn_right()
{
        CONFIGURATION second;

        r_printf("\12 Entered the bline turn right function");
        get_rob0(&second);
        def_configuration(second.x + 50.0 * cos(second.t),
         second.y + 50.0 * sin(second.t) - 75.0,
         second.t - HPI, 0.0, &second);
        bline(&second);
        while (vehicle.t > second.t);
}       /* end bline_turn_right() */


/*****************************************************************************
FUNCTION: turn_left()
PARAMETERS: none
PURPOSE: Turns the robot left 50.0 cm from its current
configuration.
RETURNS: void
CALLED BY: user
CALLS: none
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void turn_left()
{
        CONFIGURATION second;

        r_printf("\12 Entered the turn left function");
        get_rob0(&second);
        def_configuration(second.x + 50.0 * cos(second.t),
         second.y + 50.0 * sin(second.t),
         second.t + HPI, 0.0, &second);
        line(&second);
        while (vehicle.t < second.t);
}       /* end turn_left() */



/*****************************************************************************
FUNCTION: initialize()
PARAMETERS: CONFIGURATION ps
PURPOSE: Starts the location trace function,
enables all appropriate sonars,
gets the robot's initail speed from the user
sets up all sonar logging and linear fitting functions.
```

```
RETURNS: void
CALLED BY: user
CALLS:
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void initialize(first)
CONFIGURATION *first;
{
        double s = 10.0;

        buffer_loc = index_loc = malloc(300000);
        bufloc = indxloc = (double *) malloc(60000);
        loc_tron(2, 0x3f, 30);
        enable_sonar(FRONTL);
        enable_sonar(BACKL);
        enable_sonar(BACKR);
        enable_sonar(FRONTR);

/*      get_robot_speed(); */
        speed(15.0);
        size_const(s);
/*      get_initial_position(); */
        set_rob(&first);
        r_printf("\12 In the initialize() function.");
        report_configuration();
        enable_sonar(LEFTF);
        set_log_interval(LEFTF, 1);
        enable_sonar(RIGHTF);
        set_log_interval(RIGHTF, 1);
        enable_linear_fitting(LEFTF);
        enable_linear_fitting(RIGHTF);
        enable_data_logging(LEFTF, 2, SEG_FILE);
        enable_data_logging(RIGHTF, 2, 1);

} /* end initialize() */


/*****************************************************************************
FUNCTION: cleanup()
PARAMETERS: none
PURPOSE: Disables all sonars
Ends all segments
Disables data logging
Ends the location trace function
Turns off the robot wheel motors
Tranfers line segment data back to the host computer
Transfers robot trajectory data back to the host computer
RETURNS: void
CALLED BY: user
CALLS:
```

```
COMMENTS: 27 June 93 - Dave MacPherson
TASK: Level 0
*****************************************************************************/
void cleanup(PW)
Map_World *PW;
{

        r_printf("\12 Performing the cleanup function");
        disable_sonar(LEFTF);
        disable_sonar(RIGHTF);
        finish_segment(LEFTF);
        finish_segment(RIGHTF);
        disable_linear_fitting(LEFTF);
        disable_linear_fitting(RIGHTF);
        disable_data_logging(LEFTF, 2);
        disable_data_logging(RIGHTF, 2);
        loc_troff();

        motor_on = NO;
        xfer_world_to_host(PW, "world.12July93");
        xfer_segment_to_host(1, "segment7.12July93");
        xfer_segment_to_host(0, "segment4.12July93");
        loc_trdump("loc_dump.12July93");
} /* cleanup() */
```

LIST OF REFERENCES

[AAAI 92]"AAAI 1992 Fall Symposium Series Reports," *AI Magazine*, v. 14(1), p. 10, 1993.

[AAAI 92a] "Robot Competition and Exhibition Entries," *AI Review*, pp. A32-A48, July, 1992.

[Abresch 92]Abresch, R., *Path Tracking Using Simple Planar Curves*, Master's Thesis, Naval Postgraduate School, Monterey, California, March 1992.

[Airey 90]Airey, J., Rohlf, J., and Brooks, F., "Towards Image Realism With Interactive Update Rates in Complex Virtual Building Environments," *Computer Graphics*, v. 24(2), pp. 68-87, March, 1990.

[Alexander 93]Alexander, J., *Motion Control and Obstacle Avoidance for Automobiles Vehicles Using Simple Planar Curves*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1993.

[Arkin 89] Arkin, R., "Navigation Path Planning for a Vision-Based Mobile Robot," *Robotica*, v. 7, pp. 49-63, 1989.

[Arkin 90]Arkin, R., and Murphy, R., "Autonomous Navigation in a Manufacturing Environment," *IEEE Transaction on Robotics and Automation*, v. 6(4), pp. 445-454, August, 1990.

[Arkin 93]Arkin, R., and Grupen, R., "Behavior-Based Reactive Robotic Systems," *IEEE Tutorial T4*, pp. 1-34, May, 1993.

[Asada 90] Asada, M., "Map Building for a Mobile Robot from Sensory Data," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 312-322, May, 1990.

[Bares 89] Bares, J., et al., "Ambler: An Autonomous Rover for Planetary Exploration", *Computer*, v 22(6), pp. 18-26, June, 1989.

[Barshan 90]Barshan, B., and Kuc, R., "Differentiating Sonar Reflections from Corners and Planes by Employing Intelligent Sonar," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, v. 12(6), pp. 560-569, June, 1990.

[Basye 92] Basye, K., "An Automata-Based Approach to Robotic Map Learning," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 1-4, 1992.

[Beckerman 90]Beckerman, M., "Treatment of Systematic Errors in the Processing of Wide-Angle Sonar Sensor Data for Robotic Navigation," *IEEE Transaction on Robotics and Automation*, v. 6(2), pp. 137-145, April, 1990.

[Bloch 87] Bloch, N., *Abstract Algebra with Applications*, pp. 1-9, Prentice Hall, Inc., 1987.

[Borenstein 90]Borenstein, J., and Koren, Y., "Real-Time Obstacle Avoidance for Fast Mobile Robots in Cluttered Environments," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 572-577, 1990.

[Borenstein 91]Borenstein, J., and Koren, Y., "The Vector Field Histogram: Fast Obstacle Avoidance for Mobile Robots," *IEEE Journal of Robotics and Automation*, v. 7(3), pp. 278-288, 1991.

[Borenstein 92]Borenstein, J., and Koren, Y., "Noise Rejection for Ultrasonic Sensors in Mobile Robot Applications," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1727-1732, May, 1992.

[Bowditch 84]Bowditch, N., *American Practical Navigator*, pp. 1-58, Defense Mapping Agency, 1984.

[Brogan 85]Brogan, W. L., *Modern Control Theory*, pp. 23-25, 2d ed., Prentice-Hall, 1985.

[Brooks 83]Brooks, R., "Solving the Find-Path Problem by Good Representation of Free Space," *IEEE Trans. on Systems, Man, and Cybernetics*, v. SMC-13(3), 1983.

[Brooks 86] Brooks, R., "A Layered Intelligent Control System for a Mobile Robot," *Robotics Research*, pp. 365-372, Edited by Fangeras and Girald, MIT Press, 1986.

[Brooks 86a]Brooks, R., "A Robust Layered Control System for a Mobile Robot," *IEEE Transaction on Robotics and Automation*, pp. 14-21, March, 1986.

[Brooks 89]Brooks, R., "A Robot that Walks; Emergent Behavior from Carefully Evolved Networks," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 692-696, 1989.

[Brooks 89a] Brooks, R., "Robot Beings," *Proceedings of the IEEE International Workshop on Intelligent Robots and Systems*, pp. 2-12, 1989.

[Brooks 90]Brooks, R., "Elephants Don't Play Chess," *Robotics and Autonomous Systems*, v. 6, pp. 3-15, 1990.

[Brooks 90a] Brooks, R., Massachusetts Institute of Technology Artificial Intelligence Laboratory, A. I. Memo 1227, "The Behavior Language; User's Guide," pp. 1-13, April, 1990.

[Brooks 91] Brooks, R., "Intelligence Without Representation," *Artificial Intelligence*, v. 47(1-3), pp. 139-159, January, 1991.

[Brooks 93]Lecture given by Rodney Brooks at the NATO-ASI Workshop on the Biology and Technology of Intelligent Autonomous Agents, Levico Terme, Italy, 1-12 March, 1993.

[Brown 85]Brown, R., et al., "Map-Making and Localization for Mobile Robots Using Shape Metrics," *IEEE Journal on Robotics and Automation*, v. RA-1(4), pp.

191-205, December, 1985.

[Brown 92]Brown, M., "Feature Extraction Techniques for Recognizing Solid Objects with Ultrasonic Range Sensors," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 10-17, 1992.

[Brutzman 92]Brutzman, D., *NPS AUV INTEGRATED SIMULATOR*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1992.

[Brutzman 92a] Brutzman, D., Compton, M., and Kanayama, Y., "Autonomous Sonar Classification using Expert Systems," *Proceedings of the IEEE Oceanic Engineering Society Conference*, p. 5, October, 1992.

[Busnel 79]Busnel, R., and Fish, J., *International Interdisciplinary Symposium on Animal Sonar Systems*, pp. 355-380, 2nd Ed., Plenum Press, 1979.

[Byrnes 93]Byrnes, R., *The Rational Behavior Model: A Multi-Paradigm, Tri-Level Software Architecture for Control of Autonomous Vehicles*, Doctoral Dissertation, Naval Postgraduate School, Monterey, California, March, 1993.

[Canny 88]Canny, J., and Donald, B., "Simplified Voronoi Diagram," *Discrete and Computational Geometry*, v. 3(3), pp. 219-236, 1988.

[Carlin 60]Carlin, B., *Ultrasonics*, 2nd Ed., pp. 1-37, McGraw-Hill, Inc., 1960.

[Chatial 82]Chatial, R., "Path Planning and Environmental Learning in a Mobile Robot System," *Proc. ECAI*, pp. 3-5, August, 1982.

[Compton 92]Compton, M., *Minefield Search and Object Recognition for Autonomous Underwater Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, California, March, 1992.

[Congdon 93] Congdon, C., et. al., "CARMEL Versus FLAKEY - A Comparison of Two Winners," *AI Magazine*, v. 14(1), pp. 49-56, 1993.

[Connell 92]Connell, J., "SSS: A Hybrid Architecture Applied to Robot Navigation," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2719-2724, 1992.

[Connell 92a]Connell, J., "Extending the Navigation Metaphor to Other Domains," AAAI *Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 29-35, 1992.

[Cox 89] Cox, I., "Blanche: Position Estimation for an Autonomous Mobile Robot," Proceedings *of the IEEE/RSJ International Workshop on Robots and Systems (IROS '89)*, pp. 432-439, 1989.

[Cox 90] Cox, I., and Wilfong, G., *Autonomous Mobile Robots*, Springer-Verlag, pp. 191-197, 1990.

[Cox 91] Cox, I., "Blanche - An Experiment in Guidance and Navigation of an Autonomous Robot Vehicle," *IEEE Transaction on Robotics and Automation*, v. 7(2),

pp. 193-204, April, 1991.

[Cracknell 80]Cracknell, A., *Ultrasonics*, pp. 11-37, Wykeham, Publications, LTD, 1980.

[Crowley 85]Crowley, J., "Dynamic World Modeling for an Intelligent Mobile Robot Using a Rotating Ultra-Sonic Ranging Device," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 128-135, March, 1985.

[Crowley 85a] Crowley, J., "Navigation for an Intelligent Mobile Robot," *IEEE Journal of Robotics and Automation*, v. RA-1(1), pp. 31-41, March, 1985.

[Crowley 86] Crowley, J., "Representation and Maintenance of a Composite Surface Model," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1455-1461, 1986.

[Crowley 89]Crowley, J., "World Modeling and Position Estimation using Ultrasonic Ranging," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 674-680, 14-19 May, 1989.

[Curran 93] Curran, A., and Kyriakopoulos, K., "Sensor-Based Self-Localization for Wheeled Mobile Robots," *Proceedings of the IEEE Conference on Robotics and Automation*, v. 1, pp. 8-13, 1993.

[Daily 88] Daily, M., et al., "Autonomous Cross-Country Navigation With The ALV," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 718-726, 1988.

[Dario 86] Dario, P., *Sensors and Sensory Systems for Advanced Robots*, pp. 128-129, Springer-Verlag, 1986.

[Davis 93] Davis, R, Shrobe, H., and Szolovits, P., "What is Knowledge Representation?," *AI Magazine*, v. 14(1), pp. 17-30, 1993.

[Dean 93] Dean, T., and Bonasso, R., "1992 AAAI Robot Exhibition and Competition," *AI Magazine*, v. 14(1), pp. 35-48, 1993.

[Dolezal 87]Dolezal, M., *A Simulation Study of a Speed Control System for Autonomous On-Road Operation of Autonomous Vehicles*, Master's Thesis, Naval Postgraduate School, Monterey, California, June, 1987.

[Drumheller 87]Drumheller, M., "Mobile Robot Localization Using Sonar," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, v. PAMI-9(2), pp. 325-332, March, 1987.

[Dudek 91] Dudek, G., Jenkin, M., Milios, E., and Wilkes, D., "Robotic Exploration as Graph Construction," *IEEE Transaction on Robotics and Automation*, v. 7(6), pp. 859-869, December, 1991.

[Dutton 78]Maloney, E., *Dutton's Navigation and Piloting*, pp. 1-6, Naval Institute Press, 1978.

[Elfes 87] Elfes, A., "Sonar-Based Real-World Mapping and Navigation," *IEEE Journal*

of Robotics and Automation, v. RA-3, no. 3, pp. 149-165, 1987.

[Elfes 90] Elfes, A., "Occupancy Grids: A Stochastic Spatial Representation for Active Robot Perception," *Proceedings of the Sixth Conf. on Uncertainty in AI*, July, 1990.

[Engelson 92]Engelson, S., and McDermott, D., "Error Correction in Mobile Robot Map Learning," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2555, May, 1992.

[Engelson 92a]Engelson, S. and McDermott, D., "Maps Considered As Adaptive Planning Resource," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 36-44, 1992.

[Engelmore 88]Engelmore, R., and Morgan, T., *Blackboard Systems*, pp. 159-188, Addison Welsey, 1988.

[Everett 82]Everett, H., *A Microprocessor Controlled Autonomous Sentry Robot*, Master's Thesis, Naval Postgraduate School, Monterey, California, October, 1982.

[Everett 89]Naval Ocean Systems Command, Technical Document 1450, *RobartII - A Robotic Security Testbed*, by H. Everett, and G. Gilbreath, p. 17, January 1989.

[Everett 90]Naval Ocean Systems Command, Technical Document 1835, *Modeling the Environment of a Mobile Security Robot*, by H. Everett, G. Gilbreath, and J. Nieusma, p. 3, June 1990.

[Everett 92]Naval Command, Control and Ocean Surveillance Center, Technical Note 1194, Update 1, *Survey of Collision Avoidance and Ranging Sensors for Mobile Robots*, by Everett, H., and Stitz, E., pp. 33-56, December, 1992.

[Everett 93] Naval Ocean Systems Command, Technical Note 1710, Rev. 1, *Multiple Robot Host Architecture*, H. Everett, R. Laird, G. Gilbreath, and T. Heath-Pastore, p. 3, June 1993.

[Fish 93] Fish, R., *An Expert System for High Level Motion Control for an Autonomous Mobile Robot*, Master's Thesis, Naval Postgraduate School, Monterey, California, June, 1993.

[Fryxell 88]Fryxell, R., "Navigation Planning Using Quadtrees," *Mobile Robots II*, v. 852, pp. 256-261, Wolfe, W., Chun, W., Eds., *Proc. SPIE*, 1988.

[Fu 87] Fu, K., Gonzalez, R., and Lee, C., *Robotics Control, Sensing, Vision, and Intelligence*, pp. 12-76, McGraw-Hill, 1987.

[Gat 91] Gat, E., *Reliable Goal-Directed Reactive Control of Autonomous Mobile Robots*, Doctoral Dissertation, Virginia Polytechnic Institute, Blacksburg, VA., pp. 2-23, April, 1991.

[Gat 91a] Gat, E., "Robust Low-Computation Sensor-Driven Control for Task-Directed Navigation," *Proceedings of the IEEE Conference on Robotics and Automa-*

tion, pp. 2484-2489, April, 1991.

[Gevarter 85]Gevarter, W. B., *Intelligent Machines: An Introductory Perspective of Artificial Intelligence and Robotics*, pp. 23-45, Prentice-Hall, 1985.

[Giralt 79]Giralt, G., Sobek, R., and Chatila, R., "A Multi-Level Planning and Navigation System for a Mobile Robot: A First Approach to HILARE," *Proc. 6th Int Conf Artificial Intelligence*, pp. 335-337, 1979.

[Gould 88]Gould, R., *Graph Theory*, p.3, Benjamin/Cummings Inc., 1988.

[Hamming 83]Hamming, R., *Digital Filters*, 2nd. Ed., pp. 6-9, Prentice-Hall, 1983.

[Hinkel 88]Hinkel, R., Knierieman, T., and von Puttkamer, E., "Mobot-III: an Autonomous Mobile Robot for Indoor Applications," *International Symposium and Exhibition on Robots*, pp. 233-239, 1990.

[Holenstein 92]Holenstein, A., Muller, M., and Badreddin, E., "Mobile Robot Localization in Structured Environment Cluttered with Obstacles," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2576, May, 1992.

[Hubert 88] Hubert, M., Kanade, T., and Kweon, I., "3-D Vision Techniques for Autonomous Vehicles," *NSF Range Image Understanding Workshop*, pp. 273-337, 1988.

[Hubert 92] Hubert, M., et al., "Plan Recognition for Real-World Autonomous Robots: Work in Progress," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 68-73, 1992.

[Hutchinson 90]Hutchinson, S. A. and Kak, A. C., "Spar: A Planner That Satisfies Operational and Geometric Goals in Uncertain Environments," *AI Magazine*, v. 11(1), pp. 30-61, Spring 1990,

[Hwang 92] Hwang, Y. and Ahuja, N., "Gross Motion Planning - A Survey," *ACM Computing Surveys*, pp. 219-292, September, 1992.

[Ingold 92]Ingold, B., *Key Feature Identification from Image Profile Segments Using a High Frequency Sonar*, Master's Thesis, Naval Postgraduate School, Monterey, California, December, 1992.

[Ironics 91]Ironics Inc., *Sparc User's Manual*, Ironics Inc, pp. 24-32, 1991.

[Iyengar 91]Iyengar, S., and Elfes, A., *Autonomous Mobile Robots*, v. 1, p. 5, IEEE Computer Society Press, 1991.

[Jarvis 93] Jarvis, R., "A Perspective on Range Finding Techniques for Computer Vision," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1017-1024, May, 1993.

[Johnson 92]Johnson, E. and Reichard, K., *X Window Applications Programming*, 2nd. ed., MIS Press, 1992.

[Kanayama 83]Kanayama,Y., "Concurrent Programming of Intelligent Robots," *Proceed-*

ings of the Eighth Int. Conf on Artificial Intelligence, pp. 834-838, August, 1983.

[Kanayama 86]Kanayama,Y., and Norihisa, M., "Trajectory Generation for Mobile Robots," *Robotics Research*, pp. 333-340, MIT Press, 1986.

[Kanayama 88]Kanayama,Y., and Yuta, B., "Vehicle Path Specification by a Sequence of Straight Lines," *IEEE Journal on Robotics and Automation*, v. 4(3), pp. 265-276, June, 1988.

[Kanayama 89]Technical Report of the University of California, Santa Barbara, TRCS 89-06, *Spatial Learning by an Autonomous Mobile Robot with Ultrasonic Sensors*, by Y. Kanayama and T. Noguchi, pp. 1-13, 1989.

[Kanayama 89a]Kanayama,Y., and Hartman, B., "Smooth Local Path Planning for Autonomous Vehicles," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1265-1270, 1989.

[Kanayama 90]Kanayama, Y., Kimura, Y., Miyazaki, F., and Noguchi, T., "A Stable Tracking Control Method for an Autonomous Mobile Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 384-389, May, 1990.

[Kanayama 90a]Technical Report of Naval Postgraduate School, *Locomotion Functions in the Mobile Robot Language, MML*, Y. Kanayama and M. Onishi, pp. 1-13, 1990.

[Kanayama 91]Kanayama, Y., "Introduction to Spatial Reasoning," Lecture Notes of the Advanced Robotics, Dept. of Computer Science, Naval Postgraduate School, Fall Quarter, 1991.

[Kanayama 91a]Kanayama, Y., and Onishi, M., "Locomotion Functions in the Mobile Robot Language, MML," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1110-1115, 1991.

[Kanayama 93] Kanayama, Y., MacPherson, D., and Krahn, G., "Vehicle Motion Control and Analysis Using 2D Transformations," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 3-13 - 3-18, 1993.

[Kinsler 82] Kinsler, L., et. al., *Fundamentals of Acoustics*, 3rd ed., pp. 98-123, Wiley, 1982.

[Kosaka 93]Kosaka, A., Meng, M., and Kak, A., "Vision Guided Mobile Robot Navigation Using Retroactive Updating of Position Uncertainty," *Proceedings of the IEEE Conference on Robotics and Automation*, v. 2, pp. 1-7, 1993.

[Krotkov 89]Krotkov, E., "Mobile Robot Localization Using a Single Image," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 978-983, 1989.

[Kuan 85] Kuan, D., Zamiska, J., and Brooks, R., "Natural Decomposition of Free Space for Path Planning," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 168-173, 1985.

[Kuc 87] Kuc, R., and Siegel, M., "Physically Based Simulation Model for Acoustic Sensor Robot Navigation," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, v. 9(6), pp. 766-778, November, 1987.

[Kuc 90] Kuc, R., "A Spatial Sampling Criterion for Sonar Obstacle Detection," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, v. 12(7), pp. 686-690, July, 1990.

[Kuc 91] Kuc, R., and Viard, V., "A Physically Based Navigation Strategy for Sonar-Guided Vehicles," *The International Journal of Robotics Research*, v. 10(2), pp. 75-87, April, 1991.

[Kuipers 88]Kuipers, B., and Byun, Y., "A Robust, Qualitative Approach to a Spatial Learning Mobile Robot," *SPIE*, v. 1003, 1988.

[Laird 91] Naval Ocean Systems Command, Technical Document 2171, *Development of a Modular Robotic Architecture*, R. Lair, R. Smurlo, and S. Timmer, p. 3, September, 1991.

[Latombe 91]Latombe, J., *Robot Motion Planning*, pp. 7-54, Kluwer Academic Publishers, 1991.

[Laumond 83]Laumond, J., "Model Structure and Concept Recognition: Two Aspects of Learning for a Mobile Robot," *Proc. Eight IJCAI-83*, pp. 839-841, August, 1983.

[Leary 92]Leary, W., "Robot Named Dante to Explore Inferno of Antarctic Volcano," *New York Times*, p. B7, December 8, 1992.

[Leonard 90]Leonard, J., Cox, I. and Connell, J., "Dynamic Map Building for an Autonomous Mobile Robot," *Proceeding of the IEEE International Workshop on Intelligent Robots and Systems '90 (IROS'90)*, pp. 89-96, 1990.

[Leonard 91]Leonard, J. and Durrant-Whyte, H., "Mobile Robot Localization by Tracking Geometric Beacons," *Proceedings of the IEEE Conference on Robotics and Automation*, v. 7(3), pp. 376-382, June 1991.

[Leonard 92]Leonard, J., *Directed Sonar Sensing for Mobile Robot Navigation*, pp. 1-45, Kluwer Academic Publishers, 1992.

[Leonard 93] Interview between Dr. John Leonard, MIT Sea Grant College, Cambridge Massachusetts, and the author, 15 March 1993.

[Lim 92] Lim, J., and Cho, D., "Physically Based Sensor Modeling for a Sonar Map in a Specular Environment," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1714-1719, May, 1992.

[Lozano-Perez 79]Lozano-Perez, T., and Welsey, M. "An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles," *Communications of the ACM*, v. 22 (10), pp. 560-570, 1979.

[Lozano-Perez 83]Lozano-Perez, T., "Spatial Planning: A Configuration Space Approach," *IEEE Transaction on Computers*, v. C-32(2), pp. 108-119, February 1983.

[MacPherson 92]Naval Postgraduate School Monterey Ca., Technical Publication, Draft, *Yamabico User's Manual.*, D. MacPherson, p. 32, August 1992.

[Manber 89]Manber, U., *Introduction to Algorithms A Creative Approach*, AddisonWelsey, pp. 189-197, 1989.

[Mandel 87]Mandel, K., and Duffie, N., "On-Line Compensation of Mobile Robot Docking Errors," *IEEE Journal of Robotics and Automation*, v. RA-3(6), pp. 591-598, December, 1987.

[Mason 93]Mason, M., "Kicking the Sensing Habit," *Artificial Intelligence Magazine*, pp. 58-63, Spring, 1993.

[Mataric 92]Mataric, M., "Integration of Representation Into Goal-Driven Behavior-Based Robots," *IEEE Transactions on Robotics and Automation*, v. 8(3), pp. 304-312, June, 1992.

[McDermott 92]McDermott, D., "Robot Planning," *AI Magazine*, v. 13(2), pp. 55-79, Summer, 1992.

[Meystel 91]Meystel, A., *Autonomous Mobile Robots Vehicles with Cognitive Control*, v. I, World Scientific Publishing Co., Inc., 1991.

[Modada 93]Mondada, F., *Khepera - A Miniature Mobile Robot*, unpublished description of the robot Khepera, Laboratoire de Microinformatique, Lausanne, Switzerland, 1993.

[Moravec 81]Moravec, H., "Robot Rover Visual Navigation," *UMI Res. Press*, 1981.

[Moravec 82]Moravec, H., "The CMU Rover," *Proc. Nat. Conf. Artificial Intelligence*, pp. 377-380, August, 1982.

[Moravec 87]Moravec, H. "Certainty Grids for Mobile Robots," *Proceedings of the Workshop on Space Telerobotics*, JPL, Pasadena, CA, January, 1987.

[Motorola 85]Motorola, "MC68020 32-Bit Microprocessor User's Manual," 2nd ed., pp. 5-26, Prentice Hall, 1985.

[Myers 92]Myers, K. and Konolige, K., "Semi-Autonomous Map-Making and Navigation," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 129-133, 1992.

[Nachtigall 86]Nachtigall, P., *Animal Sonar*, Plenum Press, pp. 123-137, 1988.

[Nehmezow 92]Nehmezow, U., *Experiments in Competence Acquisition for Autonomous Mobile Robots*, Doctoral Dissertation, University of Edinburgh, pp. 231-247, 1992.

[Nehmezow 93]Nehmezow, U., "Animal and Robot Navigation," submitted to NATO-ASI

Workshop on the Biology and Technology of Intelligent Autonomous Agents, March, 1993.

[Nelson 88]Nelson, W. and Cox, I., "Local Path Control for an Autonomous Vehicle," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1504-1510, 1988.

[Nilsson 69]Nilsson, N. J., "A Mobile Automaton: An Application of Artificial Intelligence Techniques," *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 509-520, Washington, D. C., May 7-9, 1969.

[Noborio 90]Noborio, H., Kondo, K., and Noda, A., "A Good Extension Method of Free-Space in an Uncertain 2D Workspace by Using Ultrasonic Sensors," *Proceedings of the IEEE Conference on Robotics and Automation*, p. 384, 1990.

[Noreils 89]Noreils, F., and Chatila, R., "Control of Mobile Robot Actions," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 701-707, May, 1989.

[Noreils 92]Noreils, F., and Chatila, R., "Control of Mobile Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1482-1488, May, 1992.

[Olin 91] Olin, K., and Tseng, D., "Autonomous Cross-Country Navigation: An Integrated Perception and Planning System", *IEEE Expert*, v. 6(4), pp. 16-30, August, 1991.

[O'Rourke 87]O'Rourke, J., *Art Gallery Theorems and Algorithms*, pp. 1, 125-126, 141, Oxford University Press, 1987.

[Paul 84] Paul, R., *Robot Manipulators: Mathematics, Programming, and Control*, pp. 7-12, MIT Press, 1984.

[Payton 90]Payton, D., Rosenblatt, J., and Keirsey, D., "Plan Guided Reaction," *IEEE Transactions on Systems, Man, and Cybernetics*, v. 20(6), pp. 1370-1382, November/December, 1990.

[Payton 91] Payton, D., and Bihari, T., "Intelligent Real-Time Control of Robotic Vehicles," *Communications of the ACM*, v. 34(8), pp. 48-63, August, 1991.

[Ratering 92] Ratering, S. and Gini, M., "Robot Navigation in a Known Environment with Unknown Moving Obstacles," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 154-158, 1992.

[Reister 91] Reister, D., "A New Wheel Control System for the Omnidirectional Hermies-III Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2322-2327, April, 1991.

[Reister 91a]Reister, D., Jones, J., Butler, P., Beckerman, M., and Sweeney, F., "Demo 89 - The Initial Experiment with the Hermies-III Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2562-2567, April, 1991.

[Roa 91] Roa, N., Stoltzfus, N., and Iyengar, S., "A Retraction Method for Learned Nav-

igation in Unknown Terrians for a Circular Robot," *IEEE Transaction on Robotics and Automation*, v. 7(5), pp. 699-707, October, 1991.

[Rolfe 86] Rolfe, J., and Staples, K., *Flight Simulation*, pp. 45-51, Cambridge University Press., 1986.

[Sales 74] Sales, G., and Pye, D., *Ultrasonic Communications by Animals*, pp. 23-68, Chapman and Hall, Inc., 1974.

[Schwartz 83]Schwartz, J., and Sharir, M., "On the Piano Mover's Problem II: General Techniques for Computing Topological Properties of Real Algebraic Manifolds," *Adv. Applied Math.*, v. 4, pp. 298-351, 1983.

[Schwartz 88]Schwartz, J., and Sharir, M., "A Survey of Motion Planning and Related Geometric Algorithms," *Artificial Intelligence Journal*, v. 37, pp. 157-169, 1988.

[Sherfey 91] Sherfey, S., *A Mobile Robot Sonar System*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1991.

[Smurlo 92]Smurlo, R., and Everett, H. "Intelligent Security Assessment for a Mobile Robot," *Proceedings Sensor Expo 1992*, pp. 125-132, Helmers Publishing, Inc., 1992.

[Simmons 91]Simmons, R., "Concurrent Planning and Execution for a Walking Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 300-305, 1991.

[Simmons 92]Simmons, R., Fedor, C., and Basista, J., "Task Control Architecture: Programmers Guide to Version 6.17," Carnegie-Mellon Univ, School of Computer Science / Robotics Institute, November, 1992.

[Skewis 92]Skewis, T., and Lumelsky, V., "Experiments with a Mobile Robot Operating in a Cluttered Unknown Environment," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 1482-1482, May, 1992.

[Sugihara 87]Sugihara, K., "Location of a Robot Using Sparse Visual Information," *Fourth International Symposium on Robotics Research*, pp. 319-326, MIT Press, 1987.

[Sweeney 90]Sweeney, F., "DOE/NE University Program in Robotics for Advanced Reactors - Program Plan: FY1990 - FY 1994," DOR/OR - 884/R2, U.S. Dept. of Energy, Oak Ridge, Tennessee, 1990.

[TAE 90] Transportable Applications Environment (TAE) Plus User's Manual, National Aeronautics and Space Adminstration, Goddard Space Flight Center, January, 1990.

[Thorpe 84]Thorpe, C., *FIDO: Vision and Navigation for a Robot Rover*, Doctoral Dissertation, Carnegie-Mellon University, Pittsburgh, PA., December, 1984.

[Thorpe 91] Thorpe, C., et al., "Towards Autonomous Driving: The CMU Navlab," *IEEE*

Expert, pp. 31-52, August, 1991.

[Thrun 92] Thrun, S., "Exploration and Model Building in Mobile Robot Domains," *IEEE International Conference on Neural Networks*, Van Nostrand, pp. 123-135, 1992.

[Thrun 93] Thrun, S., "The Role of Exploration in Learning Control," *Handbook of Intelligent Control: Neural, Fuzzy, Adaptive Approaches*, p. 23., March 28 - April 1, 1993.

[Turk 88] Turk, M. A., et al., "VITS—A Vision System for Autonomous Land Vehicle Navigation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. 10(3), pp. 342-361, May, 1988.

[van Turennout 92]van Turennout, P., Honderd, G., and van Schelvan, L., "Wall-following Control of a Mobile Robot," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 280-285, May,1992.

[Wallich 91]Wallich, P. "Silicon Babies," *Scientific American*, pp. 124-134, December, 1991.

[Watanabe 90]Watanabe, Y., and Yuta, S., "Position Estimation of Mobile Robots With Internal and External Sensors Using Uncertainty Evolution Technique," *Proceedings of the IEEE Conference on Robotics and Automation*, pp. 2011-2016, 1990.

[Williams 92]Williams, T., and Kelley, C., *GNUPLOT An Interactive Plotting Program Manual*, Version 3.0, 10 June 1992.

[Webster 87]*Webster's New Collegiate Dictionary*, Merriam Co., 1987.

[Weisbin 89]Weisbin, C., et al., "Autonomous Mobile Robot Navigation and Learning," *Computer*, pp. 29-35, June 1989.

[Winston 92]Winston, P., *Artificial Intelligence*, pp. 443-469, Addison-Wesley, 1992.

[Wolfe 88]*Robots II*, v. 852, pp. 256-261, Wolfe, W., Chun, W., Eds., *Proc. SPIE*, 1988.

[Yen 92]Yen, J., and Pfluger, N., "A Fuzzy Logic Based Robot Navigation System," *AAAI Fall Symposium on the Applications of Artificial Intelligence to Real World Robots*, pp. 195-199, 1992.

[Zelinsky 88] Zelinsky, A., "Environmental Mapping with a Mobile Robot Using Sonar," *Proc. of the Australian Joint Artificial Intelligence Conf. - AI'88*, pp. 373-388, 1988.

[Zhang 92] Zhang, Z., and Faugeras, O., "A 3D World Model Builder with a Mobile Robot," *The International Journal of Robotics Research*, v. 11(4), pp. 269-285, August, 1992.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Department of Computer Science Code CS Naval Postgraduate School Monterey, California 93943	1
4.	Computer Technology Programs Code 37 Naval Postgraduate School Monterey, California 93943	1
5.	Professor Yutaka Kanayama, Code CS/Ka Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
6.	Professor Michael Zyda, Code CS/Zk Department of Computer Science Naval Postgraduate School Monterey, California 93943	1
7.	Professor Craig Rasmussen, Code MA/Sh Department of Mathematics Naval Postgraduate School Monterey, California 93943	1
8.	Professor A. J. Healey, Code ME/Hy Department of Mechanical Engineering Naval Postgraduate School Monterey, California 93943	1

10. Professor Timothy Shimeall, Code CS/Sm — 1
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

11. Mr. David Leonard MacPherson, Sr. — 1
96 Kaydeross Park Road
Saratoga Springs, NY 12866-8702

12. Mr. Edward J. LaVigne — 1
900 Ridge Road
Lansing, NY 14882

13. MAJ R. B. Byrnes, Jr. — 1
Software Technology Branch, ARL
115 O'Keefe Building
Georgia Institute of Technology
Atlanta, GA 30332-0800

14. LCDR Donald Brutzman, Code CSPH — 1
Department of Computer Science
Naval Postgraduate School
Monterey, California 93943

15. Jean-Claude Latombe — 1
Robotics Laboratory
Department of Computer Science
Stanford University
Stanford CA 94305