

Software Architecture, Path Planning, and Implementation for an Autonomous Robot

by

Terence Y. Chow

S.B., Massachusetts Institute of Technology (1994)

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of


Master of Science in Mechanical Engineering

at the

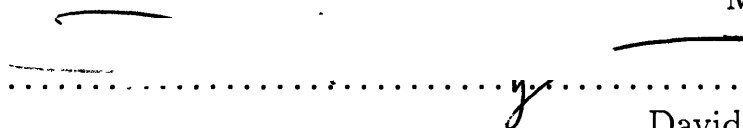
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Terence Y. Chow, 1996. All rights reserved.

Author 

Department of Mechanical Engineering
May 1996

Approved by 

David S. Kang
Technical Supervisor

Certified by 

Kamal Youcef-Toumi
Associate Professor
Thesis Supervisor

Accepted by 

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Ain Sonin
Chairman, Departmental Committee

JUN 27 1996

LIBRARIES
Eng.

Software Architecture, Path Planning, and Implementation for an Autonomous Robot

by
Terence Y. Chow

Submitted to the Department of Mechanical Engineering
on May 1996, in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

A software architecture, including path planning capabilities, was designed and implemented for Companion, an autonomous mobile robot. The software architecture consists of modules with specific responsibilities, and these modules were implemented in the C programming language on the QNX operating system.

Modules performing functions such as navigation, trajectory following, and path planning were implemented and tested. Navigation, achieved through dead reckoning, was accurate to a position within 1.2 percent of path length. A trajectory following system, which allowed the robot to follow lines and circles, worked adequately despite some transient overshoot and a steady-state off line error of 0.03 meters. Satisfactory path planners for a point robot in a plane and for a rectangular robot with a turning constraint were implemented and tested.

Although individual functions were tested, the primary focus was to achieve good overall system performance. The software architecture and implementations are a strong framework from which improvements and additions can be made.

Technical Supervisor: David S. Kang

Thesis Supervisor: Kamal Youcef-Toumi

Title: Associate Professor

Acknowledgments

For this opportunity, I must first thank my Draper supervisor Dr. David Kang, my MIT advisor Prof. Kamal Youcef-Toumi, and my first Draper colleagues: Mark Abramson, Bill Hall, and Bob Powers.

I am forever grateful to Steve Steiner, my partner in crime, who endured endless hours of debugging with me. Kudos to everyone else who participated on the project, in particular to Sean Adam, for his patience in debugging; Bill Kaliardos, for replacing all those pots and encoders; and Chuck Tung for the new gyro that got really going. I must also acknowledge all the other guys in the lab who provided endless entertainment.

I thank all of my friends. They have made MIT much more than an engineering experience.

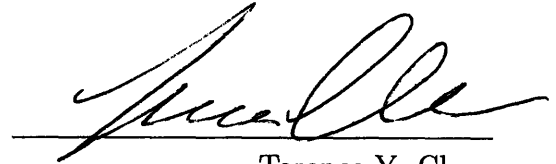
Finally, thanks to family: Irene, Mom, and Dad.

Biographical Note

Terence Chow was born and raised near Dayton, Ohio. He graduated from Centerville High School in 1990 and completed dual Bachelor of Science Degrees in Mechanical Engineering and Mathematics at the Massachusetts Institute of Technology in May 1994. He completed his Master of Science degree at MIT in May 1996 and will begin a career in software development at Oracle in California.

This thesis was supported by The Charles Stark Draper Laboratory, Inc. (Integrated Sensor Fusion Demo Project IR&D 514). Publication of this thesis does not constitute approval by The Charles Stark Draper Laboratory, Inc., of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

I hereby assign my copyright of this thesis to The Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.

A handwritten signature in black ink, appearing to read 'Terence Y. Chow', is written over a horizontal line.

Terence Y. Chow

Permission is hereby granted by the Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce part or all of this thesis.

Contents

1	Introduction	15
1.1	Companion: A Brief History	16
1.2	Reaching Companion Goals	17
1.3	This Thesis	18
2	Hardware and Operating System	19
2.1	Mobility Platform	19
2.2	Sensors	19
2.2.1	Hazard Detection Sensors	20
2.2.2	Mapping Sensors	20
2.2.3	Navigation Sensors	20
2.2.4	Actuator State Sensors	21
2.2.5	Other Proposed Sensors	21
2.3	Computers	21
2.3.1	Tower	21
2.3.2	Laptop	22
2.4	Operating System	22
2.4.1	QNX: An Overview	23
2.4.2	Message Passing in QNX	23
3	Software Architecture and Implementation Models	25
3.1	Software Architecture	25
3.2	Implementation Models	27
3.2.1	Blocked Process Model	28
3.2.2	Spinning Process Model	29
3.3	Modules	31
3.3.1	Sound Module	31
3.3.2	Cycle Module	32
3.3.3	Sonar Module	33
3.3.4	Laser Module	33
3.3.5	Trajectory Module	34
3.3.6	Mapper Module	34
3.3.7	Search3d Module	35
3.3.8	Search2d Module	35
3.3.9	Planner Module	35

3.3.10	User Module	35
4	Implementation and Testing of the Cycle Module	37
4.1	Hazard Detection	37
4.2	Motor Actuation	37
4.2.1	Drive Motor Actuation	38
4.2.2	Steering Motor Actuation	38
4.3	Navigation	40
4.3.1	Dead Reckoning Equations	40
4.3.2	Testing of Dead Reckoning	44
5	Implementation and Testing of the Trajectory Module	47
5.1	Circle Following	47
5.1.1	Circle Following Controller Derivation	47
5.1.2	Stopping Condition	50
5.1.3	Parameter Selection and Testing	50
5.2	Line Following	52
5.3	Multiple Command Execution	52
6	Path Planning: Background and Overview	55
6.1	Requirements	55
6.2	Background	56
6.2.1	A* Algorithm	56
6.2.2	Visibility Graph Path Planning	57
6.2.3	Configuration Space Path Planning	59
6.2.4	Potential Field Path Planning	61
6.3	Overview of Companion Path Planning	61
7	Implementation and Testing of the Search3d Module	63
7.1	Search3d Implementation Details	63
7.1.1	Neighboring Rule	64
7.1.2	Cost Calculation	67
7.1.3	Heuristic Calculation	67
7.2	Testing of the Search3d Algorithm	68
8	Implementation and Testing of the Search2d Module	73
8.1	Derivation of the Search2d Algorithm	73
8.1.1	Revised A* Algorithm	74
8.1.2	Neighboring Rules	75
8.1.3	Cost and Heuristic Calculation	77
8.1.4	Review of the Search2d Algorithm	77
8.2	Testing of the Search2d Module	77
9	Implementation and Testing of the Planner Module	79

10 Conclusions and Recommendations	83
10.1 Ideas for Improvement	83
10.2 Ideas for New Development	84
10.3 End Game—A Final Commentary	84
A Search2d Neighbor Tests	87
B Implementation of the User Module	89

List of Figures

1-1	Companion	15
3-1	Modules in the Software Architecture	26
3-2	Flow Chart for Blocked Process Model	28
3-3	Flow Chart for Blocked Process Parent	29
3-4	Flow Chart for Spinning Process Model	30
3-5	Flow Chart for Spinning Process Parent	30
4-1	Steering Model	38
4-2	Dead Reckoning: Motion During a Cycle	41
4-3	Dead Reckoning: Center-Rear Motion	43
4-4	Dead Reckoning: Robot Center Motion	44
4-5	Long Distance Testing of Dead Reckoning	45
4-6	Sharp Turning Testing of Dead Reckoning	45
5-1	Geometry of Circle Following	48
5-2	Transformed Geometry of Circle Following	49
5-3	Circle Following—No Error Case	50
5-4	Circle Following Tests, $d_t = 0.25$	51
5-5	Circle Following Tests, $d_t = 0.50$	51
5-6	Multi-Command Trajectories	52
5-7	Multi-Command Cusp Trajectory	53
6-1	A Visibility Graph	58
6-2	A Reduced Visibility Graph	58
6-3	A Configuration Space	60
6-4	A Neighborhood	60
6-5	An Expanded Neighborhood	61
7-1	Neighbors for Search3d—First Iteration	64
7-2	Adjustment of Path at a Cusp	65
7-3	Adjustment of Path at a Steering Transition	65
7-4	Neighbors for Search3d—Second Iteration	66
7-5	Turning Circles for Search3d Heuristic Calculation	68
7-6	External Tangent Paths for Search3d Heuristic Calculation	69
7-7	Internal Tangent Paths for Search3d Heuristic Calculation	69
7-8	Example of the four Paths for One Tangent	70

7-9	Sample Paths Generated by the Search3d Module	71
8-1	Sample Search Situation	74
8-2	Example of a New Obstacle Found	76
8-3	Sample Run of Search2d	78
9-1	Corner-Turning Plan Execution	80
9-2	Doorway Plan Execution	80
A-1	Tangency Test for Neighbors	88
A-2	Close up of an Obstacle	88

List of Tables

- 2-1 Modules on the Companion Tower 22
- 3-1 Module Implementation Summary 31
- 4-1 Results of Steering Control Testing 39
- 4-2 Results of Long Distance Dead Reckoning Testing 44
- 4-3 Results of Sharp Turning Dead Reckoning Testing 46
- 7-1 Allowed Neighbor Transitions 67

Chapter 1

Introduction

Companion, shown in Figure 1-1, is an autonomous mobile robot in the Intelligent

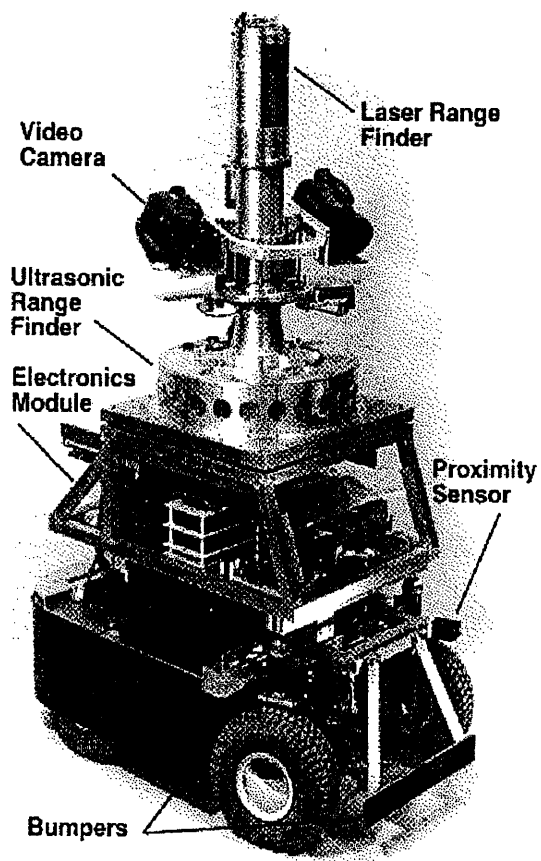


Figure 1-1: Companion

Unmanned Vehicle Center at the Charles Stark Draper Laboratory. It is a test bed

for mapping, navigation¹, and path planning. It features an array of sensors including bumpers, proximity detectors, sonars, a laser range finder, a gyroscope, and encoders. Its mobility platform is an outfitted electric wheelchair. Its on board processing is done by a networked pair of computers running a real time operating system.

This thesis is a description of the design, development, and testing effort on Companion since October 1994. At the time, most of Companion’s electrical and mechanical components had been built and some software had been written, but the system had not been tested. Not surprisingly, the robot did not work. The task was to bring Companion to a “working” state. What “working” meant was nebulous at the time, but a set of goals was eventually formulated. The achievement of those goals are the basis of this thesis.

The software development of Companion since late 1994 was led by this author and another M.I.T. graduate student, Steve Steiner. The goals, as they evolved during work on the project were the following:

- Write device drivers to interface all sensors and actuators with software
- Design a software architecture capable of coordinating robot motion, world mapping, and path planning, and also develop implementation models for software components that leverage the existing operating system
- Implement the software system—motion control, mapping, and planning
- Test the system and evaluate the system’s performance

The work was collaborative on most of these areas. Exceptions are that Steiner designed and implemented most of the mapping software and this author implemented most of the planning software. In this thesis, all of the above topics are covered, with the exception of device drivers² and the mapping software.³

The remainder of this chapter is an introduction to Companion, including a history of the project and a description of goals for Companion. In the final section, a road map for the rest of this document is provided.

1.1 Companion: A Brief History

The Companion project began in 1993 with the initiative of Dr. David Kang and internal funding from the Charles Stark Draper Laboratory. At the time, the unnamed robot was billed as an earth-based, integrated sensor fusion platform. Soon after, the

¹The meaning of navigation throughout this document is the practice of recording the position of the robot, usually as a triple (x, y, θ) .

²The device drivers are a difficult issue. Their development was the most frustrating, difficult, and time-consuming problem overcome on the project. For that reason, they cannot be neglected. However, they are more a means than an end: sensors and actuators had to function in order to test mapping and planning programs.

³The mapping software is fully documented in Steiner’s thesis, [16].

official focus shifted to developing a robot that could serve as a soldier's companion in the battlefield. From this original proposal, the robot was dubbed "Companion."

A team of students and Draper employees hence began work on Companion. To expedite the development of a working mobility platform, Companion took the form of an outfitted electric wheelchair. It was designed to house a variety of sensors, with significant processing capabilities. In the ensuing two years, work on Companion included construction on the mobility platform, wiring of electrical circuits, and installation of an operating system for the robot's computers. This continuing effort resulted in a collection of components, some reliable and some hacked. The interface of the hardware to software (device drivers) also spanned the spectrum of reliability.

In late 1994, Companion still suffered from problems ranging from incomplete mechanical systems to electronic circuitry bugs to software bugs. At that time, little focus had been placed on software—neither the development of custom planning and mapping software nor the testing of device drivers. Over the following year, we worked to build reliable device drivers and debug previously unseen hardware problems. In February 1996, the last of the known critical hardware problems was solved, and the drivers had been completed.

In nearly all cases, efforts were spent on writing reliable drivers for *existing* sensors and actuators. This work, with respect to hardware, was a debugging role rather than a design role. The foresight of Companion's first designers resulted in a sensor-rich robot capable of autonomous behavior.

During the development and debugging of device drivers, a parallel effort resulted in the design of the robot's software architecture. Strategies for mapping and path planning were also researched. The eventual implementation and testing of mapping and path planning software is the most recent event in Companion's evolution.

1.2 Reaching Companion Goals

The real-world goal for Companion is the ability to autonomously navigate though cluttered environments. As an example, Companion might be able to roam the halls of the Draper Laboratory without colliding with photocopiers, door jambs, or engineers. Such performance requires minimum competence in the areas listed at the beginning of this chapter. This section is an expansion of the intermediate tasks that progressed Companion towards its ultimate goal.

The first area of work consisted of the implementation of reliable device drivers. These routines linked the hardware and software. Companion, having a wide array of sensors and actuators, required a substantial development and debugging effort in this area.

The second area of work was the design of a software architecture and implementation models for its components. The architecture organized the function and interaction of the different software components. The models were paradigms from which actual code was developed. Following these skeletal examples provided a sane manner in which to manage the multi-process, multi-processor, multi-person development effort.

Implementation of the software was the third task, and it was divided into three main areas. The first area, motion control, required navigation and the ability to follow simple trajectories such as lines and circles. The implementation of mapping involved the fusion of sonar and laser readings into a single representation of the environment. This map had to be accurate so that the robot could plan paths based on the information obtained from it. The third area of implementation was path planning. Companion's path planner had to quickly generate commands to the actuators to move the robot around obstacles to a goal position.

The final job, which actually occurred all through development, was the testing of the system. The testing included testing on individual segments of the software system as well as on the overall system.

1.3 This Thesis

The main purpose of this thesis is to document Companion's main software components, with the exception of the mapping software. The next chapter is an overview of Companion's electrical hardware, mechanical hardware, computers, and operating system. Following that is a description of the software architecture and implementation models. Two components of the software system, Cycle and Trajectory, are then discussed. After that, four chapters discuss the robot's planning system. Finally, the conclusion of this document contains suggestions for future work. The time has come to begin.

Chapter 2

Hardware and Operating System

Companion is a collection of commercial and homemade components, acquired or built by Draper students and staff. This chapter describes many of the robot's important components, including the mobility platform, sensors, and computers. The QNX operating system running on the system's computers is also discussed.

2.1 Mobility Platform

Companion began as a Joystick Sparky electric wheelchair made by the Electric Mobility Corporation. The main considerations in the selection of the mobility platform were the performance of the vehicle and the ease with which it could be outfitted for robotic purposes. The Joystick Sparky was selected because it has a powered drive motor, a powered steering motor, and a small turning radius (0.7 meters). According to the Joystick Sparky specifications, the platform has a top speed of 2.5 meters per second, a load capacity of 200 kilograms, and a range of 32 kilometers. Because Companion's batteries supply power to other motors and electronic components, the range is actually smaller.

The electric drive motor passes through a two-speed gear box and a differential to provide power to the rear two wheels. The front wheels are not powered but steerable. During a turn, the inner wheel turns more sharply than the outer wheel, such that the projections of the axles of all four wheels meet at a single point. This is the geometry of the Ackermann steering gear layout[1] that minimizes tire slippage.

The Joystick Sparky's original hardware included a joystick that drove circuitry to control speed and steering. Although this circuitry has undesirable hysteresis and delay, it was left intact. The joystick was replaced with outputs from a D/A converter. Hence, control of the mobility platform's motors first passes through the original Joystick Sparky control circuitry.

2.2 Sensors

Companion features a variety of sensors. They can be divided into four categories by their purpose: hazard detection, mapping, navigation, and actuator state. This

section describes these sensors and briefly discusses some others that were at one time considered for implementation and that may be added in the future.

2.2.1 Hazard Detection Sensors

For hazard detection, Companion has bumpers and proximity detectors. The eight bumpers form a ring around the robot's outer circumference, with the exception of the front wheels, which are exposed. The bumpers behave as electric switches when objects come in contact with them.

Companion also has eight Aromat area reflective photoelectric sensors. These proximity sensors are tuned to trigger when objects are about 0.15 meters away. Four of the sensors are rear-facing. Two are mounted on the steering mechanism and rotate with the front wheels. The last two are side-facing near the front of the robot.

2.2.2 Mapping Sensors

For environmental mapping, Companion possesses an array of sonars and a laser range finder. The sonar array consists of 24 Polaroid ultrasonic transducers and ranging modules. The sonars have a cone width of 0.32 radians and a range up to 10 meters. An 8-bit A/D conversion limits the resolution of the readings to about 0.04 meters. The sonars are configured in an outward-facing ring of radius 0.21 meters mounted 0.76 meters off the ground (refer back to Figure 1-1); this provides good sonar coverage in lateral directions.

Companion also has a laser range finder (an Acuity Research AccuRange 3000) configured for 300 samples per second at 0.008 meter resolution. The laser is mounted vertically inside a cylindrical housing protruding from Companion's sonar structure (again see Figure 1-1). The housing also contains a mirror that can both yaw and pitch. This provides a full lateral coverage around the robot at pitch angles ranging from straight down to beyond horizontal. Both motors have encoders and are controlled by a Motion Engineering motor controller board (see Subsection 2.3.1). Although use of the laser requires actuation of yaw and pitch motors, the laser and its motors are usually regarded as a single sensor.

2.2.3 Navigation Sensors

For navigation, Companion carries a gyroscope and two encoders. The rate gyroscope, a Systron Donner GyroChip II, drifts at less than 0.06 radians per hour and a linear acceleration sensitivity less than 0.06 degrees per second per g . The gyroscope has a dedicated Little Giant Zilog processor responsible for sampling and integrating angular rate to angular position. The processor also manages serial line communication to other computers.

Companion has BEI incremental optical quadrature encoders mounted on each of the front two wheels. The encoders yield 580 counts per meter traveled by the robot, for a resolution of 0.00172 meters.

2.2.4 Actuator State Sensors

The actuator state sensors provide feedback for the control of motors. The laser pitch and yaw motors have built-in encoders monitored by the motion control board. The only actuator state sensors are two steering potentiometers. These potentiometers measure the position of each of the front wheels relative to the chassis. Their use in the actuation of the steering motor is discussed in Subsection 4.2.2.

2.2.5 Other Proposed Sensors

Other sensors have been proposed for Companion, including a compass, accelerometers, a vision system, and GPS. Some of these sensors are partially completed, but because they are not yet integrated into the system, they are not of specific interest. However, the software architecture should allow for easy integration should these sensors become available.

2.3 Computers

Companion's main processing is done by two networked 486 computers.¹ One computer (referred to as the "Tower") is dedicated to communication with the robot's hardware—the reading of sensors and the actuation of motors. The other computer (called the "Laptop") is devoted to sensor fusion and path planning. The two computers are networked via ethernet and are both running the QNX operating system (described in Section 2.4).

The motivation behind the selection of these computers is that they are off-the-shelf, commercial products. This provides the advantages of technical support, easily replaced parts, and in some cases, more affordable components. In addition, it allows work to focus on integration of the components rather than debugging them.

2.3.1 Tower

The first computer, the Tower, is a stack of PC/104 boards: a CPU module, a serial/parallel port module, a digital/analog I/O module, an ethernet module, and a motor control module. The manufacturers and relevant specifications of these modules are summarized in Table 2-1.

The primary function of the Tower is to communicate with the robot's sensors and actuators. Most of the Tower modules were purchased prior to the full design of the robot system, i.e. before it was known how many serial lines, digital I/O lines, etc. would be needed. The requirements of the modules were overestimated so that we could accommodate the addition of new sensors and actuators to the system. The minimal requirements were several digital I/O lines, several analog I/O lines, several

¹Companion actually has a third processor, a Z-World Z180 mentioned in Subsection 2.2.3. It is not discussed here because it is dedicated to Companion's gyroscope.

Table 2-1 Modules on the Companion Tower	
Module	Manufacturer and Product Specifications/Features
Processor	Ampro CoreModule/486 Cyril CX486SLC CPU and 2MB RAM
I/O	Real Time Devices DM406 16 analog and 16 digital I/O lines
Serial/parallel port	Ampro MiniModule/SSP 2 serial and 1 parallel ports
Ethernet	Ampro MiniModule/Ethernet-TP
Motor Control	Motion Engineering 104/DSP 4 axis control

serial ports and a parallel port. The use of a PC/104 bus also provides the ability to add special modules to the stack (such as the motor control module).

The configuration of the Tower went through several iterations before the final state described here. However, the primary function of the Tower has not changed: it is still a processor dedicated to interfacing with the robot's hardware.

2.3.2 Laptop

Companion's second computer is a Winbook XP laptop computer with an Intel 486DX4/100 CPU, 16MB of RAM, and a Linksys Combo PCMCIA EthernetCard. It is called the Laptop. Its function is to perform sensor integration and path planning. In the selection of this computer, interest was primarily in portability, processor speed, and cost. In essence, the requirement was a fast, affordable, laptop computer. The Laptop, with its own battery and display, has the additional feature of being an off-line development environment.

Historically, the Winbook is the second of two computers used for Companion's mapping and planning functions. An Inex Notebook Computer (486SLC/25) preceded it. Companion's performance improved with the faster computer, and because the transition from one laptop to another was relatively simple, processor upgrading may in the future be a efficient means of improving the robot's performance.

2.4 Operating System

At the heart of Companion's software is an operating system that manages the computers' resources, including the scheduling of programs running on the system and communication among them. This section discusses the QNX operating system, first as

an overview, and then in terms of its interprocess communication (IPC) architecture.

2.4.1 QNX: An Overview

QNX is the operating system running on Companion's computers. QNX is a UNIX-like, real time operating system achieving its efficiency, modularity, and simplicity through its microkernel architecture and its message-based IPC[14]. The QNX Kernel is small (8K) and dedicated to only two functions: the routing of messages among processes running on the system and the scheduling of processes to execute. IPC, a key in developing applications with multiple cooperating processes, is handled in QNX with *message passing*. QNX messages are packets of bytes sent from one process to another—the meaning of the bytes is left for the two processes to interpret.

The beauty of QNX's message passing IPC is that it can be done transparently over a network. Packets can be sent to and received from processes running on remote resources as though they were on the same computer. This architecture allows the development of a multi-process software system executing on several computers simultaneously without the difficult task of managing the inter-computer communication. In addition, it provides a convenient way to decrease the load of any one processor: distribute simultaneously running processes over more computers on the same network.

As a historical note, in the early stages of Companion design, a third computer was included as a dedicated processor for a vision system. This system has not yet been completed, but if it ever is, the use of QNX will allow easy integration of the new computer into the network and communication to processes running on it.

2.4.2 Message Passing in QNX

QNX message passing is accomplished through four library functions. These functions, provided in the C programming language,² are *Send()*, *Receive()*, *Creceive()*, and *Reply()*. To describe the procedural flow of a program utilizing these functions, a typical interaction of two processes using message passing is presented. The following sequence is based on an example in [14]. Suppose two processes, *A* and *B*, are running on the same network. The following events occur:

1. Process *A* issues a *Send()* request to process *B*. The request contains a packet of bytes for process *B* to interpret. After issuing the request, process *A* halts execution until it receives a reply to that request.
2. To accept the message, process *B* issues a *Receive()* request. It can then interpret the packet sent from process *A*. If no message is waiting for process *B* when it issues the *Receive()* request, it halts execution until a message becomes available.

²The implementation of QNX's message passing functions in C all but required Companion's software to be developed in C, but no one is complaining.

3. Once process *B* receives the message, it may interpret the data and possibly execute other instructions. It then issues a *Reply()* to the message. The reply also contains a packet of bytes that may be interpreted by process *A*. Process *A* resumes execution when a reply from process *B* becomes available, and the cycle is complete.

The *Creceive()* function was not used in this example. This function is similar to *Receive()* except that it does not block the issuing process. Instead, it accepts a message if there is one waiting and returns immediately. *Creceive()* allows a process to periodically poll for messages without causing the process to block. In the example, had process *B* used *Creceive()* instead of *Receive()*, it would have only accepted the message sent from process *A* if it were already waiting when *Creceive()* was called.

Processes in QNX are always in one of four states with respect to their communication with other processes. An executing process is in a READY state. If it issues a *Send()* request, it is SEND-blocked until the destination process receives the message, at which time it becomes REPLY-blocked. It returns to the READY state after it receives the reply. A process is RECEIVE-blocked if it has issued a *Receive()* request; it returns to the READY state after the message arrives.

As a preface to the next chapter, one finds that message passing is the best way to manage IPC in QNX. Failure to use message passing can only create inefficiencies in a QNX software system. With that in mind, read on.

Chapter 3

Software Architecture and Implementation Models

An important step in Companion’s development was the design of a software architecture to allow Companion to execute its mission. The point of the software architecture was to divide the large implementation task into smaller pieces. In the end, the development task was divided into individual *modules* that had their own specific responsibilities. The modules were integrated as they were completed.

The modules themselves also conformed to a standard. Implementation models were created to serve as paradigms for the real development. By following these paradigms, a standard interface among the modules was maintained. This made the integration of the components an easy task. The paradigms were also important because they leveraged the message-passing IPC feature of QNX.

This chapter first presents the software architecture—a network of modules. Then it describes the module models—a pair of implementation examples. Finally, the function of the individual modules are described. In later chapters, the inner workings of th some of the modules are covered. For now, the main interest is in the organization of the system and the interconnectivity of the modules.

3.1 Software Architecture

The software architecture, a product of legacy code, iteration, and foresight into the needs of the system, appears in Figure 3-1. To date, this figure is the best one-page summarization of Companion’s software yet devised. Each of the boxes, rounded and square, is one module in the system. Each module is a separate process running on Companion’s computers with its own responsibilities.

There is a hierarchical implication from Figure 3-1. “Parent” modules are connected to “child” modules via directed arrows. The User module tops the hierarchy; the Sound and Cycle module reside at the bottom. Although planning hierarchies are a current topic of research, they are not a primary interest in this thesis. The evolution of this structure came about mostly from an interest in manageable software implementation.

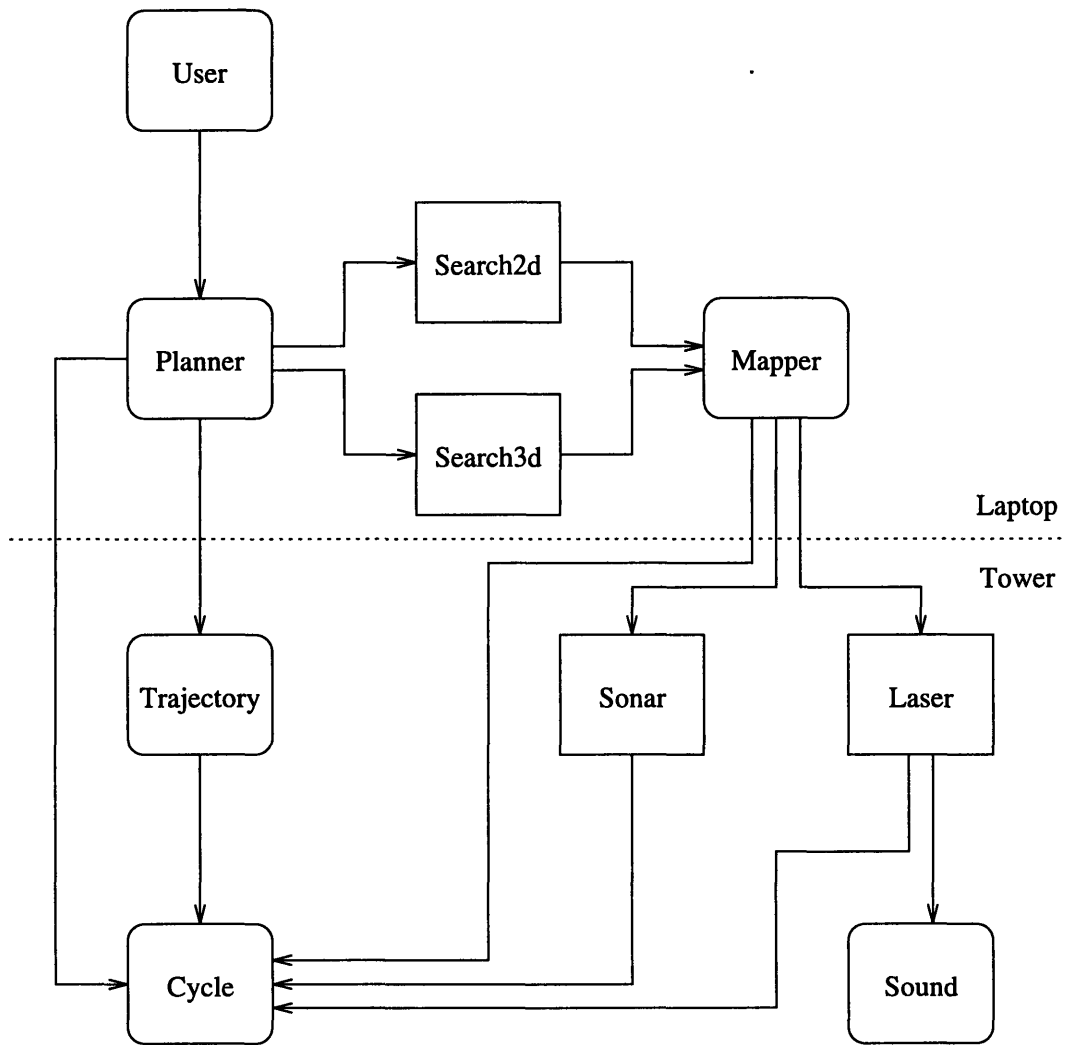


Figure 3-1: Modules in the Software Architecture

Although specific details about the modules appear later in this chapter, it is informative to have a brief functional overview here:

Sound This module manages the built-in speakers on the two computers.

Cycle This module reads hazard sensors, actuates motors, and performs navigation.

Sonar This module collects readings from the sonar array.

Laser This module collects readings from the laser range finder.

Trajectory This module commands the robot to follow line and circle trajectories.

Mapper This module integrates sensor readings into a map of the environment.

Search2d This module is a low-detail path planner.

Search3d This module is a high-detail path planner.

Planner This module coordinates path planning with plan execution.

User This module is the interface to the user.

The architecture is designed for extensibility. As an example, consider the vision system was once proposed for Companion. Based on the architecture, a Vision module would fit nicely in parallel with the Sonar and Laser modules. Naturally, the Mapper module would change to accommodate the new sensor, but the Search2d and Search3d modules' interface to the Mapper would not. As another example, a compass could be added as a new navigation sensor. Integration of the compass would require changes in the Cycle module, but to no other.

3.2 Implementation Models

QNX's interprocess communication features made it clear that each of the modules in the software architecture should be implemented as separate *processes*. Each process is a main program running on either the Tower or the Laptop that communicates via message passing to other processes. The models provide guidelines on how to implement a process with standard procedural execution and a standard interface. By developing all modules as processes following these models, a system consisting of uniform components is created. The hope is that such a system is easier to develop, debug, and learn.

there is a semantic clarification to make about the use of the terms “module” and “process.” A module, such as Cycle, is a component of the software architecture and exists independently of its implementation. Now, as discussion shifts to actual development in QNX, the Cycle process is discussed; this is the implementation of the Cycle module.

The models are based on the execution of commands. A process accepts a command (via message passing) from a parent process, executes it, and replies to the

parent (again via message passing). A process may have more than one parent. A complete system consists of a set of active processes, each commanding their child processes and executing commands from their parents. Two models for the modules have been developed: the *blocked process* and the *spinning process*. A blocked process performs no function until requested to do so. A spinning process continually performs its functions and periodically checks if a command has been sent to it. These models are discussed in detail below.

3.2.1 Blocked Process Model

A blocked process normally sits idle. When it receives a command, it executes it, and replies to the commanding process with data or an acknowledgment of the completion of the command. It is then idle once again. A flow chart for this type of process appears in Figure 3-2. The parent of a blocked process is not required to wait

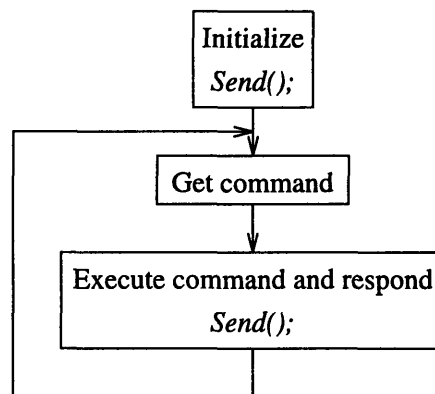


Figure 3-2: Flow Chart for Blocked Process Model

for the completion of the command before continuing execution. Instead, it can issue the command, go about its own business, and periodically check for the completion of the command. A flow chart for such a parent program is shown in shown in Figure 3-3.

The blocked process is tailored for tasks that take “a long time” to complete. A long time is really just the length time that one is unwilling to halt execution to wait for the completion of a command. In the Companion system, a long time is on the order of several milliseconds or longer. An example of a blocked process is the Sonar module. For an object 10 meters away, roughly 50 milliseconds elapse between a sonar ping and the return signal. Since this program uses the blocked model, a parent process can request a sonar reading, perform other useful tasks for 50 milliseconds, and then collect the range reading.

From Figure 3-2 and Figure 3-3, there is a reversed *Send()*, *Creceive()*, and *Reply()* sequence between the blocked process and its parent. The parent process “spawns” the blocked process and waits for a message from it. The blocked process initializes by sending a message to the parent and remains suspended until it gets a reply. The parent issues a command by replying to the blocked process. At this point, both

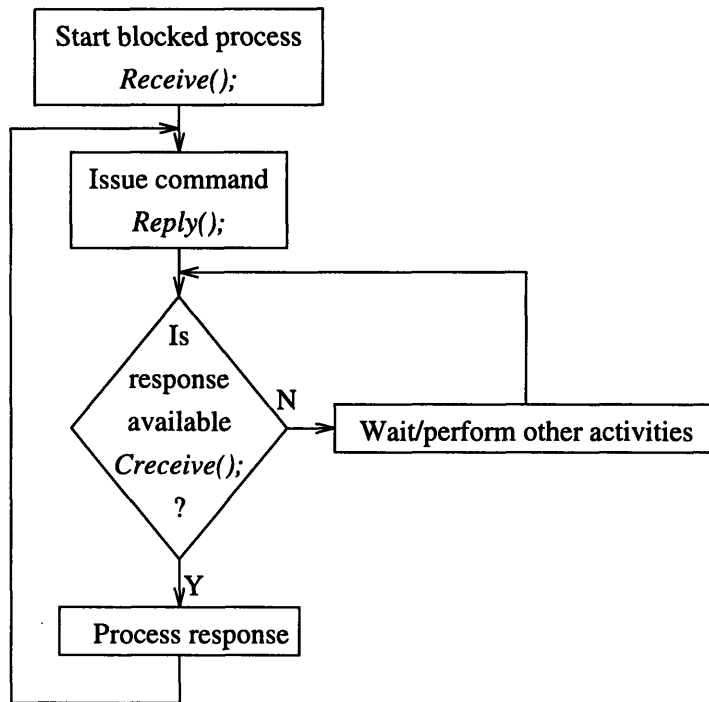


Figure 3-3: Flow Chart for Blocked Process Parent

processes are in the READY state. The child executes the command and when it is finished, it sends a message to the parent and repeats the cycle. The parent process continues execution and periodically checks if a response is available. When it is, the cycle is complete and the parent can issue another command with a *Reply()*.

3.2.2 Spinning Process Model

The spinning process model is simpler than the blocked model. Flow charts for this model and its parent appear in Figure 3-4 and Figure 3-5.

The spinning process continually cycles through an execution sequence. One step in the sequence is a *Creceive()* call that checks if a command is waiting. If one is, then it is immediately processed, and a reply sent. The spinning process then continues. A parent can issue a command to the spinning child but is blocked until the child replies. Thus, for this model to be efficient, the spinning process must execute in a fast loop, so that the parent does not have to wait long for the execution of a command.

An example of a spinning process module on Companion is the Cycle module, which is responsible for navigation. It is implemented as a spinning process so that parent processes can quickly obtain the most recent position information and so that Cycle can continually update the robot's position.

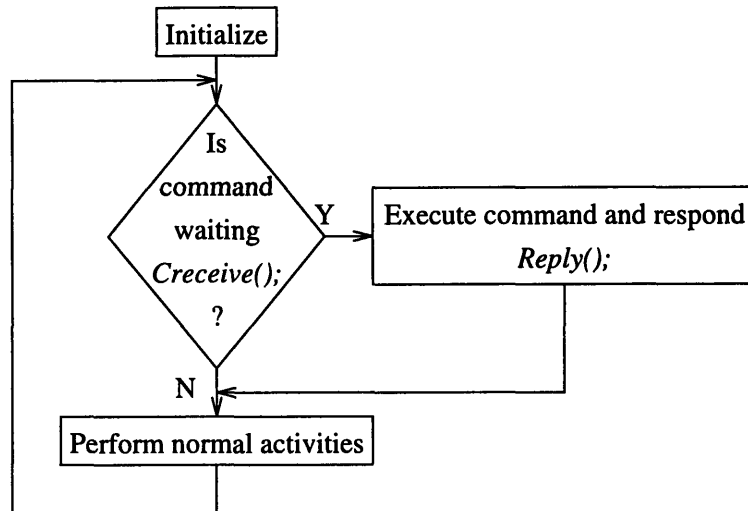


Figure 3-4: Flow Chart for Spinning Process Model

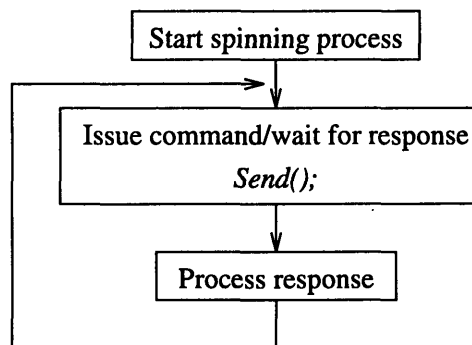


Figure 3-5: Flow Chart for Spinning Process Parent

3.3 Modules

This section discusses the modules in the software architecture in more detail. This includes a general description of their function and full sets of commands that each module is able to execute. By enumerating these commands, one can gain insight on how each modules is used in the overall software scheme. Table 3-1 is a summary of the

Module	Model	Computer	Timing Notes
Sound	spinning	Tower	spins at about 20 cycles per second
Cycle	spinning	Tower	spins at about 5 cycles per second
Sonar	blocked	Tower	takes about 0.025 seconds for a reading
Laser	blocked	Tower	takes about 2 seconds for a full theta sweep
Trajectory	spinning	Tower	spins at about 2 cycles per second
Mapper	spinning	Laptop	n/a
Search3d	blocked	Laptop	takes about 2 seconds for a search
Search2d	blocked	Laptop	takes about 1 second for a search
Planner	spinning	Laptop	n/a
User	n/a	Laptop	n/a

implementation of the modules on Companion. Note that although the Mapper and Planner modules are implemented as spinning processes, their spinning rates are not of particular significance because they do not really affect the systems' performance. Implementation and testing results for the Cycle, Trajectory, Search2d, Search3d, and Planner modules are provided in later chapters of this document. The User module is explained in Appendix B.

3.3.1 Sound Module

The Sound module, implemented as a spinning process, is the simplest of the Companion modules. It is responsible for managing the built-in speakers on the Tower and the Laptop. Each built-in speaker (one per computer) is capable of emitting a single tone at one frequency. The Laptop is node 1 on the network, and the Tower is node 2.

The Sound module is used primarily as a diagnostic and debugging tool. For example, as a safety feature, whenever the laser beam is powered, the Tower speaker is commanded to PULSE. Sound has also used during program testing and debugging to detect when certain blocks of code were executing. The final version of Companion's software system uses Sound only as a warning for the laser. It features the following commands:

STOP() Causes the Sound module to exit.

ON(n, p, oct) Turns on the speaker on node n at pitch p in octave oct .

OFF(n) Turns off the speaker on node n .

BEEP(n, p, oct, dur) Turns on the speaker on node n at pitch p in octave oct for a dur seconds, then off.

PULSE(n, p, oct, dur) Repeatedly toggles the speaker on node n at pitch p in octave oct at intervals of dur seconds.

3.3.2 Cycle Module

The Cycle module, also a spinning process, is the interface to actuators, navigation, and hazard sensors. Through this module, we can command the robot to move and turn. A parent module can also query the position of the robot and the state of its hazard sensors. Although the Cycle module monitors the hazard detectors, it takes no action when they are triggered—that is the responsibility of the Trajectory module (see Subsection 3.3.5).

The Cycle module internalizes the sensory information required to determine the robot's position. Companion happens to use a gyroscope and encoders for navigation, but if, say, it used GPS instead, the interface to Cycle would remain the same. From the perspective of a parent of the Cycle module, the means by which the position is determined is not relevant—at that level, the parent just wants to know where the robot is.

The Cycle module supports the following commands:

STOP() Causes the Cycle module to exit.

READ() Requests the current state of the robot: elapsed time since the beginning of the mission; the robot's position (x, y, θ); the state of the eight bumpers and eight proximity detectors; and the robot's current speed and curvature (s, ρ).

MOVE(s) Commands the robot to move at a speed s meters per second. (The value of s is positive for forward motion and negative for reverse motion.)

TURN(ρ) Commands the robot to turn at a radius $1/\rho$. (The value of ρ is positive for right turns and negative for left turns.)

FIXX($newx$) Changes the x position of the robot to $newx$.

FIXY($newy$) Changes the y position of the robot to $newy$.

FIXH($new\theta$) Changes the θ position of the robot to $new\theta$.

3.3.3 Sonar Module

The Sonar module, not unexpectedly, manages Companion's sonar array. A sonar reading consists of both a range measurement and the position of the robot when the reading was taken. Hence the Sonar module is a parent of the Cycle module. Sonar uses the Cycle READ() command to determine the robot's position. It is a blocked process and supports the following commands:

STOP() Causes the Sonar module to exit.

PING(x) Pings the x th sonar and responds with the range of the reading, the (x, y, θ) position of the robot during the ping, and the angle of the sonar with respect to the robot's chassis.

3.3.4 Laser Module

The Laser module, a blocked process, manages Companion's laser range finder and its associated positioning motors. Like Sonar, Laser uses the Cycle module to determine the position of the robot at read times. The Laser module is also a parent of the Sound module: as a safety feature, whenever the laser beam power is on, the the Laser module commands the Tower speaker to pulse. This module supports the following commands:

STOP() Causes the Laser module to exit.

POSITION() Requests the current position of the laser (θ, ϕ) , where θ is the yaw angle and ϕ is the pitch angle.

PHI_SWEEP($new\phi, n$) Sweeps the laser from its current position (θ, ϕ) to the position $(\theta, new\phi)$ while taking up to n readings. The Laser module responds with an array of range readings, the positions of the laser, and the positions of the robot at which the readings were taken.

THETA_SWEEP($new\theta, n$) Sweeps the laser from its current position (θ, ϕ) to the position $(new\theta, \phi)$ while taking up to n readings. The Laser module responds with an array of range readings, the positions of the laser, and the positions of the robot at which the readings were taken.

READ() Takes a single reading of the laser at its current position. The module responds with the range reading, the position of the laser, and the position of the robot at which the reading was taken.

HOME() Moves the laser to its "home" position $(0, \pi/2)$.

BEAM_POWER($flag$) Turns on or off the laser beam depending on the value of $flag$. Naturally, readings can only be taken when the beam is on.

MOTOR_POWER($flag$) Turns on or off the position controller for the laser motors depending on the value of $flag$. The laser will not move unless the motor power is on.

3.3.5 Trajectory Module

Companion's Trajectory module provides powerful commands that allow Companion to perform line following and circle following. Naturally, it must issue commands to Cycle in order for the robot to move. Trajectory can accept a set of lines and circles and execute them sequentially. Trajectory also protects the robot by halting its motion when hazards are detected and has provisions for escaping from hazard-triggered states.¹ Commands supported by the Trajectory module are:

STOP() Causes the Trajectory module to exit.

READ() Requests the status of a line or circle following command. Returned parameters include the robot's off line distance, the distance remaining on the command, flags for hazard detection, and the number of commands yet to be executed.

FOLLOWLINE(x, y, θ, dir) Adds to the queue a command that causes the vehicle to follow the line passing through (x, y) with direction θ . The sign of dir specifies whether robot should move forwards or backwards in following the line. The robot stops when it passes the point (x, y) .

FOLLOWCIRCLE(x, y, r, θ, dir) Adds to the queue a command that causes the vehicle to follow the circle with center (x, y) and radius r . The sign of dir determines whether the robot moves forwards or backwards. The robot stops when its heading is θ .

CANCEL() Clears the queue of commands

HALT() Temporarily stops the robot from moving without disturbing the queue.

CONTINUE() Resumes execution of commands disrupted by a HALT command.

3.3.6 Mapper Module

The Mapper module, the main topic of [16], is implemented as a spinning process. It is the parent of both the Laser and the Sonar processes. Its responsibility is to generate a representation of the robot's environment. It supports the following commands:

STOP() Causes the Mapper module to exit.

FREEZE($flag$) Depending on the value of $flag$, causes the Mapper to halt or resume updating of the map. This is useful because it allows the search modules to perform their function on a static map. When halted, the Mapper may continue to collect sensor readings but cannot incorporate them into the map.

SENSE($flag$) Depending on the value of $flag$, causes the Mapper to halt or resume sensing of the environment.

¹not yet implemented

3.3.7 Search3d Module

The Search3d module, a blocked process, is a high-resolution search routine. It generates paths that consider the robot's environment and its kinematic constraint. It is really an extension of the Planner module, but implemented separately because of its special functionality. It has only two commands:

STOP() Causes the Search3d module to exit.

SEARCH($x_s, y_s, \theta_s, x_g, y_g, \theta_g$) Generates a sequence of commands compatible with the Trajectory module for moving the robot from (x_s, y_s, θ_s) to (x_g, y_g, θ_g) .

3.3.8 Search2d Module

The Search2d module, also a blocked process, is a low resolution search routine. It generates coarse paths consisting of one or more way points and ignores the robot's kinematic constraint. Each way point is an intermediate position on the path specified by x and y coordinates. It, like Search3d, is an extension of the Planner module. Its two commands are:

STOP() Causes the Search2d module to exit.

SEARCH(x_s, y_s, x_g, y_g) Generates a sequence of way points for moving the robot from (x_s, y_s) to (x_g, y_g) .

3.3.9 Planner Module

The Planner module, implemented as a spinning module, is the coordinator of path planning and plan execution. In the future, it may support a variety of missions. At this time, it only supports a way point command that moves the robot to the point (x_g, y_g, θ_g) . Its commands are:

STOP() Causes the Planner module to exit.

WAYPT(x_g, y_g, θ_g) Causes the Planner to begin a mission to move the robot to the goal point (x_g, y_g, θ_g) (using the Search3d, Search2d, and Trajectory modules).

HALT() Causes the Planner to give up on the mission and stop the robot.

READ() Requests that the Planner provide information regarding the status of the mission.

3.3.10 User Module

The User module is the top level of Companion's software. It is not a true module because it has no parents, and hence cannot accept commands. Instead, it is the parent of the Planner module. Through the user interface in this module, Companion's operator can specify a way point or abort a mission.

Chapter 4

Implementation and Testing of the Cycle Module

The Cycle module has three responsibilities: hazard detection, motor actuation, and navigation. Hazard detection serves as the last line of defense against collisions with obstacles. Motor actuation provides an interface to the robot's drive and steering motors. Navigation is the monitoring of the robot's position. These three functions, as implemented for Companion, are described in this chapter.

4.1 Hazard Detection

Cycle's hazard detection consists of reading Companion's bumpers and proximity detectors. When these sensors are triggered, an obstacle has violated the robot's safety radius. In the Cycle process, this hazard detection is the simple matter of regularly reading all bumpers and proximity detectors (via the digital I/O board on the Tower).

4.2 Motor Actuation

Cycle controls Companion's drive motor and steer motor. To provide a convenient interface to external programs, the Cycle module allows the drive and steer motors to be commanded with speed and curvature (the inverse of turning radius) arguments, respectively. This interface is convenient for other modules, such as Trajectory, and portable to other robots.

As described in Section 2.1, Companion's mobility platform originated as an electric wheelchair. Cycle's interface to the motors are D/A lines, each having a range of 0 to 4095 with 8 bits of resolution. The wheelchair's control system uses these analog signals as a reference for its own closed loop control over the drive speed and steering angle. The task here is to determine the appropriate digital values to make the robot move at the desired speed and curvature.

4.2.1 Drive Motor Actuation

It turns out that fine control over the robot's speed is not necessary—for safety reasons, all testing was done with the robot moving at very slow speeds. Consequently, no time was spent developing a real controller for speed. However, the digital value which resulted in a speed of zero was found. Values greater than that give forward motion, and smaller values give reverse motion. The Cycle module has five speeds: fast reverse, slow reverse, stop, slow forward, and fast forward. These speeds are not calibrated but instead represent what can be considered reasonable. They are slow enough that there is no fear of a runaway robot, but fast enough that patience is not lost.

4.2.2 Steering Motor Actuation

To command the robot's curvature, the desired curvature (ρ) is first mapped to the equivalent angles of the front wheels. This mapping is based on a no-slip model of motion. The vehicle is modeled as rectangle, with a wheel a fixed distance, D , from each vertex. The rear wheels are aligned with the robot chassis and cannot be steered. The front wheels may be steered, but only such that the projections of all four wheels' axes intersect a common point (see Figure 4-1). The front wheels rotate

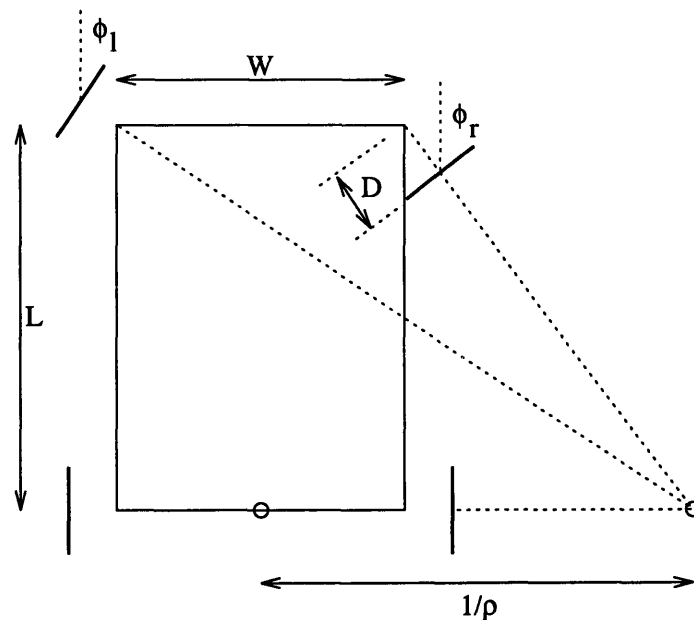


Figure 4-1: Steering Model

about one vertex of the rectangle, but the effective contact point with the ground is offset from the vertex by the distance D . The axle-to-axle length of the robot is L , and the width of the robot is W . With this model, the instantaneous direction of travel is perpendicular to the rear axle, and no wheel is ever slipping.

The curvature command uses the center of the rear axle as the reference point of the curvature, as shown in Figure 4-1. In other words, this point ideally travels in the path of a circle with radius $1/\rho$. The angle that each of the front wheels (ϕ_r and ϕ_l) should have to achieve the desired curvature are given by:

$$\phi_r = \tan^{-1} \left(\frac{\rho L}{1 - \frac{1}{2}W\rho} \right) \quad (4.1)$$

$$\phi_l = \tan^{-1} \left(\frac{\rho L}{1 + \frac{1}{2}W\rho} \right) \quad (4.2)$$

These are the desired steering angles for the front two wheels. Note that for a straight line, $\rho = 0$ and the equations are still well-behaved.¹

To move the steering wheels to the desired angle, a lookup table is used. The lookup table maps the desired steering angle to the digital value that causes the wheelchair control system to maintain position control at that angle. The lookup table was generated once and stored as a static array in the Cycle program. It was known that the lookup table would provide only marginal performance, but it was an easy way to get the robot up and running.

The performance of the lookup table was tested by commanding curvatures and observing the actual turning radius of the robot. For the rest of this section, commands are given as radius commands, which are easier to interpret. Testing involved left and right turns with radii ranging from 0.5 to 1.5 meters (negative radius for left turns), as shown in Table 4-1.

Desired Radius meters (1/meters)	Radius Commanded meters (1/meters)	Actual Radius meters (1/meters)	Error
-1.500 (-0.667)	-1.504 (-0.665)	-1.613 (-0.620)	7.53% (7.00%)
-1.250 (-0.800)	-1.250 (-0.800)	-1.350 (-0.741)	8.00% (7.38%)
-1.000 (-1.000)	-1.006 (-0.994)	-1.073 (-0.932)	7.30% (6.80%)
-0.750 (-1.333)	-0.760 (-1.316)	-0.823 (-1.216)	9.73% (8.80%)
-0.500 (-2.000)	-0.501 (-1.998)	-0.585 (-1.709)	17.00% (14.55%)
0.500 (2.000)	0.546 (1.830)	0.583 (1.717)	16.60% (14.15%)
0.750 (1.333)	0.776 (1.288)	0.805 (1.242)	7.33% (6.85%)
1.000 (1.000)	1.022 (0.978)	1.068 (0.937)	6.80% (6.30%)
1.250 (0.800)	1.287 (0.777)	1.338 (0.748)	7.04% (6.50%)
1.500 (0.667)	1.572 (0.636)	1.655 (0.604)	10.30% (9.40%)

In each test, a turning radius was selected (first column of table). The lookup

¹Using curvature instead of turning radius avoids the numerical problem of specifying an infinite turning radius when the robot is commanded to move straight.

table selects a digital value, sends it to the steering motor controller, and the wheels move. The inverse lookup table then takes readings from the steering potentiometers and predicts the radius that the robot will actually move (second column). The robot then actually goes in a circle. The radius, as physically measured, is shown in column three. Finally, the computed error between the measured and commanded radii is shown in column four. The average error was less than 10%—not particularly good, but adequate.

4.3 Navigation

Navigation, the monitoring of Companion's x , y , and θ positions, can be accomplished in many ways, depending on the availability and accuracy of sensors. Some sensors, such as GPS and compasses, give reading relative to the earth. Other sensors, such as encoders and gyroscopes, provide readings relative to a starting position. Companion only has sensors of the latter type. Hence, Cycle's navigation is done through dead reckoning—the robot's position is found by integrating readings from sensors and is known relative to a starting position.

This section first presents the geometry required to determine the motion of the robot from the available sensors. Results of testing with this dead reckoning system are then shown.

4.3.1 Dead Reckoning Equations

Companion's position is represented by the triple (x, y, θ) . The robot position is recorded in a coordinate frame fixed relative to the ground. Typically, this frame is oriented such that the robot begins its mission at the origin.

Dead reckoning is accomplished using three sensors. The integrated gyroscope reading gives the heading of the robot. Thus, the θ position of the robot is simply a matter of reading the gyroscope. The other navigation sensors are wheel encoders, one attached to each of the two front wheels. The wheels encoders give the distance traveled by these wheels. The dead reckoning scheme resolves the measured heading and wheel motions to the motion of the center of the robot. The motion is based on the steering model described in Subsection 4.2.2.

The strategy for dead reckoning is to discretize time into cycles (lasting from about 0.01 to 0.1 seconds). At the end of each cycle, the program notes how much the heading of the robot has changed since the beginning of the cycle, and it reads the encoders to determine how far (and in which direction) each of the front wheels has moved during the cycle. It is assumed that the steering angle of the robot was fixed during that time; hence the path of the robot is an arc of a circle (if the heading has not changed, the arc degenerates to a straight line). The dead reckoning scheme described here can be divided into three steps. First, it determines the radius about which the center-rear of the robot has traveled. Second, based on the heading change, it computes the displacement of the center-rear of the robot (relative to the robot position at the beginning of the cycle). Finally, the motion of the center-rear is transformed to the

motion of the center of the robot in global coordinates. The scheme yields a new position for the robot at the i th cycle:

$$\theta_i = \theta_i \quad (4.3)$$

$$x_i = x_{i-1} + \Delta x \quad (4.4)$$

$$y_i = y_{i-1} + \Delta y \quad (4.5)$$

During a cycle, as shown in Figure 4-2, the robot has moved forward and turned

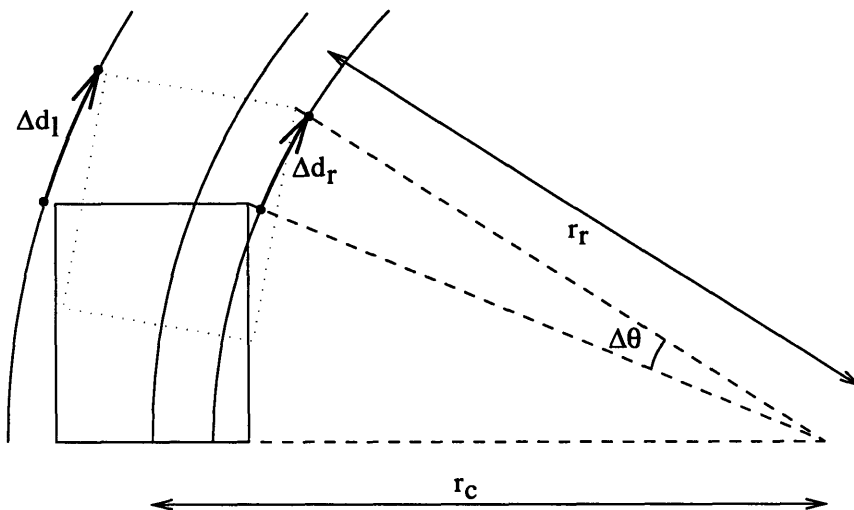


Figure 4-2: Dead Reckoning: Motion During a Cycle

clockwise. In doing so, the robot's gyroscope measures a heading change of $\Delta\theta$; the left and right encoders measure distances Δd_l and Δd_r , respectively.² The figure also shows r_c , the radius traveled by the center-rear of the robot and r_r , the radius traveled by the right-front corner of the robot. To avoid cluttering the figure, the radius of the left-front corner, r_l , is not shown. Note that due to the wheel offset, r_r is slightly larger than the radius traveled by the right-front wheel on a right turn, but smaller on a left turn.³ On the other hand, r_l is smaller than the radius traversed by the left-front wheel for right turns, and larger for left turns. The difference in the radii is the size of wheel offset D shown in Figure 4-1.

Two values for r_c are now computed, one from each of the front encoders in combination with the heading change. They are r_{cl} and r_{cr} . The two values are

²To determine the heading change, the heading as given by the gyroscope at the beginning of the i th cycle, θ_{i-1} , is subtracted from the heading at the end of the i th cycle, θ_i , i.e.:

$$\Delta\theta = \theta_i - \theta_{i-1}$$

Similarly, encoder distances are the distances traveled during the i th cycle.

³A right turn is one in which the center of the circle is to the right of the robot. A left turn has the center on the left.

averaged as the mechanism for smoothing the readings. Note first that:

$$r_r = \frac{\Delta d_r}{\Delta \theta} + D \quad (4.6)$$

$$r_l = \frac{\Delta d_l}{\Delta \theta} - D \quad (4.7)$$

where r_r and r_l are positive for right turns and negative for left turns. From the geometry of Figure 4-2:

$$r_{cr} = \pm \left(\sqrt{r_r^2 - L^2} + \frac{1}{2}W \right) \quad (4.8)$$

$$r_{cl} = \pm \left(\sqrt{r_l^2 - L^2} - \frac{1}{2}W \right) \quad (4.9)$$

where again the left turn/right turn sign convention is used. Substituting for r_r and r_l , r_{cl} and r_{cr} can be rewritten:

$$r_{cr} = \frac{\Delta d_r}{\Delta \theta} \left(\sqrt{\left(1 + \frac{D\Delta \theta}{\Delta d_r}\right)^2 - \left(\frac{L\Delta \theta}{\Delta d_r}\right)^2} + \frac{W\Delta \theta}{2\Delta d_r} \right) \quad (4.10)$$

$$r_{cl} = \frac{\Delta d_l}{\Delta \theta} \left(\sqrt{\left(1 - \frac{D\Delta \theta}{\Delta d_l}\right)^2 - \left(\frac{L\Delta \theta}{\Delta d_l}\right)^2} - \frac{W\Delta \theta}{2\Delta d_l} \right) \quad (4.11)$$

This form preserves the turn direction sign convention. Once the two are averaged, the radius about which the center-rear of the robot is rotating can be obtained:

$$r_c = \frac{1}{\Delta \theta} \frac{1}{2} \left(\Delta d_l \sqrt{\left(1 - \frac{D\Delta \theta}{\Delta d_l}\right)^2 - \left(\frac{L\Delta \theta}{\Delta d_l}\right)^2} + \Delta d_r \sqrt{\left(1 + \frac{D\Delta \theta}{\Delta d_r}\right)^2 - \left(\frac{L\Delta \theta}{\Delta d_r}\right)^2} \right) \quad (4.12)$$

With r_c , the displacement of the center-rear during the cycle relative to the position of the robot at the beginning of the cycle can be determined. From Figure 4-3, the center axis of the robot is tangent to the circle of radius r_c at the center-rear of the robot. The change in position of the rear-center of the robot is:

$$\Delta x_c = r_c \sin \Delta \theta = (r_c \Delta \theta) \left(\frac{\sin \Delta \theta}{\Delta \theta} \right) \quad (4.13)$$

$$\Delta y_c = r_c (1 - \cos \Delta \theta) = (r_c \Delta \theta) \left(\frac{1 - \cos \Delta \theta}{\Delta \theta} \right) \quad (4.14)$$

A singularity occurs when $\Delta \theta = 0$. As $\Delta \theta$ approaches 0, Δx_c approaches $\frac{1}{2}(\Delta d_l + \Delta d_r)$ and Δy_c approaches 0, as expected. The expression $(r_c \Delta \theta)$ is well behaved for all $\Delta \theta$, since the equation for r_c already contains $\Delta \theta$ in the denominator. To perform dead reckoning numerically, $\Delta \theta$ is assumed to be small, and the ill-behaved

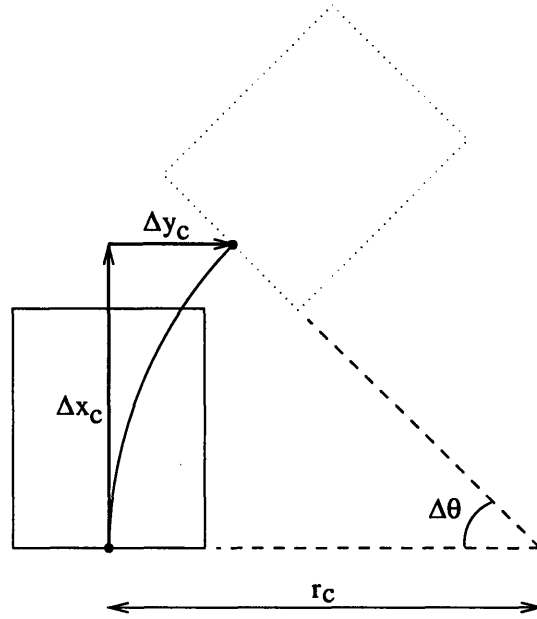


Figure 4-3: Dead Reckoning: Center-Rear Motion

terms are replaced by the first few terms of their series expansions:

$$\Delta x_c \approx (r_c \Delta \theta) \left(1 - \frac{(\Delta \theta)^2}{6} + \frac{(\Delta \theta)^4}{120} \right) \quad (4.15)$$

$$\Delta y_c \approx (r_c \Delta \theta) \left(\frac{\Delta \theta}{2} - \frac{(\Delta \theta)^3}{24} + \frac{(\Delta \theta)^5}{720} \right) \quad (4.16)$$

A problem also arises if the quantities under the square root are negative. If this is the case, assumptions about the robot's geometry and no-slip motion have been violated, and dead reckoning is not performed in this case.

The final step of the dead reckoning scheme is to transform the motion of the center-rear in robot coordinates to the motion of the center of the robot in global coordinates. In Figure 4-4, the problem can be seen vectorially. The motion of the center of the robot is a matter of adding four vectors together:

$$\Delta x = -\frac{L}{2} \cos \theta_{i-1} - \Delta y_c \sin \theta_{i-1} + \Delta x_c \cos \theta_{i-1} + \frac{L}{2} \cos \theta_i \quad (4.17)$$

$$\Delta y = -\frac{L}{2} \sin \theta_{i-1} + \Delta y_c \cos \theta_{i-1} + \Delta x_c \sin \theta_{i-1} + \frac{L}{2} \sin \theta_i \quad (4.18)$$

The formulation of the dead reckoning scheme is now complete:

$$\theta_i = \theta_i \quad (4.19)$$

$$x_i = x_{i-1} + \frac{L}{2} (\cos \theta_i - \cos \theta_{i-1}) + \Delta x_c \cos \theta_{i-1} - \Delta y_c \sin \theta_{i-1} \quad (4.20)$$

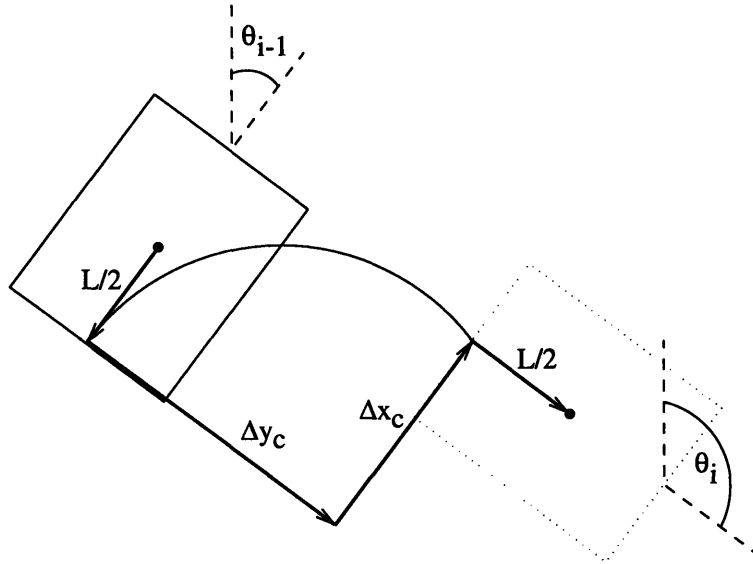


Figure 4-4: Dead Reckoning: Robot Center Motion

$$y_i = y_{i-1} + \frac{L}{2}(\sin \theta_i - \sin \theta_{i-1}) + \Delta x_c \sin \theta_{i-1} + \Delta y_c \cos \theta_{i-1} \quad (4.21)$$

4.3.2 Testing of Dead Reckoning

Among the most interesting of all tests on Companion were for its dead reckoning. Figure 4-5 shows the paths of four long distance dead reckoning tests and Table 4-2 summarizes the results. In each case the robot was required to move at least 85 meters

Path Length (meters)	Final Position Error (meters)	Error
94.12	1.05	1.12%
99.52	0.08	0.08%
216.18	1.55	0.72%
85.40	0.26	0.30%

in a path that returned it to its starting location.⁴ Of the four paths shown, only the bottom right sample had the robot moving in reverse. The dead reckoning performed well: in all test runs, the error in dead reckoning was less than 1.2% of the length of the path.

Tests that required the robot to make sharp turns and changes in direction were also conducted. Four of the paths are shown in Figure 4-6 and performance is sum-

⁴It is not a coincidence that the paths trace out the hallways of the Draper Laboratory.

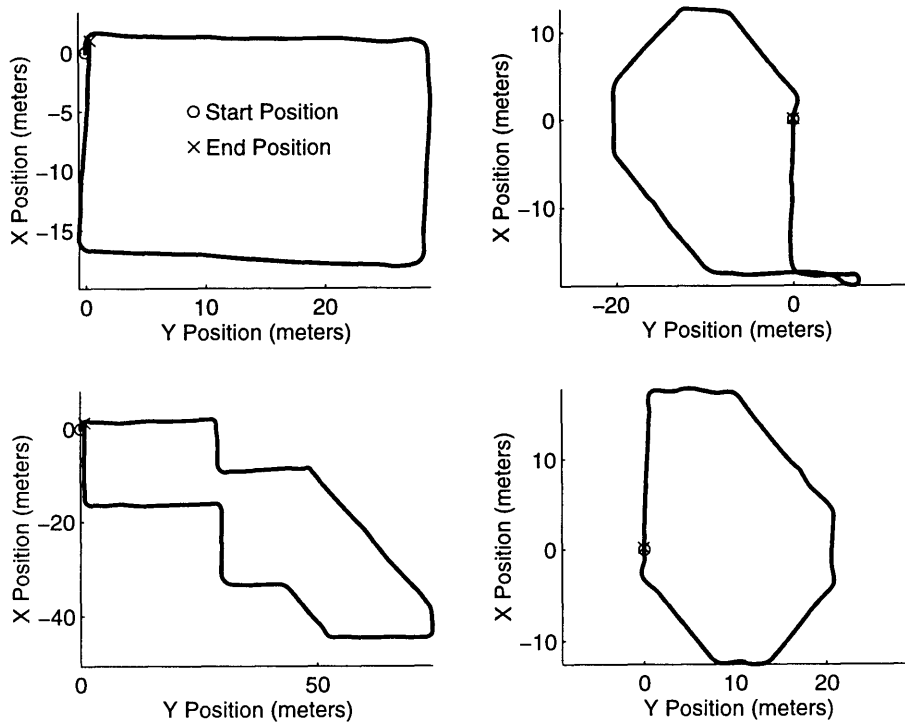


Figure 4-5: Long Distance Testing of Dead Reckoning

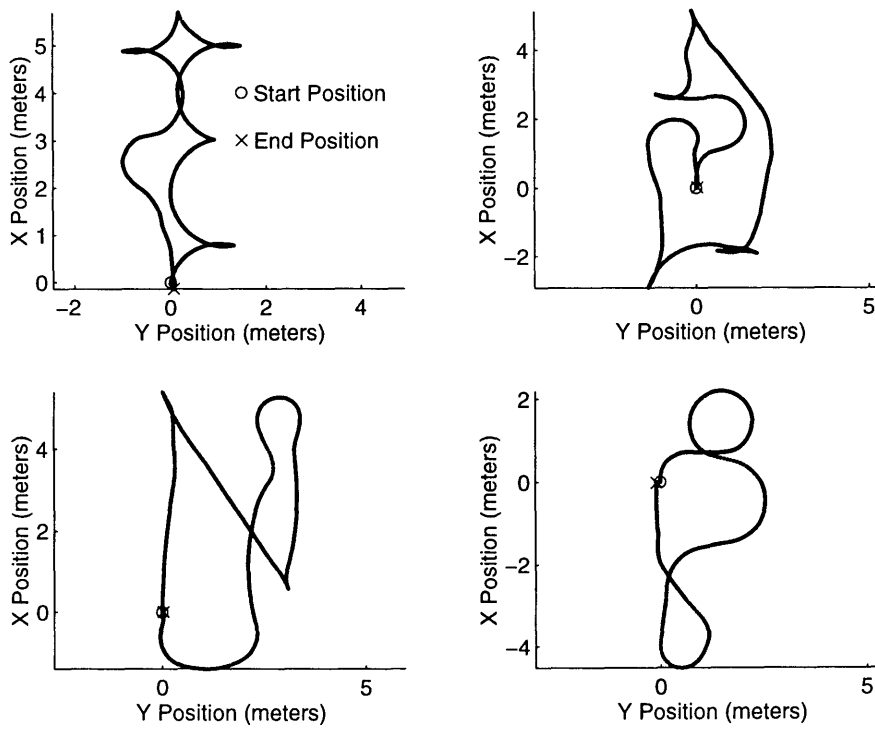


Figure 4-6: Sharp Turning Testing of Dead Reckoning

marized in Table 4-3. Again, the final position was the same as the initial position.

Path Length (meters)	Final Position Error (meters)	Error
18.59	0.11	0.59%
30.36	0.09	0.30%
26.36	0.11	0.42%
19.44	0.05	0.26%

The robot performed well on these tests as well. The largest dead reckoning error was 0.59% of the length of the path. This quality of navigation was fine for this robot.

Chapter 5

Implementation and Testing of the Trajectory Module

The Trajectory module's main function is to provide the correct sequence of speed and curvature commands to the Cycle module so that the robot follows a specified line or circle. In addition, the Trajectory module manages sequences of lines and circles to follow. The idea is that one can input a series of lines and circles and have the Trajectory module execute them one-by-one.

With Companion's system architecture, the initial position of the robot is assumed to be near the line or circle to be followed (within a couple of centimeters and a fraction of a radian). This provides an important advantage in developing the Trajectory module: initial conditions far from the desired path are not critical. However, providing smooth transitions between commands is an issue.

This chapter presents the controller used for circle following and its special use as a line following controller. Plots of testing with different parameters are shown, along with measures the quality of the robot's performance. Finally, results of multi-command trajectories are shown.

5.1 Circle Following

This section contains the derivation of the proportional controller used in circle following. It also explains the stopping condition and shows results of experiments run with the controller.

5.1.1 Circle Following Controller Derivation

In circle following, the robot is to trace a trajectory along the circumference of a circle. As shown in Figure 5-1, the circle is specified by its center (x_g, y_g) and radius r . The robot, located at (x, y, θ) , should stop when its heading reaches θ_g ; for convenience, the center and stopping heading are discussed together as (x_g, y_g, θ_g) . In the figure, the radial vector is $(x_g, y_g, \theta_g - \pi/2)$ because the robot's heading is tangent to the circle. Using a translation and a rotation, the robot is mapped to $(0,0,0)$ and

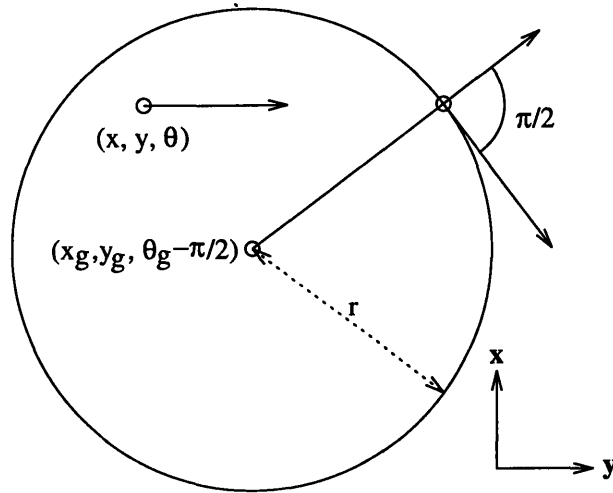


Figure 5-1: Geometry of Circle Following

the circle's center relative to it:

$$\begin{bmatrix} x_r \\ y_r \\ \theta_r \end{bmatrix} = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_g - x \\ y_g - y \\ \theta_g - \theta \end{bmatrix} \quad (5.1)$$

Without the loss of generality, the robot at $(0, 0, 0)$, and the desired circle is of radius r with center (x_r, y_r, θ_r) , as shown in Figure 5-2.

A simple proportional controller for circle following is now devised. From Figure 5-2, the point (x_t, y_t, θ_t) is a distance d_t (along the perimeter of the circle) from the robot's projection onto the circle. The heading θ_t is the heading from the position of the robot directly to (x_t, y_t) . The measure of error θ_e is the difference between the heading of the robot and θ_t . Since the robot's heading is identically zero:

$$\theta_e = \theta_t \quad (5.2)$$

The commanded curvature ρ_{cmd} is proportional to the heading error:

$$\rho_{cmd} = K_p \theta_e \quad (5.3)$$

From the geometry of this configuration intermediate quantities are computed:

$$\phi = \tan^{-1} \left(\frac{0 - y_r}{0 - x_r} \right) + \frac{d_t}{r} = \tan^{-1} \left(\frac{-y_r}{-x_r} \right) + \frac{d_t}{r} \quad (5.4)$$

$$x_t = x_r + r \cos \phi \quad (5.5)$$

$$y_t = y_r + r \sin \phi \quad (5.6)$$

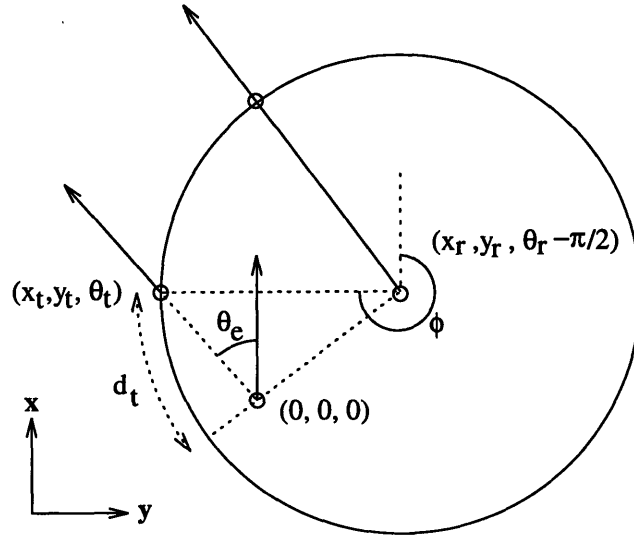


Figure 5-2: Transformed Geometry of Circle Following

$$\theta_t = \tan^{-1} \left(\frac{y_t - 0}{x_t - 0} \right) = \tan^{-1} \left(\frac{y_t}{x_t} \right) \quad (5.7)$$

Using all these equations, ρ_{cmd} is expressed in terms of known quantities and the two parameters K_p and d_t :

$$\rho_{cmd} = K_p \tan^{-1} \left(\frac{y_r + r \sin \left[\tan^{-1} \left(\frac{-y_r}{-x_r} \right) + \frac{d_t}{r} \right]}{x_r + r \cos \left[\tan^{-1} \left(\frac{-y_r}{-x_r} \right) + \frac{d_t}{r} \right]} \right) \quad (5.8)$$

A special situation, when the robot is exactly on line and headed in the right direction, as in Figure 5-3, is the case where:

$$\theta_e = \frac{d_t}{r} \quad (5.9)$$

Since the robot is on the circle with the right heading, it is commanded the very curvature that gives us the radius r :

$$\rho_{cmd} = \frac{1}{r} \quad (5.10)$$

Combining Equations 5.3, 5.9, and 5.10:

$$\rho_{cmd} = K_p \theta_e = K_p \frac{d_t}{r} = \frac{1}{r} \quad (5.11)$$

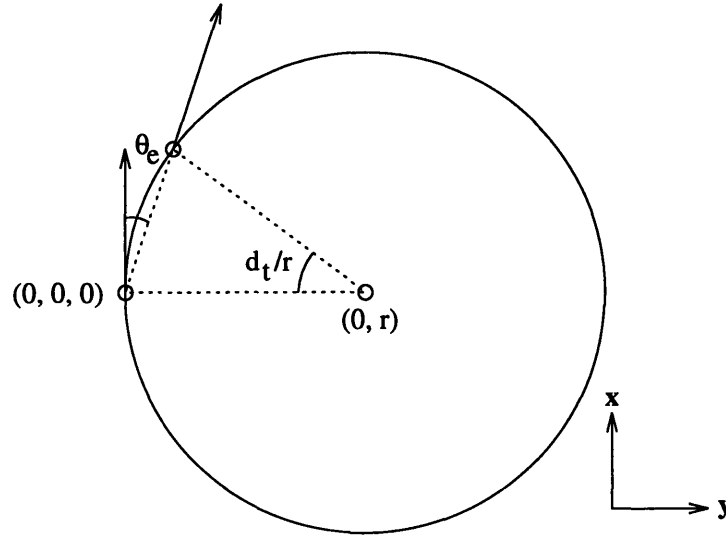


Figure 5-3: Circle Following—No Error Case

and solving for K_p :

$$K_p = \frac{1}{d_t} \quad (5.12)$$

So, in general, the commanded curvature is:

$$\rho_{cmd} = \frac{1}{d_t} \tan^{-1} \left(\frac{y_r + r \sin \left[\tan^{-1} \left(\frac{-y_r}{-x_r} \right) + \frac{d_t}{r} \right]}{x_r + r \cos \left[\tan^{-1} \left(\frac{-y_r}{-x_r} \right) + \frac{d_t}{r} \right]} \right) \quad (5.13)$$

and a single parameter d_t must be selected.

5.1.2 Stopping Condition

The robot is instructed to stop when its heading (θ) approaches the stopping heading (θ_g). In the transformed coordinate system, the stopping heading is θ_r and the robot's heading is 0. Hence, as θ_r nears 0, the robot is instructed to stop. Another useful piece of data is derived here—the distance left to travel (d_{togo}) is given by:

$$d_{togo} = r\theta_r \quad (5.14)$$

In Section 5.3, d_{togo} is used to execute smooth transitions between commands.

5.1.3 Parameter Selection and Testing

Figure 5-4 and Figure 5-5 show test results for d_t equal to 0.25 meters and 0.50 meters. For each of the values of d_t , trajectories starting with different initial off

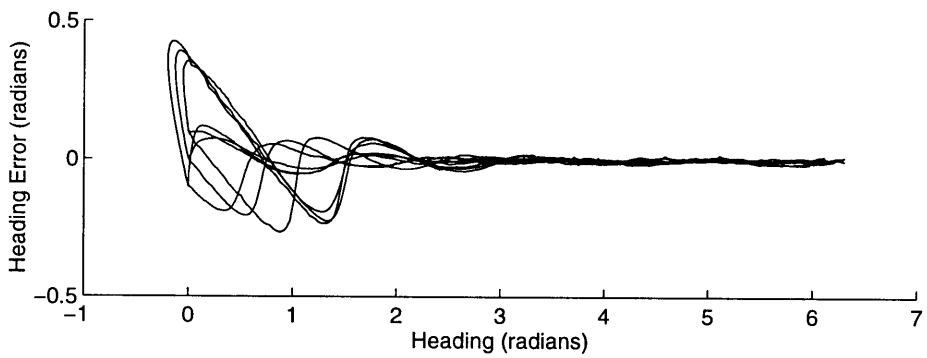
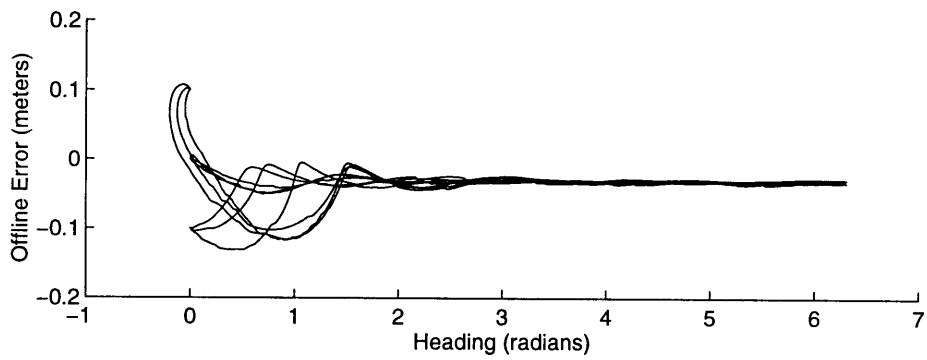


Figure 5-4: Circle Following Tests, $d_t = 0.25$

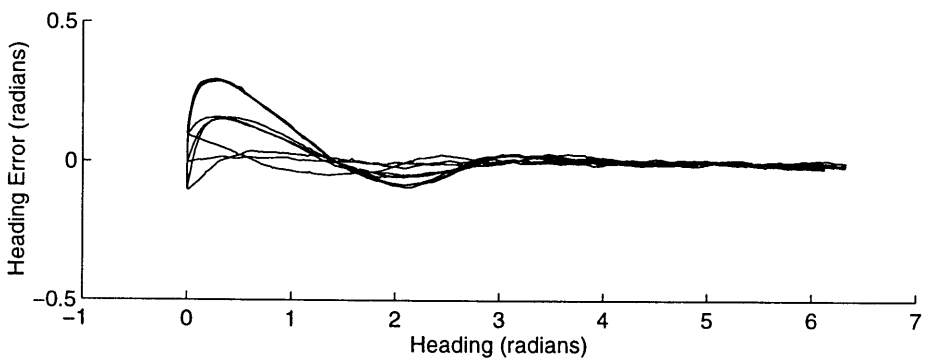
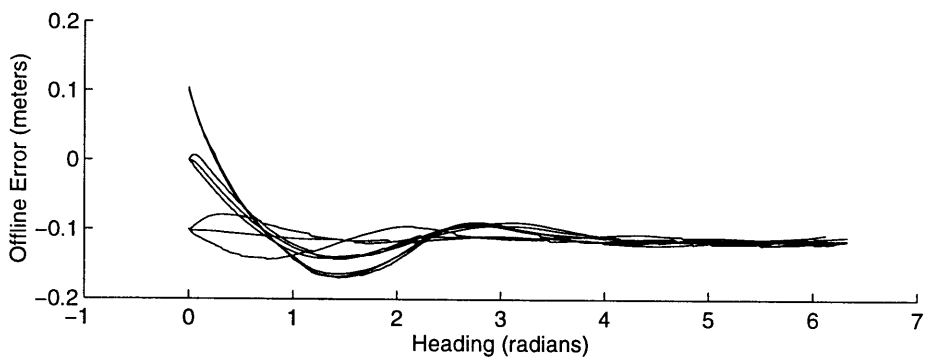


Figure 5-5: Circle Following Tests, $d_t = 0.50$

line and heading errors were recorded. With d_t at 0.25 meters, the robot performed acceptably, with an off line steady state bias less than 0.03 meters, no bias on heading, and a reasonably fast settling time.

5.2 Line Following

Line following is interpreted as a special case of circle following, in particular, when the radius is very large. Hence the circle following controller is also used for line following. The Companion implementation treats line following as circle following with radius 1000 meters. Test results showed similar results as in the case of circle following.

5.3 Multiple Command Execution

The final important task of the Trajectory module is to sequentially execute a series of commands in an open loop fashion. That is, a set of commands is issued to Trajectory to be executed sequentially without updating the commands to compensate for line/circle following errors. That means that the accumulation of off line and heading error should not render later commands impossible to complete. To demonstrate this capability, a series of tests were conducted. In Figure 5-6, the robot executed a

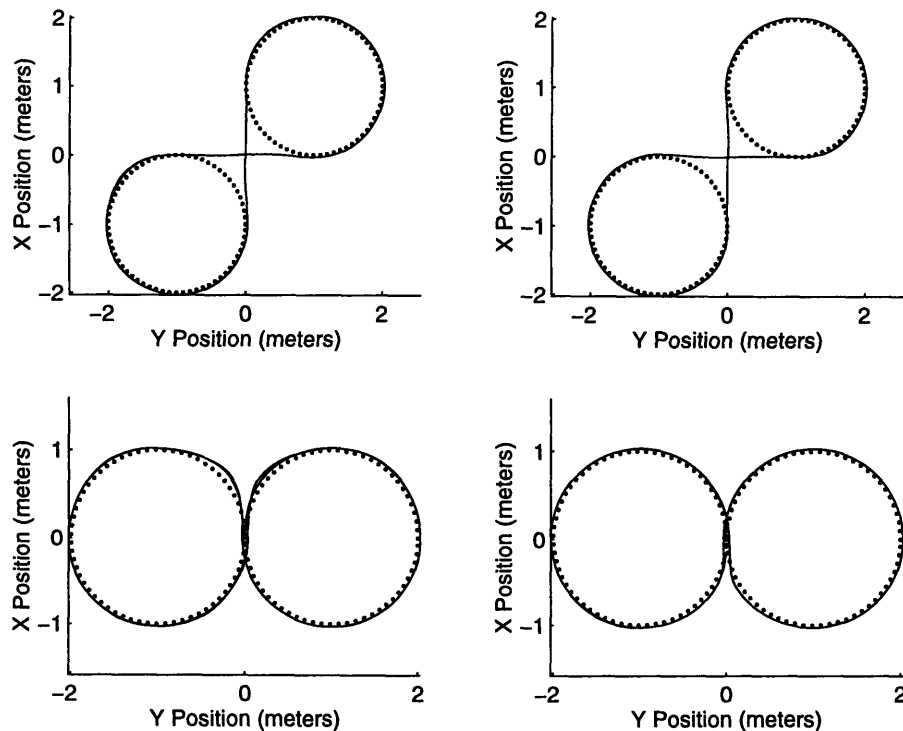


Figure 5-6: Multi-Command Trajectories

set of figure eight trajectories. In each case, the robot was commanded to traverse the path twice, all from commands issued at the beginning. The left two graphs are plots of forward motion, and the right two plots show backwards motion. In all cases, the robot performed well, staying within about 0.06 meters of the desired trajectory, shown as dotted circles in the plots. Figure 5-7 shows a particularly difficult trajectory

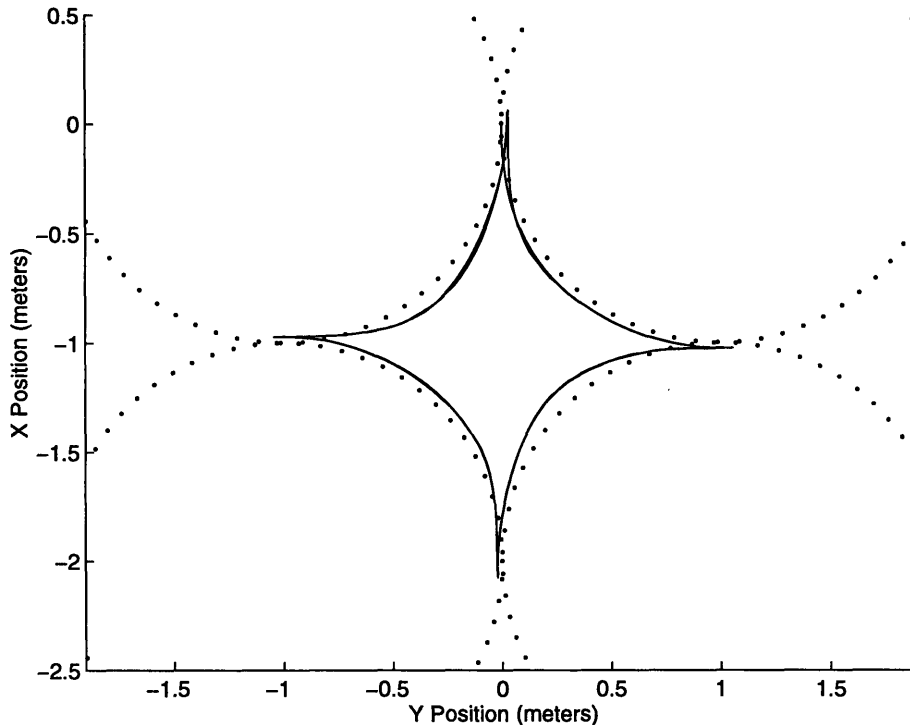


Figure 5-7: Multi-Command Cusp Trajectory

pattern—several cusps. In this pattern the robot must reverse directions and change turning directions. Again, the robot performed well.

To allow for the smooth transition between commands, a set of rules for command completion were implemented. Suppose the robot is executing a command that requires forward motion. If the next command to be executed also requires forward motion, the current command is considered complete 0.3 meters¹ before the actual goal point for that command, i.e. when d_{togo} is less than 0.3 meters. When this condition is met, Trajectory begins executing the next command. This permits a smooth transition between commands. If, however, the next command requires a change in forward/backward motion, the current command is executed until the robot is within 0.05 meters of the goal point. The multi-command trajectories in Figure 5-6 and Figure 5-7 are based on these command-transition rules.

¹The distance 0.3 meters was selected empirically by observing the robot's performance with different distances.

Chapter 6

Path Planning: Background and Overview

Although path planning is a well explored topic, Companion required its own custom system to meet its performance requirements and for compatibility with other module implementations. This chapter presents requirements for Companion's planning system and a survey of some existing path planning strategies with references to others. These systems have advantages and disadvantages, and Companion's path planner combines some of them into a practical system. As a prelude to the descriptions of the Search3d and Search2d modules, an overview of the path planner is also included in this chapter.

6.1 Requirements

There were several requirements for the path planning system to be developed for Companion. The first was that the path planner incorporate the robot's environment. The system had to be general enough to devise paths through an array of environments, some as tightly constrained as indoor hallways. In addition, for implementation purposes, the path planner had to be compatible with the Mapper module's representation of the environment. For reasons described in [16], Companion's Mapper module was implemented as a probability grid with resolution 0.015 meters per pixel. The map also had resolutions of 0.15 meters per pixel and 1.5 meters per pixel. This representation is relevant because individual obstacles were not represented as single entities in the map; instead, they were individual pixels in the map labeled as obstructed.

The second requirement was that the planner had to consider the robot's geometry and kinematic constraint. Because the robot's motion is automobile-like, the problem is nonholonomic: three variables (x, y, θ) were needed to describe the vehicle's state, yet the direction of the vehicle's motion always tangent to its heading. This kinematic constraint made path planning a more difficult task.

The final and most important requirement was that the path planner be fast enough to operate in real time situations. Lengthly delays during robot operation for path

planning would be unacceptable. As a general goal, several seconds was roughly the appropriate time scale for the path planner. This requirement underlies Companion's basic goal of being a practical, working robot.

6.2 Background

Practical path planning can be divided into two (possibly coupled) tasks. One is the task of searching. It is the process of finding a path, hopefully minimizing a cost function, between a start node and a goal node. At this level, the robot and the environment have been idealized into a mathematical construct: a graph with known connectivity costs. In the last 40 years, research on graph searching has yielded many algorithms including the A* algorithm. This algorithm, because of its efficiency and widespread use, is the focus of this survey of graph search routines.

The second task is the conversion of environmental information (the map) into a searchable form. With sensor data accumulating over time, the world map changes, and the path planner must be able to efficiently translate the representation of the map into a graph-searchable form. This need has implications on how both the map and the planner are implemented. The following subsections describe the A* algorithm and three common path planning strategies that have appeared in the literature.

6.2.1 A* Algorithm

There are a fair number of algorithms for search graphs for the shortest path between two nodes. Among these are Dijkstra's algorithm[3], variations of Moore's algorithm[10][13][11], and the A* algorithm[12]. All of these are sequential and can be implemented on single-processor machines. Among them, the A* algorithm is the most commonly used in path planning problems. It is distinguished from the others by its use of a *heuristic* to accelerate the search process. The quality of the heuristic, an optimistic estimate of the cost from a node to the goal, affects how quickly the algorithm runs.

Because the A* algorithm is used in Companion's path planners, it is explicitly described here. Descriptions of the other algorithms may be found in the references. As it appears in [12], this is the A* algorithm:

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f is minimum.
4. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s .
5. Otherwise expand n , generating all of its successors, and attach to them pointers back to n . For every successor n' of n :

- a. If n' is not already on OPEN or CLOSED, estimate $h(n')$ (an estimate of the cost of the best path from n' to some goal node), and calculate $f(n') = g(n') + h(n')$ where $g(n') = g(n) + c(n, n')$ and $g(s) = 0$.
 - b. If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n')$.
 - c. If n' required pointer adjustment and was found on CLOSED, reopen it.
6. Go to Step 2.

Although the A* algorithm is well known in AI circles, there are several points worthy of clarification. First, the algorithm maintains OPEN and CLOSED lists, which contain nodes in the search. Elements of the OPEN have yet to be expanded and elements of the CLOSED list have been completed. The value of $g(n)$ is cost of the best known path from s to n , and $c(n, n')$ is the transition cost from node n to n' . When deciding which node to expand next, the algorithm considers the value of f for all nodes on OPEN. The quantity f is the sum of the cost from s to the node and an estimate of the the cost from it to the goal. Using the minimum f as the expansion criterion, the node with the shortest possible total cost is expanded first.

In practical applications, A* works better than brute force algorithms like Dijkstra's algorithm when generation of successors (also called neighbors) is time-consuming and heuristic calculation is accurate. Under these circumstances, only a a small portion of the graph enters the search, resulting in an accelerated search. These are the important factors in the development of the Companion search routines.

6.2.2 Visibility Graph Path Planning

Visibility graphs, the first of three path planning strategies discussed here, can be used to generate optimal paths for a point robot moving about on a plane surface with polygonal obstacles. The technique can be generalized to a translating polygonal robot by augmenting all obstacles by the shape of the robot; the robot is then a point in a new obstacle field[8]. It can also be generalized into three dimensions.

The visibility graph of an obstacle field consists of all line segments joining any two vertices of any polygonal obstacles such that the segment does not intersect any of the obstacles. Adjacent vertices of one obstacle are included in the visibility graph. The start and goal points can be considered single-point obstacles and segments joining these points to the rest of the graph are included. The visibility graph necessarily contains of all line segments and vertices in the optimal path from start to goal, hence it suffices to search the visibility graph for the optimal path. Figure 6-1 is a visibility graph for a search between point A and point B in an environment with two obstacles.

A reduced visibility graph, a subgraph of the visibility graph that still contains the optimal path, can also be constructed. Different strategies for constructing reduced visibility graphs have been presented by [9], [5], and others. These techniques include omitting line segments whose two end points are not convex vertices of obstacles and removing non-obstructing obstacles from the graph altogether. Figure 6-2 shows a reduced visibility graph of the obstacle field from Figure 6-1. In it, segments touching

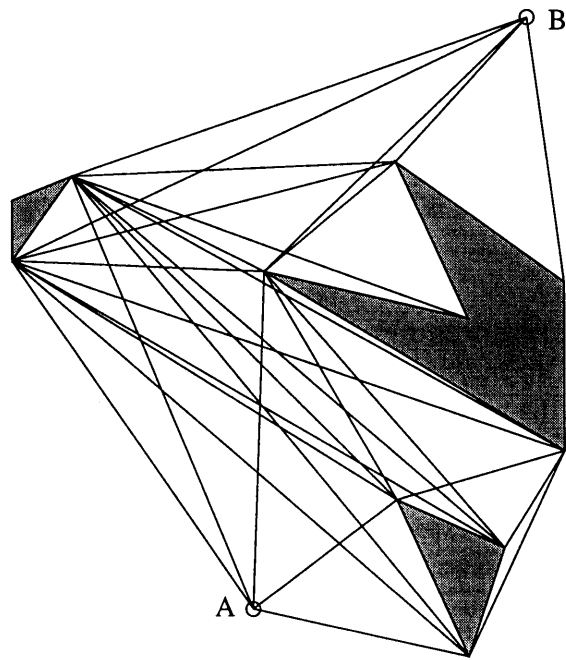


Figure 6-1: A Visibility Graph

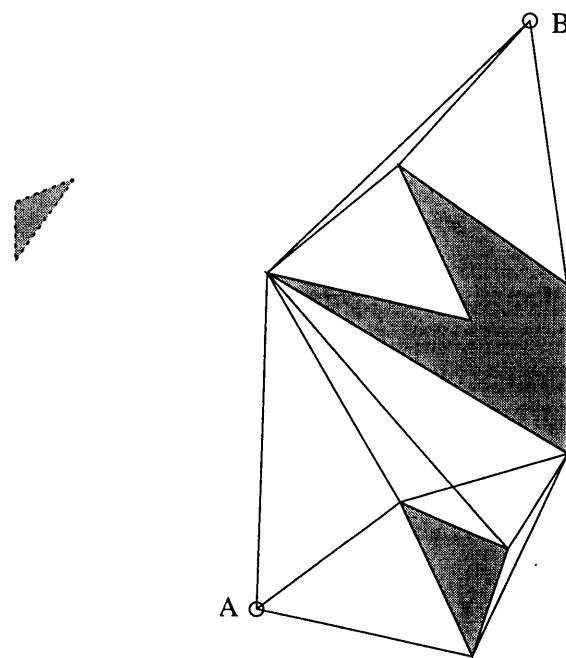


Figure 6-2: A Reduced Visibility Graph

the left-most obstacle have been removed. The A* algorithm is typically used to find the optimal path in the visibility or reduced visibility graph, and the heuristic in the A* search is usually the straight line distance between the node and the goal. Naturally, the search is faster when the graph is smaller.

The visibility graph method works well under some circumstances, but has two important shortcomings with respect to Companion. First, it assumes that any path consisting of straight line segments is feasible. Companion's kinematic constraint, however, renders many paths impossible, in particular, those with sharp turns. Second, Companion's mapping software does not represent obstacles as polygons. Hence, to construct a visibility graph, the planner must first take time to preprocess the probability map into a list of polygonal obstacles.

6.2.3 Configuration Space Path Planning

Configuration space path planning (dubbed "cost wave propagation" in [4] and [18] and "potential diffusion networks" in [15]) discretizes the states of the robot at some finite resolution. The robot can be thought of as a point moving about from pixel to pixel on a fixed grid. Each pixel is said to be one *configuration* of the robot. For a robot such as Companion, the configuration space is three dimensional: it has x , y , and θ positions. A simplified, holonomic version might ignore θ and represent the configuration space two-dimensionally.

A search algorithm such as A* can be used to find a path from the start configuration to the goal configuration. Each pixel in the grid is one node on the graph. The successors of a configuration are nearby configurations corresponding to small motions moving forward or backward, with steering full left, full right, or straight ahead, as in [18]. Successors causing collisions with obstacles are, of course, disallowed.

The simplified two-dimensional version is a good example for visualization of the configuration space planner. In Figure 6-3, the objective is to move the robot from point A to point B . The world is divided into squares and the robot occupies one of them at all times. The shaded squares are obstructed. A neighborhood, shown in Figure 6-4, defines permissible transitions and their costs. In this example, a configuration P has up to eight transitions. Sideways transitions incur a cost of 1, while diagonal motions cost $\sqrt{2}$. These costs correspond to the distance between the centers of the two configurations. Transitions do not occur if the square is obstructed. The eight-neighbor rule limits motion to sideways and diagonal motion, but the neighborhood can be expanded to allow more neighbors, as shown in Figure 6-5. The larger neighborhood makes the search larger, but improves results because it allows motion in more directions.

The important advantage of configuration space planning is flexibility in the selection of configuration successors. The robot's kinematic constraint can easily be imposed on the search by selecting appropriate neighbor nodes. This feature was not seen in the visibility graph method.

The configuration space approach also has a significant disadvantage, however. The discretization of three-dimensional space at high resolution requires a large amount of memory. Array representation of the configuration space quickly becomes

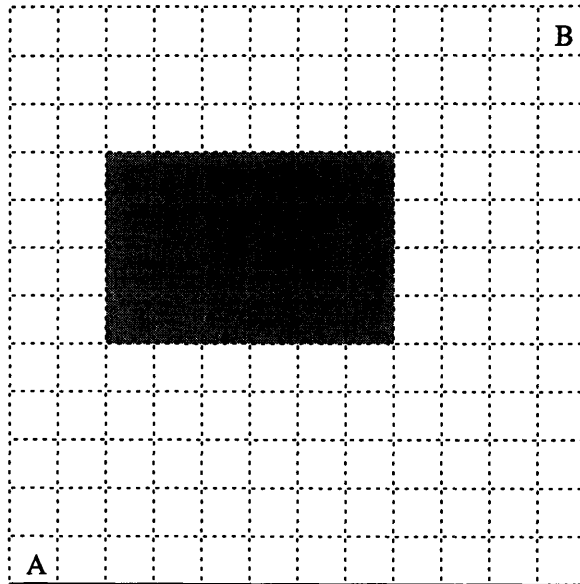


Figure 6-3: A Configuration Space

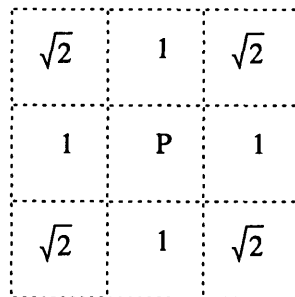


Figure 6-4: A Neighborhood

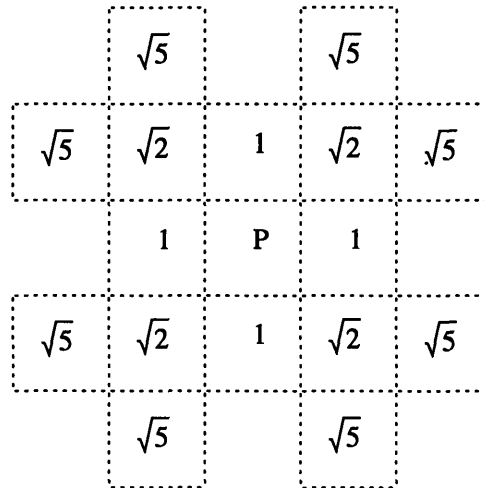


Figure 6-5: An Expanded Neighborhood

prohibitive as distance between start and goal increases. Resolution may be sacrificed for increased overall grid size.

6.2.4 Potential Field Path Planning

Potential fields are a different approach to the path planning problem. In principle, this strategy generates a safe trajectory by applying a force on the robot such that it is attracted to the goal and repelled by obstacles[6]. This tends to generate safe paths rather than optimally short ones. Although the potential field approach works for many obstacle fields, in some cases, local minima may trap the robot and oscillatory paths may be generated[6]. Some methods exist to resolve these difficulties[17][7]. The potential field approach, like the visibility graph technique, does not consider the kinematic constraint.

6.3 Overview of Companion Path Planning

Companion's path planner is a hybrid of variations of visibility graph and configuration space path planning. The principle behind the scheme is to divide and conquer. First, a visibility graph-based scheme generates a coarse path from start to goal. This coarse path consists of a series of way points connected by straight line segments. Second, a series of configuration space searches generates paths between each of the way points until a full path is complete. This approach, although it does not guarantee the optimal path, yields a feasible path given the robot's environment and kinematic constraint.

The configuration space planner, called Search3d in the software architecture, is similar in many respects to implementations seen in the literature. Important differences, described in Chapter 7, include a hash table representation of the configuration

space, a specialized heuristic calculation, and a limited neighborhood.

The visibility graph planner, called Search2d in the software architecture, is a specialized version of the visibility graph method described in this chapter. The differences are that Companion's implementation of search2d generates the visibility graph incrementally to avoid preprocessing the map before a search. This requires a change in the A* algorithm usually used to search the graph. The Search2d module is described in Chapter 8.

Chapter 7

Implementation and Testing of the Search3d Module

The Search3d module is a high resolution search program for generating obstacle-free paths for Companion. At this level, the robot is modeled as a rectangle translating and rotating in the plane. Legal motions are constrained by the no-slip condition (similar to dead reckoning) and the turning radius limitation. Because the robot has two translational coordinates (x and y) and a rotational coordinate (θ), the search is in effect, three-dimensional. It is from this understanding that the module bears the name “Search3d.”

The Search3d process uses the A* algorithm to generate the least-cost path from the start node at (x_s, y_s, θ_s) to the goal node at (x_g, y_g, θ_g) . Like most implementations of the A* algorithm, the generation of the successors of a node, the calculation of cost, and the computation of an accurate heuristic are the important areas of discussion.

This chapter first provides these implementation details. Results of testing of the implementation on simulated environments are then presented.

7.1 Search3d Implementation Details

The Search3d module uses a hash-table representation of the configuration space path planning approach with the A* algorithm. The basic trade-off in the configuration space method is between resolution and memory. For Companion, a reasonable map size is a 3.0 meter by 3.0 meter region at 0.015 meter resolution and 64 distinct headings. This corresponds to a configuration space containing over 2.5 million cells. The array is not only large, but it is also difficult to expand once created. To resolve this difficulty, the configuration space is represented in a hash table. The hash table maps the configuration space into a much smaller block of memory. Since the search typically hits only a small portion of the configuration space, the hash table is particularly appropriate. In addition, the hash table can represent an arbitrarily large region.

For searching the configuration space, a basic A* algorithm is used. It suffices to describe three aspects of the search: the neighboring rule, the cost calculation, and

the heuristic calculation.

7.1.1 Neighboring Rule

The design of neighbor generation rules for the Search3d module went through two iterations. The first attempt closely modeled the ideas in [2]. The resulting paths too difficult for the robot to follow and hence the neighboring rules were revised.

The description of the generation of neighbors as in [2] is done in terms a typical parent node n . Suppose a parent node n specified by its position (x_n, y_n, θ_n) . Then this node has at most six neighbors. The potential neighbors are small motions forward and backward with the steering full left, full right, or straight, as qualitatively shown in Figure 7-1. The numbers in the figure indicate neighbor types, e.g. a neighbor

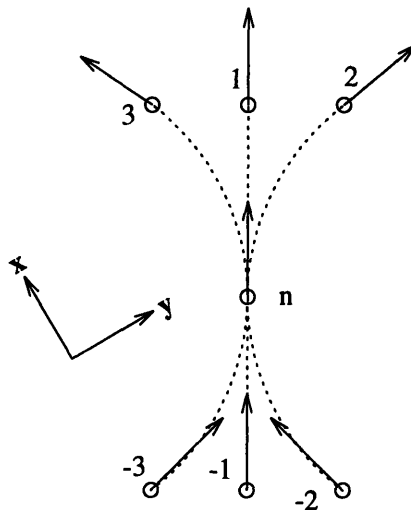


Figure 7-1: Neighbors for Search3d—First Iteration

of Type 2 is a forward curve to the right. The actual distance between node and neighbor is selected by a parameter. Small distances give a finer search resolution at the expense of a larger search. In practice, rotations of $1/64$ of a circle provide sufficient resolution in searches that take a reasonable amount of time.

The size of the search is reduced by removing unnecessary and impossible neighbors. One neighbor is always disallowed because it returns the robot to where it came.¹ For example, if the robot moved straight forward to get to the current position, the algorithm does not generate the neighbor that moves the robot straight backwards. Each potential neighbor is also checked against the map to determine if the new position causes a collision with an obstacle. To do this, a small polygon corresponding to the new space covered by the robot in moving from the current position to the neighbor position is constructed. The Mapper module provides an interface from which it can be determined if any point inside a polygon is obstructed.

¹There is an exception. When expanding the start node, the robot did not come from any other direction. In this case, all six neighbors are possible.

Paths generated using this neighboring rule were indeed optimally short, but they were difficult for Companion to follow. The reason was because the optimal paths frequently included cusps in which the the robot would have to change the direction of turning *and* the direction of motion. Companion's performance on cusps was tested (recall Figure 5-7), and although the performance was fairly good, it was decided not to require Companion to perform such tight maneuvers. The solution to the problem of cusps was to extend cusps tangentially outward, as shown in Figure 7-2. The

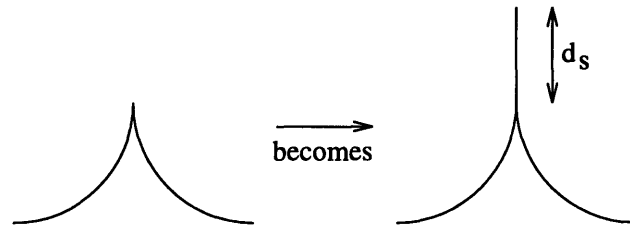


Figure 7-2: Adjustment of Path at a Cusp

parameter d_s denotes how far the robot must overshoot the cusp. To be even more conservative, immediate transitions from turning left to turning right and vice versa were also disallowed. Tests of this type of motion are shown in the bottom two plots of Figure 5-6. The adjustment to the path is shown in Figure 7-3.

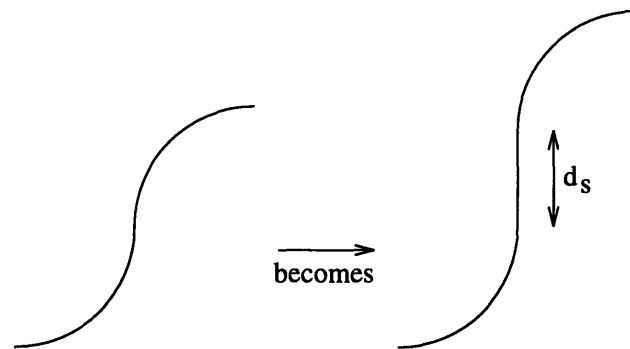


Figure 7-3: Adjustment of Path at a Steering Transition

The limitation on trajectories resulted in the addition of two new neighbors, as shown in Figure 7-4, and more constraints were imposed on the types of neighbors allowed. Specific rules on which types of neighbors could be generated depending on the type of the parent were now needed. Table 7-1 outlines case-by-case which types of neighbor were legal for each type of parent. An example assists in the interpretation of the table. The start node is of Type 0 (and is the only node of that type). From the table, this node has up to six neighbors (Types 1, -1, 2, -2, 3, and -3), depending on the obstacles in the environment. Suppose the Type 2 neighbor (forward to the

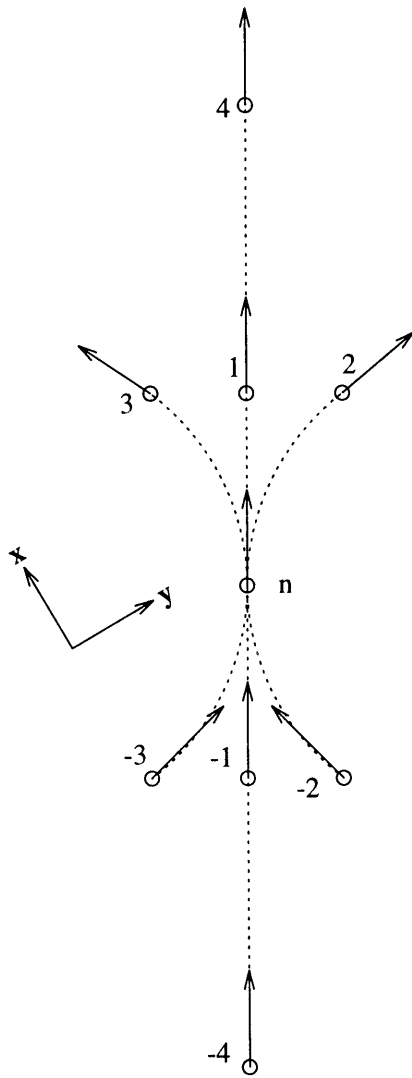


Figure 7-4: Neighbors for Search3d—Second Iteration

Table 7-1 Allowed Neighbor Transitions								
Parent Type	Neighbor Type Allowed							
	1	-1	2	-2	3	-3	4	-4
0	yes	yes	yes	yes	yes	yes	no	no
1	yes	no	yes	yes	yes	yes	no	no
-1	no	yes	yes	yes	yes	yes	no	no
2	no	no	yes	no	no	no	yes	no
-2	no	no	no	yes	no	no	no	yes
3	no	no	no	no	yes	no	yes	no
-3	no	no	no	no	no	yes	no	yes
4	yes	yes	yes	yes	yes	yes	no	no
-4	yes	yes	yes	yes	yes	yes	no	no

right) is expanded. From there the only neighbors are of Type 2 and 4, corresponding to continuing along the circle and moving straight forward a distance d_s .

The severity of the limitations on allowable neighbors provides the benefit of immediately eliminating difficult paths. Generating fewer neighbors also typically result in faster searches. Most importantly, as shown later in this chapter, it gives satisfactory performance.

7.1.2 Cost Calculation

The transition cost from a node to its neighbor is computed as the length of the path plus a penalty if moving to the neighbor requires a change between forward and backward motion. The direction-change penalty encourages the search to minimize gear changes, which tend to make the robot's motion appear jerky. In practice a direction change incurs a penalty of about 0.5 meters.

7.1.3 Heuristic Calculation

Significant computational effort is invested in generating a good heuristic for the nodes in the graph. Again the heuristic is an optimistic estimate of the cost between the current node and the goal node. The heuristic is computed by examining a set of up to 32 paths from the current node to the goal node. One of these paths is the shortest possible path from the current position to the goal. These paths each consist of three segments: a turn, a straight line, and a second turn. The generation of the paths begins by constructing the circles corresponding to left and right turns from both the current and goal nodes. The circles, with radii equal to the robot's minimum turning radius, are shown in Figure 7-5. When one is facing in the same direction as the current node, the distinction between the *left current* circle and the *right current*

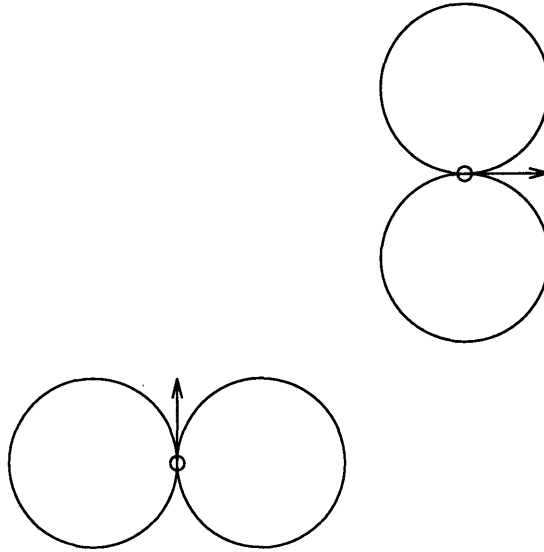


Figure 7-5: Turning Circles for Search3d Heuristic Calculation

circle becomes apparent. Similarly for the goal node, there exist a *left goal* circle and a *right goal* circle.

External tangent lines are then drawn between the circles. Figure 7-6 shows the external tangents between pairs of circles on the same side of the current and goal nodes. External tangents connecting the left current circle and right goal circle generate paths with the robot facing the wrong way at the goal. A similar problem arises from the external tangents connecting the right current circle and left goal circle. Hence those paths are not of interest.

It may also be possible to construct internal tangents between circles, as shown in Figure 7-7. In the case of internal tangents, the interesting tangents are the ones connecting the left current circle with the right goal circle and the tangents connecting the right current circle with the left goal circle. The same-side tangents cause the robot to be facing backward. Internal tangents exist only when the circles are sufficiently far apart.

For each of the tangents (four external and up to four internal), yields four possible paths. To get to the tangent line from the current node, the robot moves forward or backward along the current circle. To get from the tangent line to the goal node, it again moves either forward or backward. This results in a total of four different paths, and an examples are shown in Figure 7-8.

Lengths of each of these paths are then measured. This task is a matter of geometry and details are not provided here. The shortest of these is the shortest path from the current node to the goal. Hence it is used as the heuristic to the goal.

7.2 Testing of the Search3d Algorithm

The search routine was tested, primarily in simple environments. In Figure 7-9,

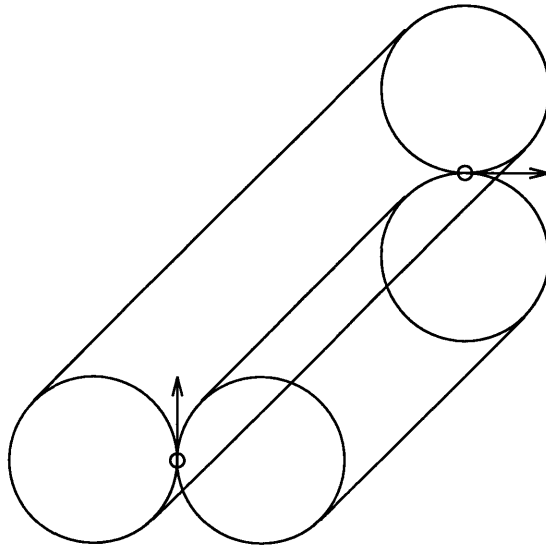


Figure 7-6: External Tangent Paths for Search3d Heuristic Calculation

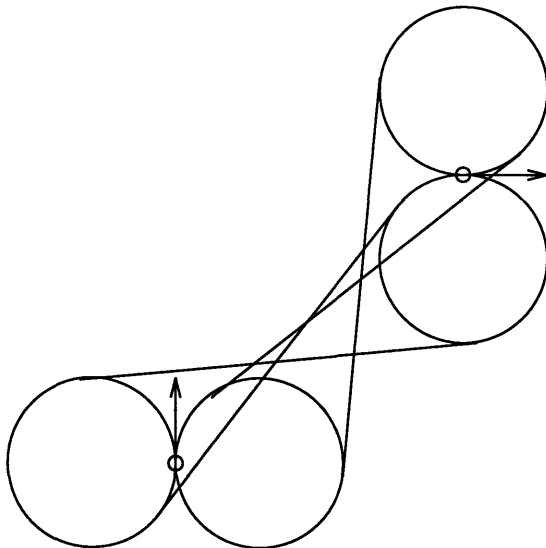


Figure 7-7: Internal Tangent Paths for Search3d Heuristic Calculation

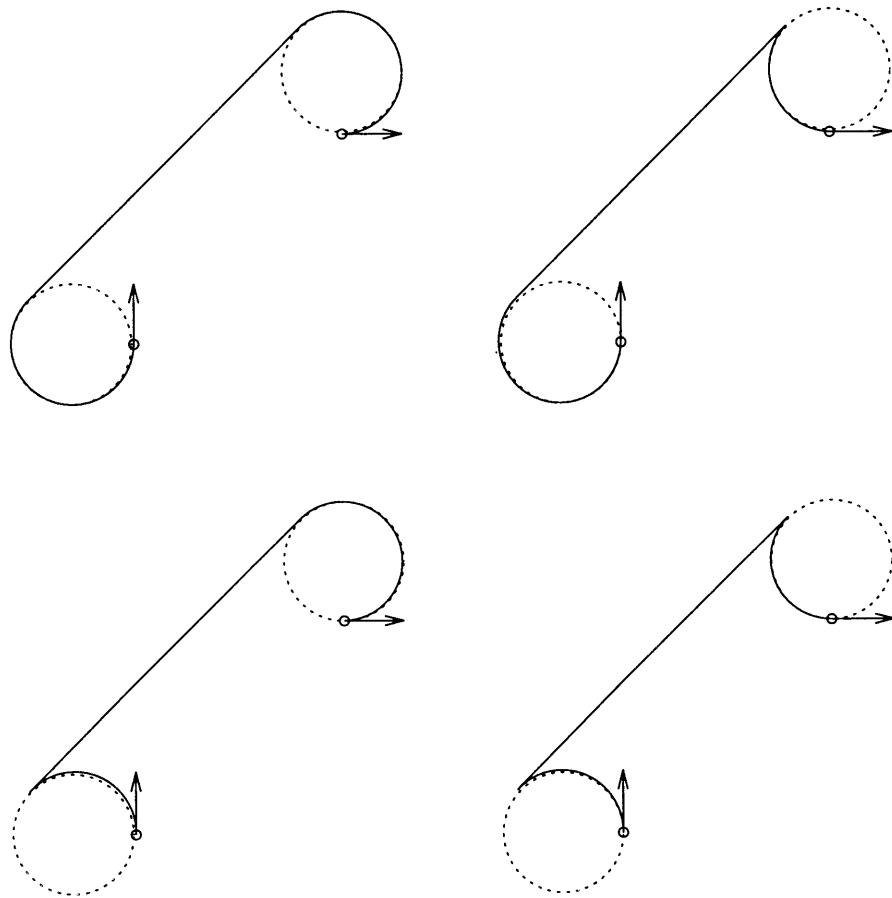


Figure 7-8: Example of the four Paths for One Tangent

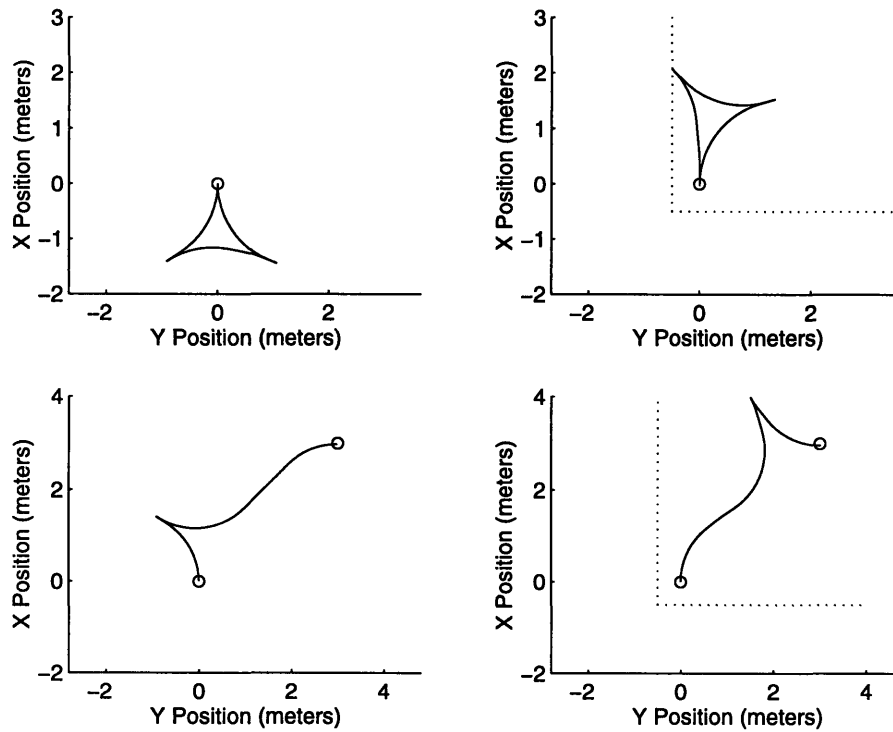


Figure 7-9: Sample Paths Generated by the Search3d Module

four typical paths planned by the Search3d program are shown. In the top two plots, the robot began at $(0, 0, 0)$ and the goal was $(0, 0, \pi)$, i.e. a turn in place. In the left plot, the environment was completely empty and the program took 0.49 seconds to generate the path. Walls, shown as the dotted lines, were added for the right plot, and the routine took 4.06 seconds. In the bottom two plots, the robot again began at the origin; its goal, however, was $(3, 3, 3\pi/2)$. The unconstrained search (left plot) took 0.35 seconds to complete, and the constrained search took 0.57 seconds. In all of these exercises, direction changes incurred a penalty of 0.5 meters, and an overshoot distance d_s of 0.3 meters was used.

The results here are representative of the result obtained from other testing. In particular, in the absence of obstacles, the the search was reliably fast. *With* obstacles, however, the variance in search times skyrocketed. As a general rule, the search routine performed reasonably well when the straight line path between the start and goal was not obstructed. As explained in the next chapter, the Search2d module guarantees that the straight line path is in fact clear. Read on.

Chapter 8

Implementation and Testing of the Search2d Module

The objective of the Search2d module is to efficiently generate an optimal path between a start and a goal point. At this level, the robot's steering constraint and its rectangular shape are ignored; instead, the robot is modeled as a square translating in a plane. The two-dimensional search generates a coarse path from which a more detailed search routine (Search3d) can fill details.

As previewed in Chapter 6, the Search2d module is implemented as a variation of the basic visibility graph/A* search method of path planning. Search2d incrementally generates a visibility graph as the search is being conducted. Only the portions of the visibility graph relevant to the search are generated. This feature saves the computational effort needed to preprocess the entire map into obstacles and to create the visibility graph.

The following two sections describe Companion's Search2d process, first in terms of implementation and second in terms of testing.

8.1 Derivation of the Search2d Algorithm

This description of the Search2d algorithm begins from simple heuristic notions for finding the shortest path between a start and a goal point on a planar surface. Suppose the robot begins at point A, as shown in Figure 8-1, and is trying to reach point B. The obstacles in the environment are "known" in the sense that one can query if a particular point region of space is obstructed or not, but it is assumed that there is no useful *a priori* representation of the obstacles.

As a first step, the straight line path between the start and goal is examined for obstacles. This is simple given the Mapper's grid representation of the environment. If the path is clear of obstacles, the search is complete—a straight path is feasible. Otherwise the nearest obstructed point (point C in the figure) becomes the starting location of an obstacle. By examining the map, the boundary of the obstacle can be found and represented as a series of segments, i.e. in polygonal form. Now, a portion of the visibility graph is constructed. In particular, unobstructed paths from

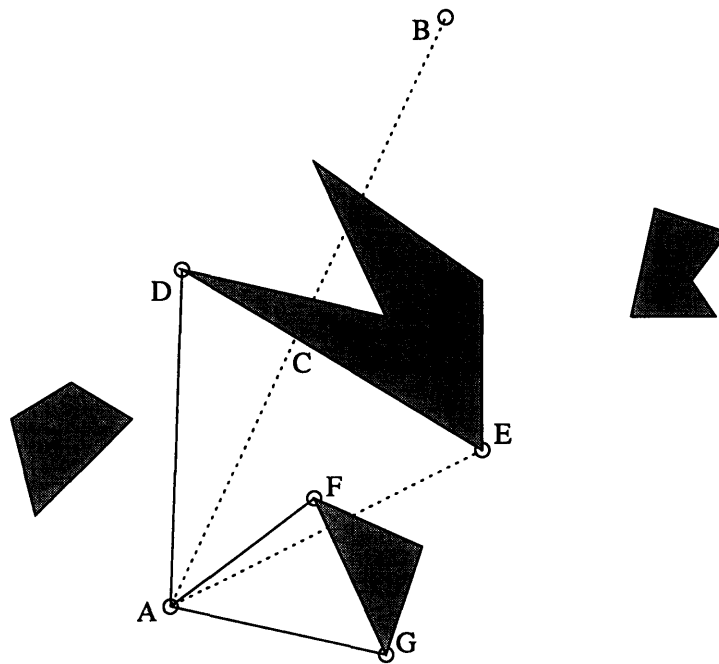


Figure 8-1: Sample Search Situation

the start node to vertices of the obstacle are found. In the figure, the path to point D is unobstructed and hence that point enters the graph as a successor of point A. The path to point E, however, is obstructed by an obstacle. The recursive formulation creates a new obstacle and ultimately creates two successors: point F and point G.

This principle of examining the map for straight line paths to the goal and creating polygonal representations of obstacles as they are encountered is the basis of the search process. Because the search is based on the generation of successors of nodes, an A*-type algorithm can be used. Subtleties, however, require that the algorithm be revised. The next few subsections present the revised A* algorithm and how it can be applied to this search problem, including neighbor generation and heuristic calculation.

8.1.1 Revised A* Algorithm

The revised A* algorithm is shown here:

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f is minimum.
4. If n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s .

5. Otherwise expand all the nodes n_i on the optimal path from n to s , generating a subset of successors, and attach to them pointers back to n_i . For every successor n'_i of n_i :
 - a. If n'_i is not already on OPEN or CLOSED, estimate $h(n'_i)$ (an estimate of the cost of the best path from n'_i to some goal node), and calculate $f(n'_i) = g(n'_i) + h(n'_i)$ where $g(n'_i) = g(n_i) + c(n_i, n'_i)$ and $g(s) = 0$.
 - b. If n'_i is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest $g(n'_i)$.
 - c. If n'_i required pointer adjustment and was found on CLOSED, reopen it.
6. Go to Step 2.

The expansion of a node in the normal A* algorithm entails finding *all* successors of that node. In that way, that node need not be revisited for further expansion. In the revised algorithm, only a subset of successors is found. No specification is made as to which successors must be found, but generating a node on the optimal path accelerates the search. This revision alone would result in a non-optimal search if a node on the optimal path is not in the subset of generated successors. This problem is resolved with a second revision: the expansion of a node now includes finding successors on all nodes on the optimal path from the start to that node. In this way, successors previously omitted can be added. The task is to ensure that all successors of a node are generated at some point.

In the context of an incremental search, the algorithm provides a means of only creating successor nodes located on known obstacles. As new obstacles are found, successors from old nodes to that obstacle are generated. This topic is the subject of the next subsection.

8.1.2 Neighboring Rules

The rules for generating successors (neighbors) are given here, and explanations follow. For each node n_i in the optimal path from s to n :

1. If n_i is n , then add the goal obstacle to OBSTACLE(n_i).
2. For each vertex v_{jk} of each obstacle b_j in OBSTACLE(n_i):
 - a. If v_{jk} is the same as n_i , this is not a neighbor.
 - b. If the angle made by n_{i-1} to n_i to v_{jk} is greater than 90° , this is not a neighbor.
 - c. If the straight line path from n_i to v_{jk} does not form a tangent at obstacle b_{ij} , this is not a neighbor.
 - d. If the straight line path from n_i to v_{jk} does not form a tangent at the obstacle associated with n_i , this is not a neighbor.

- e. If the straight line path from n_i to v_{jk} intersects any segments in the known obstacle field, this is not a neighbor.
- f. If the straight line path from n_i to v_{jk} is obstructed by an obstacle b_l , for each node n_m on the optimal path from s to n_i , add b_l to $\text{OBSTACLE}(n_m)$. This is not a neighbor.
- g. Otherwise, v_{ijk} is a neighbor.

$\text{OBSTACLE}(n_i)$ is a list of obstacles of interest to node n_i . These are the obstacles to which visibility graph lines will be constructed. The goal obstacle is just the goal point itself—in the context of this search, it is convenient to think of the goal as an obstacle with a single point. The goal point always appears on the $\text{OBSTACLE}(n_i)$; hence, the search always attempts to find the direct path from the node to the goal.

Items a–f are series of tests that must be passed before a vertex can qualify as a neighbor. They are not critical to the algorithm but rather accelerate it. For conciseness in this section, these tests are outlined in Appendix A. In test f, the straight line path between a node and a potential neighbor is examined for obstructions. If an obstruction is found, it is added to the OBSTACLE lists. When the algorithm cycles again back to testing vertices on other obstacles, this new obstacle has been inserted on the list. Hence, all new obstacles discovered during the expansion of a node are tested for legal successors.

A visualization of this circumstance appears in Figure 8-2. In this example, the

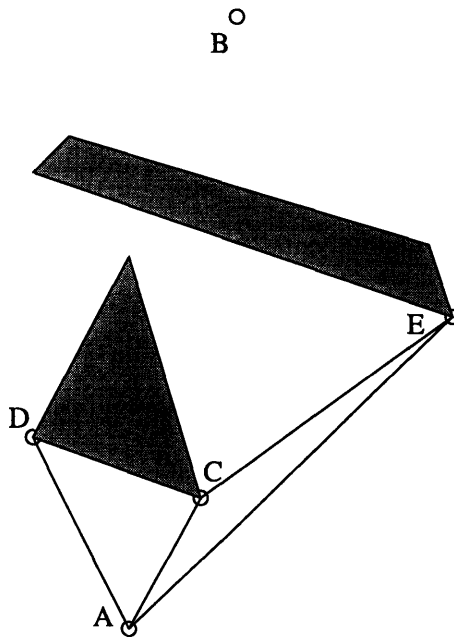


Figure 8-2: Example of a New Obstacle Found

search in progress has generated two successors, point C and point D, of the start node point A, and is about to expand point C. The discovery of the obstacle between

point C and the goal point B results in the addition of point E as a successor of point C. However, because a new obstacle was found, the search also attempts to find successors of point A on the new obstacle. This results in the successor point E of point A. The natural process of the A* algorithm immediately determines that the shortest path to point E is the direct line from point A and eliminates the path from point A to point E via point C.

8.1.3 Cost and Heuristic Calculation

The cost and heuristic functions are unusually simple. The transition cost from node n to node n' is simply the distance between them. Similarly the heuristic from node n' to the goal is just the distance from n' to the goal. It is reassuring that not all things are difficult.

8.1.4 Review of the Search2d Algorithm

A look at the algorithm, after the fact, provides insight into its construction. The algorithm solves the basic problem of transforming a grid representation obstacles to a polygon representation. Because obstacles are “created” only when they interfere with potential paths, only relevant regions of the environment are examined for obstacles. Since obstacles are first identified by boundary pixels, one can march around the obstacle, identifying vertices as they appear. Hence, the algorithm solves the problem of transforming the map representation from grid to polygon without preprocessing the entire map. This gives the Mapper the ability to cover large regions without affecting path planning.

8.2 Testing of the Search2d Module

The search2d module is one of the better performers on Companion. Figure 8-3 shows a sample test run of the program. This sample ran in less than 0.12 seconds. This result is typical for this module.

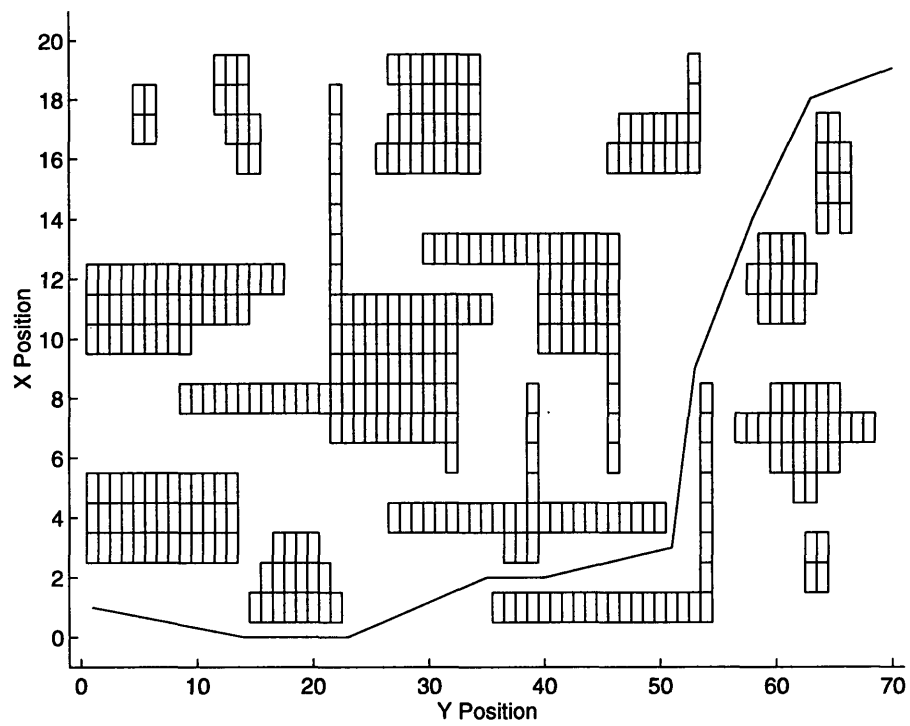


Figure 8-3: Sample Run of Search2d

Chapter 9

Implementation and Testing of the Planner Module

The Planner module is the coordinator of planning and plan execution. As the parent module of Search3d and Search2d, it manages their execution status. As the parent of Trajectory, it supervises the execution of circle and line following commands. In effect, the Planner is a middleman, interpreting way points from the User, generating plans via Search3d and Search2d, and issuing them to Trajectory.

This task can be simple and it can be complex. Due to time constraints, a full implementation of this module has not been completed. The current Planner process is instead a simple test program for the coordination of the path planning and path execution. It operates in a simple sequential manner. First, it awaits a way point from the User module. Second, it generates a coarse path to that goal using the Search2d modules. Then, for each of the points in the coarse path, it generates a fine path using the Search3d module. When the plan is complete, the entire plan is issued to the Trajectory module, which then executes the commands.

The Mapper plays only a passive role in this implementation. Again, due to time constraints, full integration of planning with mapping has not been completed. The Mapper is instead loaded with a simulated environment. The Search2d and Search3d modules treat this information as though it were real. Hence the Planner was tested with the real robot on simulated environments.

Figure 9-1 and Figure 9-2 are two samples of experiments with this planner. Figure 9-1 represents a way point that requires the robot to turn two corners (walls denoted by dotted lines). Although the robot successfully navigated the path, it collided with the simulated wall. This error can be attributed to an underestimate of the robot's size in the Search3d process. In Figure 9-2, the robot was required to pass through a doorway. The robot was originally facing in the direction of the x -axis on the left side of the opening, and the commanded way point was on the other side again with the robot facing along the x -axis. Planning and execution were successfully completed.

It is important to emphasize that this implementation of the Planner module is merely a simple demonstration of coordination among the Search2d module, the Search3d module, the Trajectory module, and to a lesser extent, the Mapper module.

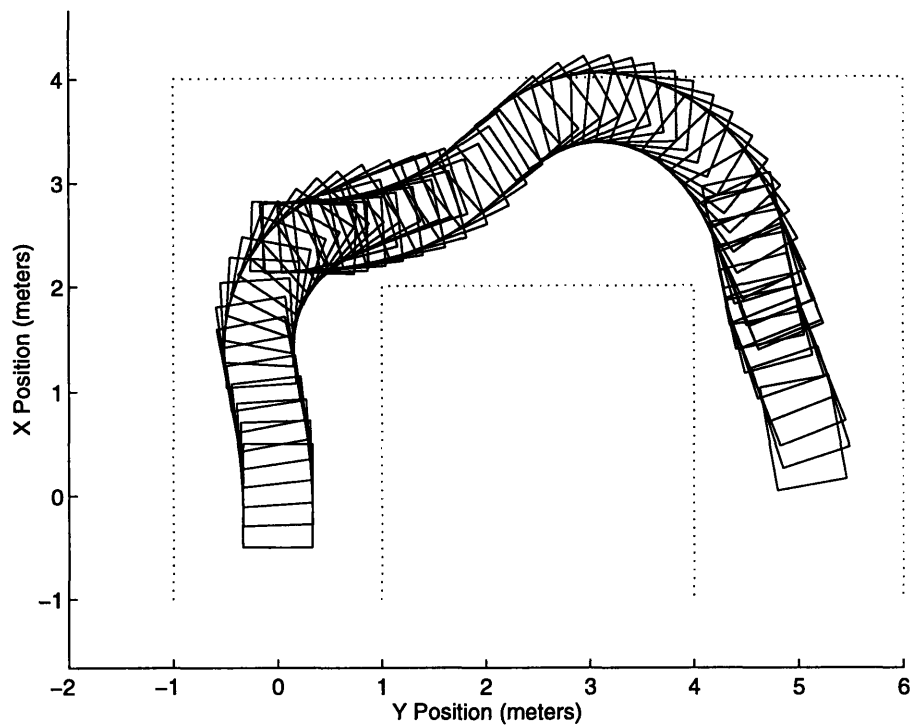


Figure 9-1: Corner-Turning Plan Execution

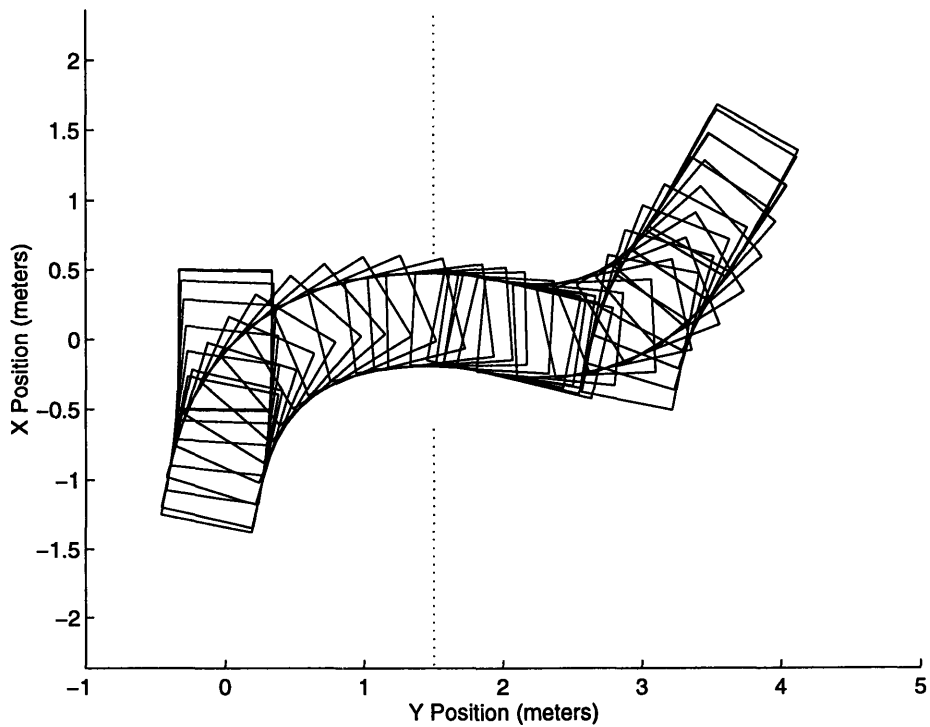


Figure 9-2: Doorway Plan Execution

This Planner module is a program that unifies all of the modules into one. During testing, more than ten processes implemented for Companion are running simultaneously on the robot's two computers.

Possible improvements to this module abound and are probably more important than the results given here. First, the planner needs a good mechanism for connecting Search3d paths. In particular, the path given by Search2d does not constrain the robot's heading at any of the intermediate points. In the current implementation, headings are artificially imposed prior to the path planning with Search3d. This results in awkward transitions between intermediate points, as evidenced in Figure 9-1. The second area of possible improvement is in the area of replanning. Currently, the planning is based on an artificial environment and executed with no regard to real obstacles. An important element of a real planner is the ability continually adjust the path as new obstacles become known and as the robot's strays too far from the original path. This requires the system to replan and to integrate the new plan into the execution sequence. It also requires the integration of real sensing in the Mapper module. These two ideas are the most important goals to achieve with the planner. Longer term ideas are provided in the conclusion.

Chapter 10

Conclusions and Recommendations

Companion has come a long way, but much more can be done. With working device drivers, a flexible software architecture, and good software implementations, Companion is a working test bed for mapping, navigation, and path planning, and it has the potential to become a more intelligent machine.

10.1 Ideas for Improvement

There are many areas of possible improvement. Consider the interface to the drive and steering motors. The software controls the speed of the drive motor and the position of the steering motor by simulating commands the wheelchair's original control system, which results in undesirable delay and hysteresis. The existing lookup table used for steering control performs only marginally, and there is no controller for speed. A better control system, one that compensates for the those idiosyncrasies, could be implemented. Better still, the wheelchair control system could be completed bypassed. The control of speed, though not a critical part of Companion, a desirable feature.

Currently, Companion lacks absolute sensors for navigation. Position is only known relative to the starting position of the robot. This is adequate when no information is carried over from previous missions. However, there are circumstances when localizing the robot is useful, for example, in localizing the robot within an *a priori* map or in generating a single map from multiple missions. The addition of GPS or at least a compass would provide some measure of absolute position.

Companion's navigation can also be improved. Currently, three sensors (a gyroscope and and two encoders) perform dead reckoning. The existing steering potentiometers are not used. A new dead reckoning scheme, one that filters the redundant sensors, could be implemented. Any absolute sensors, like GPS or a compass, could also be integrated into the filter.

The implementation of the Trajectory module performs fairly well, but improvements could be made. The 0.03 meter steady state bias and trajectory overshoot in the circle-following routine could be eliminated by replacing the current proportional controller with a better system, such as a PID controller. Improvements in Trajectory

would make higher level modules, such as the Search3d more flexible. Limitations on the allowable sequences of lines and circles could be removed, and overall performance of the robot would improve.

The main shortcoming of the implementation of the Search3d module is its speed. Although it frequently executes searches in under a second (excellent performance), it may take up to several minutes for searches to fail (e.g. in obstructed environments) or for difficult trajectories to be found. The current implementation allows a search to time out after a set period of time to avoid indefinite blocking of the search process. The most important improvement that could be made to Search3d is to make it run at a reasonable speed more reliably.

10.2 Ideas for New Development

With satisfactory performance of low-level systems, development on Companion can proceed to a higher level. The most pressing need is for a better planning system. Ideas for development appeared in Chapter 9. New types of missions are also a possibility for higher level planning. The current nominal goal of moving to a way point can be expanded. One interesting mission is exploration. The objective in exploration is to acquire a full representation of a particular region in the robot's environment. Path planning is necessary for the robot to move about, but the point to which it moves changes. One scheme might continue to move the robot towards areas not already well mapped. The exploration mission is the beginning of practical use for Companion, as it can be used to explore areas too dangerous for humans.

Multiple robot cooperation is a goal for the longer term. With its mapping and planning capability, Companion can be important component in a team of cooperating agents. A current project at Draper's Intelligent Unmanned Vehicle Center is a mission to clear unexploded land mines. Lacking a manipulator, Companion is incapable of actually picking up mines; however, its mapping ability could be useful in describing the environment for the other agents.

Teleoperation is also an interesting area to explore. Experience with other robots in the Intelligent Unmanned Vehicle Center has shown that radio modems can provide reasonable bandwidth communication between a remote robot and a ground station. A ground station can command the robot at different levels—from simple actuator commands up to way point commands. Because it can provide excellent feedback for a remote operator, a ground station is an important step towards practical use of autonomous mobile robots.

10.3 End Game—A Final Commentary

I¹ believe that the work on Companion thus far has been a marvelous effort requiring the contribution of many people with skills in a broad range of technical fields.

¹The time has come to abandon the third person.

Although at least two theses have been written on the robot, there is still more to be done than has already been finished. I sincerely hope that the ideas for new work on Companion outlined in the chapter someday come to fruition. It will be then that the effort that I have invested in Companion will be finally rewarded.

Appendix A

Search2d Neighbor Tests

In the context of a two-dimensional search, Subsection 8.1.2 enumerated a series of tests that a potential neighbor must pass in order to qualify as a neighbor. This appendix provides the rationale behind the rules. They are again listed here:

- a. If v_{jk} is the same as n_i , this is not a neighbor.
- b. If the angle made by n_{i-1} to n_i to v_{jk} is greater than 90° , this is not a neighbor.
- c. If the straight line path from n_i to v_{jk} does not form a tangent at obstacle b_{ij} , this is not a neighbor.
- d. If the straight line path from n_i to v_{jk} does not form a tangent at the obstacle associated with n_i , this is not a neighbor.
- e. If the straight line path from n_i to v_{jk} intersects any segments in the known obstacle field, this is not a neighbor.
- f. If the straight line path from n_i to v_{jk} is obstructed by an obstacle b_l , for each node n_m on the optimal path from s to n_i , add b_l to $\text{OBSTACLE}(n_m)$. This is not a neighbor.
- g. Otherwise, v_{ijk} is a neighbor.

Of these, rule e and rule f make sense by definition of a neighbor. If the straight line path between the node and the potential neighbor is obstructed either by a known or a new obstacle, that point cannot be a neighbor. Rule a is a reflexive test—if the potential neighbor and the node are one and the same, there is no reason to add it as a new neighbor.

Rule c and rule d can be made clear from Figure A-1. The claim is that the line segment joining the node and its neighbor must form tangents on the obstacles at both ends, i.e. the extension of the line segment does not immediately intersect either obstacle. In the figure, the potential neighbor point C does not form a tangent at its obstacle. A better path can be constructed by moving from point A to the vertices adjacent to point C. The tangency rule implies that all neighbors are convex vertices of obstacles.

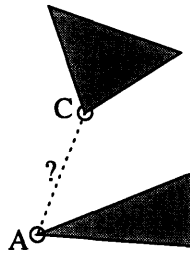


Figure A-1: Tangency Test for Neighbors

Rule b, unlike the others, is a result of the grid representation of obstacles. Because each pixel in the map is a square, obstacles, at the finest level of detail, consist of nothing but 90° turns, as shown in Figure A-2.¹ As a result, a turn of more than

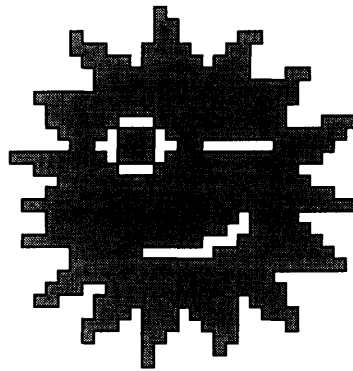


Figure A-2: Close up of an Obstacle

90° either results in intersection with the obstacle or makes the configuration a non-tangent situation. Because this is a simple test to perform, it is done second only to the reflexivity test.

¹Obstacle concept courtesy of Chuck Tung.

Appendix B

Implementation of the User Module

The User module, as previewed in Subsection 3.3.10, is the responsible for interaction with Companion's operator, i.e. the user. For now, the User module is implemented as a simple user interface from the Laptop console. By running the User process, an operator can select a way point, change the way point, halt the mission, or quit. It is really nothing more than a front end for the Planner process.

Ultimately, the User module will be implemented as an interface between the Laptop and a remote ground station. An operator at the ground station can communicate with the robot via a radio modem. The User module will be responsible for managing this communication on Companion and for its normal duties as the parent of the Planner module. Ground station implementation is an area of design and implementation yet to begin.

Bibliography

- [1] Eugene A. Avallone and Theodore Baumeister III, editors. *Marks' Standard Handbook for Mechanical Engineers*. McGraw-Hill, New York, 9th edition, 1987.
- [2] Jerome Barraquand and Jean-Claude Latombe. On nonholonomic mobile robots and optimal maneuvering. In *IEEE International Symposium on Intelligent Control*, pages 340–347, 1989.
- [3] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.
- [4] Leo Dorst and Karen Trovato. Optimal path planning by cost wave propagation in metric configuration space. In *Mobile Robots III*, pages 186–197, 1988.
- [5] Li-Chen Fu and Dong-Yueh Liu. An efficient algorithm for finding a collision-free path among polyhedral obstacles. *Journal of Robotic Systems*, 7(1):129–137, 1990.
- [6] Kikuo Fujimura. *Motion Planning in Dynamic Environments*. Springer-Verlag, Tokyo, 1991.
- [7] Y. Koren and J. Borenstein. Potential field method and their inherent limitations for mobile robot navigation. In *IEEE International Conference on Robotics and Automation*, pages 566–571, 1990.
- [8] J. C. Latombe. *Robot Motion Planning*. Kluwer Academic, Norwell, 1991.
- [9] M. Montgomery, D. Gaw, and A. Meystel. Navigation algorithm for a nested hierarchical system of robot path planning among polyhedral obstacles. In *IEEE Conference on Robotics and Automation*, pages 1616–1622, 1987.
- [10] E. F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on Theory of Switching*, pages 285–292, 1957.
- [11] U. Pape. Implementation and efficiency of moore-algorithms for the shortest route problems. *Mathematical Programming*, 7:212–222, 1974.
- [12] Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, Reading, 1984.

- [13] M. Pollack and W. Wiebenson. Solutions of the shortest-route problem—a review. *Operations Research*, 8:224–230, 1960.
- [14] QNX Software Systems, Ltd. *QNX 4 Operating System: System Architecture*, 1993.
- [15] Wei Shen, Jun Shen, and J. P. Lallemand. Path planning by potential diffusion networks. In *Mobile Robots VII*, pages 13–24, 1992.
- [16] Steve Steiner. Mapping and sensor fusion for an autonomous vehicle. Master's thesis, Massachusetts Institute of Technology, 1996.
- [17] R. B. Tilove. Local obstacle avoidance for mobile robots based on the method of artificial potentials. In *IEEE International Conference on Robotics and Automation*, pages 566–571, 1990.
- [18] Karen I. Trovato. Autonomous vehicle maneuvering. In *Mobile Robots VI*, pages 68–79, 1991.