Localization, Mapmaking, and Distributed Manipulation with Flexible, Robust Mobile Robots

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Russell Gregory Brown
May 1995

Localization, Mapmaking, and Distributed Manipulation with Flexible, Robust Mobile Robots

Russell Gregory Brown, Ph.D.
Cornell University 1995

Current mobile robots (mobots) have extremely limited usefulness in real-world applications, due primarily to low capability and low reliability. Current mobots cannot navigate in environments which have not been extensively engineered to be mobot-friendly; they are also incapable of performing useful duties *at* their destination. They often have poorly designed hardware and primitive software-development environments, making the development of new mobile robot protocols slow and difficult.

In this thesis, we describe three ways to make mobile robots more usable. We present algorithms for mobile robot self-localization, a design paradigm for reliable mobile robots, and protocols for cooperative large-scale manipulation by mobile robots.

Localization is the process of determining the robot's location within its environment. More precisely, it is a procedure which takes as input a geometric map, a current estimate of the robot's pose, and sensor readings, and produces as output an improved estimate of the robot's current pose (position and orientation). We describe an algorithm which performs mobile robot localization using a geometric model of the world and a point-and-shoot ranging device. We also describe a rasterized version of this algorithm which we've implemented on a real mobile robot equipped with a laser rangefinder we designed.

We next focus on the mobile robots themselves. We have designed and built several mobile robots at Cornell, that feature robustness, flexibility, and ease of programming. Using a modular design approach, we've attained an unusual state for a university robotics lab: our mobots are almost always functional. In their basic configuration, the robots can perform many tasks; also, it is easy to add sensors and

actuators to our mobile robots, allowing their use in many different applications (e.g. manipulation).

We are interested in protocols for manipulation of large objects (e.g., boxes, wheeled carts) by cooperating mobile robots, particularly protocols that are asynchronous, on-line, and use no communication between mobots. We've developed a *Pusher-Steerer* model for cooperative manipulation, which enables two mobots to manipulate objects through complex paths. We describe and analyze the model and describe its performance in several real experiments.

Biographical Sketch

Russell Gregory Brown was born on October 5, 1966, in Tullahoma, Tennessee. Spending eighteen years in this nice little town in the South, he acquired a great love for learning, a great love for building things (with wood, with electronics, and with code), and a great love for being somewhere else.

Combining his three great loves, in 1984 Russell went to Vanderbilt University. It was there, double-majoring in Computer Science and Electrical Engineering, that he first realized a strong aversion to single-field specialization. At Vanderbilt, his strong suits were digital electronic design, assembly language programming, automatic control theory, the theory of algorithms, and the practice of theatrical set, lighting, and costume construction. Deciding the course of his future was difficult, given this variety of interests, but he was aided in his decision by numerous graduate programs, all of whom, with the exception of Cornell, declined his services. Entering a Ph.D. program in Computer Science at Cornell, Russell was forced to select a new set of interests more fitting to a graduate student at a powerhouse of theory such as Cornell. He was able to retain most of his academic interests, finding that robotics, his new found focus, required familiarity with all of these interests and more. In the interest of academic survival, though, he was forced to abandon theatrical interests. To fill the void left in his life by this loss, Russell discovered several new interests. Among these were ice hockey, Habitat for Humanity, and banjo. Habitat for Humanity occupied a large part of his attention for several years, as he served for 1 1/2 years as one of the co-presidents of the fledgling Cornell Campus Chapter of Habitat for Humanity, two years as a board member of the Tompkins County Habitat affiliate, and four years as a core member of the work projects committees of both groups. Banjo was undertaken as an 11th hour effort to prevent Russell from finishing his thesis in a timely fashion; when this failed to slow him down sufficiently, Russell expanded his efforts from mere banjo playing to encompass banjo building as well. Russell departed Cornell to work at Sandia National Labs in Albuquerque, New Mexico.

This thesis is dedicated to my parents, Manuel and Glenna C. Brown. Were it not for their love, their devotion, and the interaction between their particular quirks and my own, I would never have gotten to where I am now. The combination of these factors insured that I would deviate significantly from the mean.

Acknowledgements

This thesis would not exist without a great deal of help and co-work from a large number of people. My thesis advisor and special committee chair, Bruce R. Donald, has been a source of support, direction, and drive throughout my time at Cornell. My other committee members, Geoffrey Brown, Daniel P. Huttenlocher, and Rich Zippel all provided me with valuable guidance during my later years.

Many members of my incoming Ph.D. class had a significant effect on both my life and my research career at Cornell. Two of these friends, James Jennings and William Rucklidge, made a task that would, for me, have been nearly impossible, merely very difficult. Jim and I have had countless research discussions, many of which led to new research insights for one or the other of us. We worked closely together and with others to develop the Cornell Mobile Robots, and worked together on the mobile robot cooperative manipulation work which is described in chapter 5. Jim also helped me to select my first banjo. Over the years, William's inspiration and brilliance have provided me with many short simple solutions for what would, without him, have been difficult problems. These ranged from questions of computer vision, computational geometry, and computer architecture to questions such as, "What's the name of the housemaid in the 'Last Will and Temperment' sketch?" In addition, James Allan, Brad Glade, Paul Stodghill and Judith Underwood provided me with around 1500 scintillating and fascinating lunch/research conversations.

L. Paul Chew helped greatly in the development of the mobile robot localization algorithms described in chapter 2. The techniques we use in that chapter are a great deal simpler and more elegant than what we had before he volunteered his assistance.

This paper describes research done in the Robotics and Vision Laboratory at Cornell University. Support for our robotics research is provided in part by the National Science Foundation under grants No. IRI-8802390, IRI-9000532, IRI-9201699, and by a Presidential Young Investigator award to Bruce Donald, and in part by the Air Force Office of Sponsored Research, the Mathematical Sciences Institute, Intel

Corporation, and AT&T Bell laboratories. The author was partially supported by a grant from Sandia National Laboratories as part of SNL contract No. AJ-7129.

Mobot Credits:

The Cornell Mobile Robots are the result of many people's efforts. Although I played an integral part in the development of virtually all in-house-developed subsystems, many others were involved in realizing many of these subsystems. The Cornell Mobile Robotics project predates me slightly; it was begun by Bruce Donald and James Jennings shortly before I joined. The modular enclosure concept, the decision to use Real World Interfaces products, and much of the overall direction for the project comes from their input. In addition, a number of undergraduate and Master of Engineering students were involved in designing, building, and programming many parts of our mobots. I'd like to list the people involved, and give them credit for the roles they played in the making of our mobots.

Early versions of the Generic Controller were designed by Electrical Engineering master's students Marius Moscovici, Bob Blondell, Steve Shollenberger, and Bill Ford. Former CS undergrads Craig Becker and Kevin Newman assisted me in developing version 3.0 of the Generic Controller.

The infrared proximity detection and serial communication system grew out of an initial implementation by Bill Ford. Craig Becker began the process of converting the robots to an all-digital infrared design; this process was continued by Kevin Newman and Greg Whelan.

Mark Battisti and David Manzanares, former undergraduate students of Mechanical Engineering, did all of the mechanical and structural design and construction for mobot parts made at Cornell. Mark made the Velcro Grabber and the pan/tilt head for Lily, and constructed the controller cards for them under my supervision. David made some small improvements on the pan/tilt head and designed and built the supplemental battery stages that allow Lily and Rosemary to run longer off charger. David also made the mounts for the pan/tilt units on Rosemary. Both Mark and David played a role in the construction of the bumpers on all of the robots. Former mechanical engineering Ph.D. student Michael O'Donnell did the original design and construction work for Camel's treaded base. Former undergraduates Todd Pack and Kevin Newman built and programmed Camel's controller.

James Jennings wrote the Cornell Generic Controller Monitor, which is the low-level operating system which runs on the GC. He also played a big part in specifying some of the subsystems we have on our robots.

All of the names mentioned here played at least some role in almost all of the subsystems. I am grateful to them for their efforts on the lab's behalf. We would have had a hard time getting to where we are without any of them. I would also like to thank Grinnell More and Tyson Sawyer of Real World Interfaces, Inc., for the work they have done on our behalf. They have made several changes to the control software for their B12 wheelbase in response to requests made by us. Some items in their product line, including the modular enclosure, grew out of design discussions between RWI and our lab.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Current mobile robots are not useful. This extremely strong statement represents personal opinion, but it clearly represents the opinion of all but an extremely small handful of people. This statement is not the theme of this thesis, but it does provide much of the motivation for its existence.

Why do we think that mobile robots are not useful? In our opinion, the cost (in money and effort) of getting an autonomous mobile robot to do almost any useful task far outweighs the benefits of doing so. The sorts of tasks that one would like to use autonomous mobile robots for include such diverse elements as[1]:

- **Household Cleaning**: vacuuming, mowing the lawn, general picking up and putting away
- **Delivery**: intra-office mail delivery, delivery of beverages and meals
- **Security**: patrolling perimeters, watching for unauthorized personnel
- **Hazardous Waste Cleanup**: exploring, mapping, measuring, and cleaning up of chemical- or radioactive-contaminated areas
- **Terrestrial and Extra-terrestrial Space Exploration**: exploring inaccessible or uninhabitable locations on other planets, or here on Earth.

Although many people are conducting research on the application of mobile robots to these and other tasks, we don't find many instances of robots performing these tasks at more than a laboratory scale. Tasks such as household cleaning, delivery,

[1]This list represents a large subset of the tasks one sees mentioned in the context of mobile robotics, when looking through the robotics scientific literature, the popular news media, and works of science fiction.

and security are more economical when performed by humans, for a number of reasons:

- **Unit cost**: At the present time, the cost of the hardware for a mobile robot capable of one of these tasks runs several tens of thousands of dollars.
- **Low Reliability**: Almost all existing mobile robots have too low a mean-time-between-failures to be useable by non-roboticists.
- **Low Capability**: The existing robotics science base has not enabled robot designers to create a combination of sensors, actuators, and algorithms capable of performing any of these tasks well enough to permit a successful implementation of a robot for one of these tasks. (For this set of tasks, we disallow solutions that require engineering the robot's environment, as we feel that this is not reasonable for a general solution. A certain amount of environment engineering is acceptable for technology of proven usefulness[2], but mobile robotics is an unproven technology: If the environment needs to be engineered for mobile robots to be usable, this creates an extra hurdle which will make it harder for the technology to gain acceptance.)

Tasks such as hazardous waste cleanup and exploration are also seldom performed by autonomous mobile robots. For these tasks, the cost of the unit is much less significant, since the alternative involves making it safe for humans to undertake the tasks. The low reliability issue becomes even more critical, though, since we can't send the repairman after the robot. The capability issue can be somewhat remedied by sufficient expenditure of resources and increased specialization; we still cannot justify engineering the environment, though, since we cannot send anyone into the environment ahead of the robot to do so. Another issue arises when considering hazardous tasks:

- **Teleoperation**: Since the cost of a human operator is affordable, but the cost of providing environmental safety for that operator may not be, a teleoperated robotic platform (where some or all of the controls are in the hands of a human operator) is often an affordable and feasible alternative to a fully autonomous mobile robot.

The reason we feel we can safely say that only a few people feel mobile robots are useful is the plain fact that only a few people have mobile robots performing useful

[2]For example, we modify the environment by building roads and running telephone wires, because we find cars and telephones useful.

tasks in their everyday lives; only a small subset of those few that do are not robotics researchers. If we were to make a similar claim with respect to televisions, cellular phones, automobiles, or other modern conveniences, our claims could be refuted by the fact that many people find these devices useful on a daily basis. We feel safe making our assertion, therefore, because it seems we cannot be refuted in a similar way with respect to mobile robots.

1.1 In This Thesis

What are we doing to remedy this lack of usefulness? In this thesis, we address what we regard as three of the fundamental requirements for mobile robot characteristics and capabilities. Specifically, we require that our mobots

1. **Perform a useful task**: The robots must be able to do something that will be regarded as a useful, practical skill.
2. **Navigate**: If the robot has a task of the form "Do action x on object y at location z", the robot must be able to identify y and z reliably.
3. **Function reliably**: The robot must perform its actions reliably, without unexpected breakdowns, over many iterations and over a long period of time.

In addition, since a mobile robot used for research should facilitate mobile robotics research, rather than inhibit it, our mobile robots must

4. **Be easy to use**: In addition to the requirement that the robot must operate reliably whenever its users apply power, we require that the robot be easy to program, reprogram, reconfigure, and modify.

In this thesis, we discuss our attempts to address these requirements. We address the navigation issue by presenting algorithms to perform mobile robot *localization* (chapter 2), which is the process of computing the mobile robot's *pose*, or position within its environment. We treat the localization problem first as a precise combinatorial problem in computational geometry. We then show how the problem can be solved by a real mobile robot with limited computational and sensing resources. We address the reliability and the ease-of-use issues in chapters 3 and 4 by describing the Cornell Mobile Robots, the SCHEME programming environment which we use to program these robots, and the modular software and hardware architecture with which we have achieved a much higher reliability than is typical for research mobile robots. Finally, we describe one of the capabilities we have developed for

Table 1.1: Contributions of this Thesis.

Contribution	Where	Description
Exact $\mathbb{R}^2$ Localization Algorithm	Ch. 2	$O(m^2n^2\log(mn))$ algorithm
Rasterized Localization Algorithm	Ch. 2	$O(mn^2)$ algorithm
Mapmaking with Time Variability	Ch. 3	Tracking time variability of regions
Robust, Flexible Mobile Robots	Ch. 4	Easy to use mobots that work
Pusher/Steerer Manipulation	Ch. 5	Multi-robot manipulation

our robots that contributes to the usefulness of our mobile robots: in chapter 5, we describe and analyze our protocols for *distributed manipulation*, which use two robots to manipulate boxes and other large objects along complex paths, potentially allowing them to serve as robotic furniture movers. We entitled this chapter "The Other Sofa Mover's Problem", to highlight a significant application to which our manipulation protocols can be applied, and also to contrast this problem against the historical "Sofa Mover's Problem"[3], which studies the issue of planning a path to move an object through a set of obstacles, without considering the physics or control issues involved in moving that object.

1.1.1 Thesis Organization

This thesis is broken down into seven chapters and three appendices. Chapters 2 and 3 describe navigation aspects of our system (most specifically localization, but also mapmaking and path planning functions). Chapter 4 and Appendices A and B describe the Cornell Mobile Robots and their subsystems (Appendix A describes the Cornell Generic Controller that is used in all of our mobots, and Appendix B the Three Beam Laser Rangefinder that runs on LILY). Chapter 5 discusses our protocols for cooperative mobile robot manipulation. Chapter 6 provides experimental results for algorithms and protocols described in chapters 2, 3, and 5, while Appendix C provides code listings for the functions described in those chapters. The navigation chapters, the mobot chapter and appendices, and the manipulation chapter do not depend on one another; it is possible to undertake chapter 4, for example, without first digesting chapters 2 and 3, or chapter 5 without any of chapters 2 through 4. It is our hope that the reader will find all of the chapters worth reading, but we realize that readers' interests will differ.

[3]also called the Piano Mover's Problem and the Spaceship Docking Problem (see, for example, [Lat92].

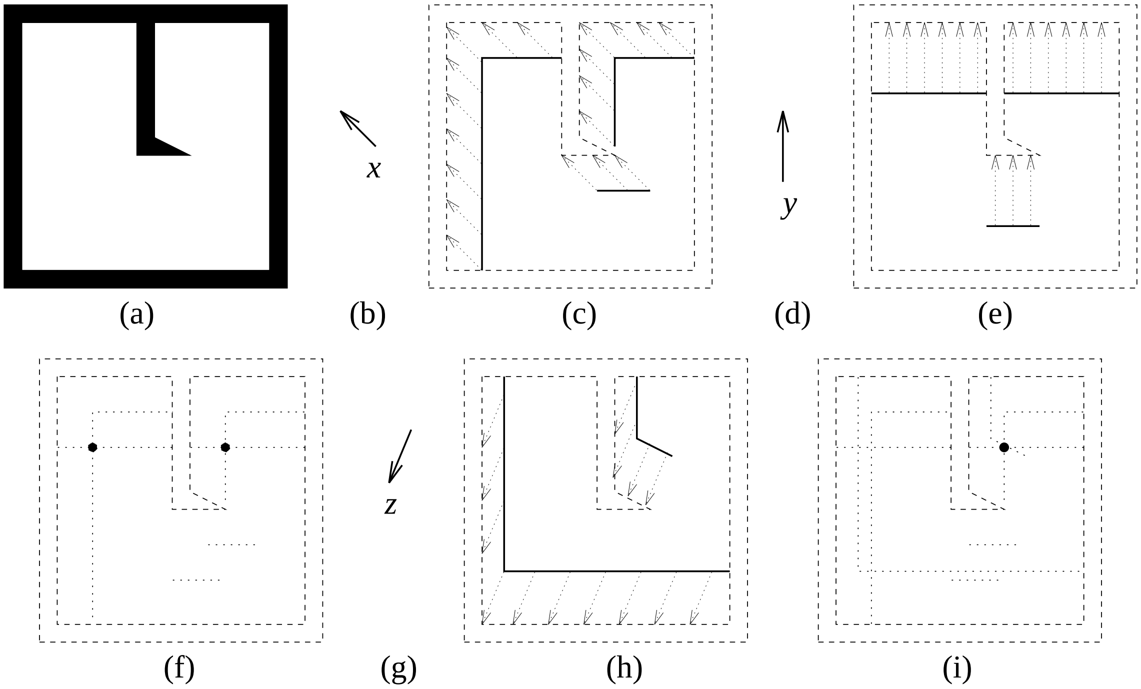


Figure 1.1: How the localization algorithm works: Given (a) a map M and (b), a range vector x, we can determine (c) the set of feasible poses $FP(M, x)$. (d) and (e) show y and $FP(M, y)$. The black dots in (f) are $FP(M, \{x, y\})$. (g) and (h) show z and $FP(M, z)$. (i) shows the single point which is $FP(M, \{x, y, z\})$. The robot must occupy that position to get range vectors x, y, and z in map M.

1.1.2 Mobile Robot Localization

Localization is the process of determining the robot's location within the environment. More precisely, it is a procedure that takes as input a map, an estimate of the robot's current pose, and a set of sensor readings (any of these may be incomplete, and all are toleranced by error bounds), and produces as output a new estimate of the robot's current pose. By *pose*, we mean either (a), the position and orientation of the robot in the world or (b), the translation and rotation necessary to make a set of sensor readings (or a local map, built from those readings) best match an *a priori*, global map.

The main concept underlying the localization algorithms presented in this thesis is that of the *feasible pose*. A feasible pose is one that is consistent with the available range and map information: our algorithms partition the robot's configuration space into poses that conflict with available range information and poses that do not; these latter are the feasible poses. Poses are feasible or infeasible relative to several parameters: a pose is feasible with respect to a map and a range vector if that pose places the robot such that the range vector terminates at an obstacle boundary and is otherwise unobstructed. A pose is feasible with respect to a map and a *set* of range vectors if it is feasible with the map and each of those range vectors, individually. We can determine feasibility within an error bound: to do this, we allow the length of a range vector to vary within an uncertainty bound and determine the poses that are consistent with some pose plus-or-minus the uncertainty bound. We can extend limited-error-bound feasibility to multiple range readings. We can also determine the minimum error bound for which there exists a feasible pose. Finally, with multiple range readings, we can count how many range readings are feasible with a given pose and map, and look for poses that are feasible with a maximal number of the supplied range readings.

How do we compute the feasible poses for a particular map M and range probe z? At an intuitive level, the procedure is to find all locations for the robot where its rangefinder would return z in the world M represents. Treat z as a vector whose tail is located at the robot and whose head lies at an obstacle boundary: if z can be situated with its tail at a point p, such that the head of z touches an obstacle boundary, but no obstacle boundary intersects z anywhere *other* than its head, then p is a feasible pose for map M and range probe z. Figure 1.1 shows how our method of localization works. We denote the feasible poses for a map M and a single range probe, z (resp. a set of range probes, Z) by $FP(M, z)$ (resp. $FP(M, Z)$). Part (a) of figure 1.1 shows a map, M. Parts (b), (d), and (g) show three vectors, x, y, and z, that represent three different range probes. Part (c) shows $FP(M, x)$: The solid black lines are $FP(M, x)$, because those are all of the positions in M where we can place the tail of x so that its head touches an obstacle, but its body doesn't. Parts (e) and (h) show $FP(M, y)$ and $FP(M, z)$. Part (f) shows $FP(M, \{x, y\})$: The two black dots are the locations in M where $FP(M, x)$ and $FP(M, y)$ intersect. Since there are two dots, the two range probes, x and y, are insufficient to localize the robot in M uniquely. By taking a third range probe, z, though, the robot can be uniquely localized: part (i) shows the intersection of $FP(M, x)$, $FP(M, y)$, and $FP(M, z)$. This intersection is located at the single black dot. Therefore, $FP(M, \{x, y, z\})$ is that single point, so the robot must be located at that point.

In chapter 2, we discuss the feasible pose concept at length and provide algorithms to compute feasible poses. We give an exact combinatorial algorithm to compute feasible poses for a set of range probes and a map specified as a set of polygons, for the case where the robot has translational uncertainty (unknown x-y position) but no rotational uncertainty (the robot knows which way it is facing). This algorithm makes use of tools from computational geometry. We provide anal-

Table 1.2: The time-complexity of our algorithms: For the combinatorial algorithms, n is the number of features in the map, m is the number of range probes, and s is the number of intersections between features, which is $O(m^2n^2)$ worst-case. For the rasterized algorithms, the map has size $n \times n$.

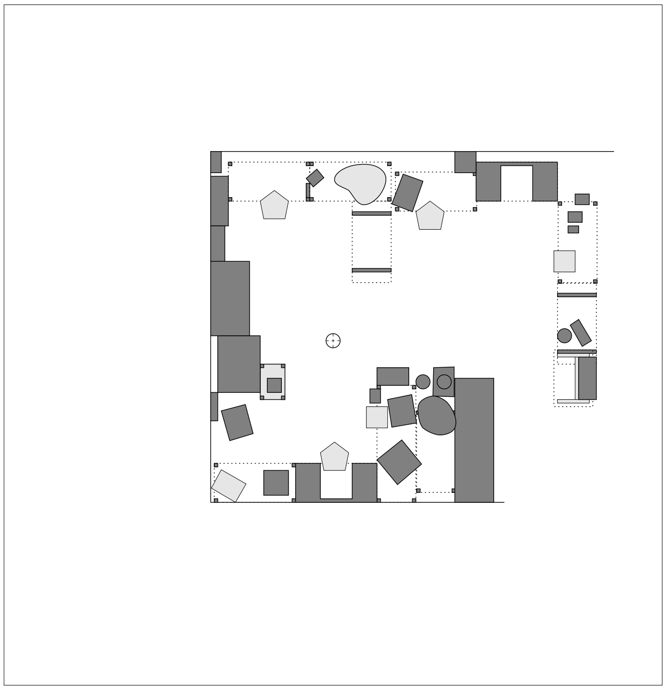
	1 probe	m probes
Exact Combinatorial	$O((n+s)\log n)$	$O((mn^2+s)\log(mn))$
Rasterized	$O(n^2)$	$O(mn^2)$

ysis of the algorithm, and describe how it can be extended to the case where the robot *doesn't* know which way it is facing. We then present and analyze algorithms for computing feasible poses in the *rasterized* domain; in this case, the map is a binary array where ones represent obstacle cells and zeros represent empty cells. We feel that rasterized algorithms are more appropriate for solving many robotics problems; one reason for this opinion is that rasterized algorithms often are easier to implement and run faster than their exact counterparts. Figures 1.2 and 1.3 show maps of our lab (a hand-drawn map for reference, and a map generated by LILY) and the output of the rasterized localization procedure for the map in figure 1.2a and instantaneous range data. For more examples, see Chapter 6. Table 1.2 summarizes the asymptotic complexities of the various localization algorithms.

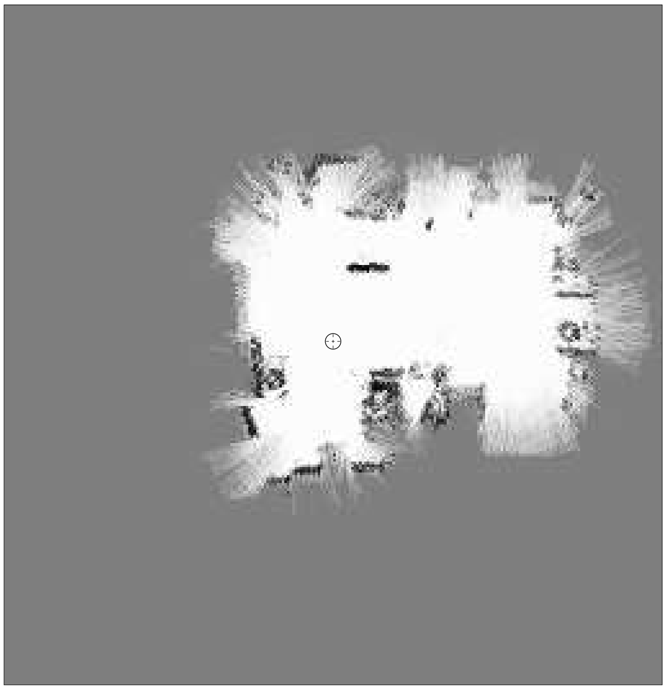
1.1.3 The Mobot Experimental System

Chapter 3 could be informally titled "The Rest of the Navigation System". We have implemented a navigation system on our mobile robots which includes the rasterized localization algorithms described in chapter 2, but has a number of other capabilities including mapmaking, localization, and path planning and execution.

The mapmaking technique we chose for this system is a variant of the *statistical occupancy grid*. Moravec and Elfes introduced this method of mapmaking in [Elf87, Mor89,ME85] as a way of making detailed geometric maps using noisy sensors (sonar rangefinders, in their case). A simple occupancy grid is a bitmap, where ones represent occupied cells and zeroes vacant cells. The main idea behind the statistical occupancy grid is that, since range data is inexact, the contents of each cell should be a probability: a cell containing a high probability is likely to be occupied, while a cell with low probability is likely empty. Moravec and Elfes use an update rule based on Bayesian probability to maintain the contents of each cell based on range



(a)



(b)

Figure 1.2: The circles in these figures show the actual location of the robot for localization test 3. The circles are overlaid on (a) the hand-drawn map and (b) Lily's map.

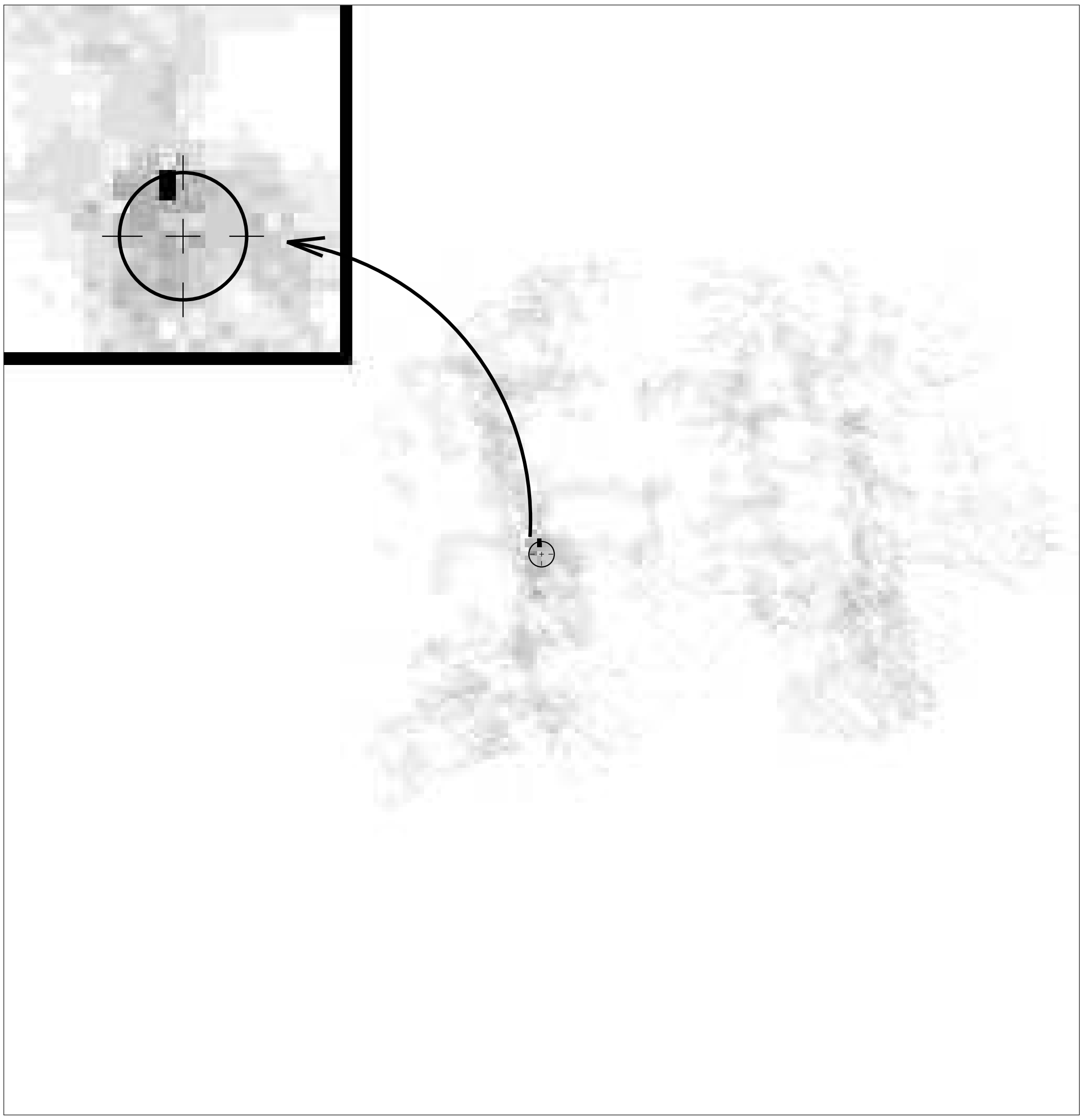


Figure 1.3: The output of the localization procedure. In this picture, the darkness of each cell is proportional to the number of range readings for which that cell is feasible. The black cells are the most feasible: these are the most likely positions for the robot. The circle overlaid on the output shows the actual location of the robot for this localization test.

readings that impinge upon that cell. We chose to adapt their work for a number of reasons, but primarily because a grid-based mapmaking technique fit best with our ideas about rasterized computational geometry for robotics (as discussed in chapter 2). Our method differs somewhat from theirs: we use a different structure for maintaining occupancy data, and we use an update rule that, while keeping track of the probability of occupancy for each cell, integrates absolute change in probability. This way, frequently changing cells can be distinguished from those whose values seldom change. These ideas will be discussed further in section 3.2. Figure 1.2(b) shows an occupancy-grid map of our lab generated by LILY using range data obtained from her laser rangefinder.

The path-planning module in this system is also a grid-based/rasterized technique. We've chosen to implement a simplified version of the C-space potential field path planner of [LRDG90]. They implemented a fast $\mathbb{R}^2 \times S^1$ planner using specialized graphics hardware to compute the configuration space of a planar obstacle map and robot and to grow a potential field from the specified goal, such that from any start position, the robot need only follow the descending gradient of the potential field to reach the global minimum in the potential field, which occurs at the goal. We instead implement an $\mathbb{R}^2$ planner by computing a planar C-space and growing the potential field in that space. This is sufficient for our purposes, because our robots are rotationally symmetric.

Finally, we discuss a number of minor functions that use the laser rangefinder to perform various navigation-related tasks. We have implemented several primitive-recognition routines: they identify features such as convex and concave corners and long, straight edges in specified portions of the environment. These functions can be used as subroutines to the localization procedure: detecting nearby corners is useful for determining which range probes are most useful for the localization procedures outlined in chapter 2. In addition, the ability to detect long edges in the environment is useful for rotational localization. Several of these minor parts of the system, along with their applications, are discussed in section 3.4.

The high-level environment we use for programming our robots is a SCHEME byte-code interpreter running on the robot that can be tied to a byte-code compiler and symbolic debugger running on a SPARCstation. This system, implemented by Jonathan Rees on top of a SCHEME-48 virtual machine (and described in detail in [RD92]), allows the user to establish a remote read-eval-print loop on the robot (across a detachable RS-232 serial line). This enables the user to define functions, download programs, and evaluate forms on the robot, with the output displayed on the SPARCstation. It allows the user to execute programs on the robot *with-*

out the SPARCstation, simply by removing the serial line. The built-in library to Mobot SCHEME allows the user to access all of the sensors and actuators on the robot through SCHEME functions, and to define arbitrary actions to be executed upon certain events; for example, each robot has several push-buttons that can be bound to execute an arbitrary zero-argument lambda expression. Mobot SCHEME has multiple threads, allowing the user to define light-weight threads that execute asynchronously.

1.1.4 Robust, Flexible, Easy to Use Mobile Robots

The purpose of chapter 4 is to describe in some detail the Cornell Mobile Robots and the design goals that drove their development. We introduce the robots and list their capabilities, then describe the robots' architecture, both hardware and software. Finally, we detail each of the sensors and actuators on the robots. We argue that we met our goals of robustness, flexibility, and ease of use, through a combination of modular design and careful engineering.

Our main goal has been to design and build robots that will be flexible and easy to use. In section 4.2, we describe the basic architecture of the Cornell Mobile Robots and why we feel that the architecture makes the mobots flexible and easy to use. We show and describe the topology of the hardware architecture, which is a loosely coupled set of independent microcontrollers operating sensors and actuators under the control of a central, more powerful processor. We then categorize the processors in the robots by the basic role they play. The microprocessors in our robots perform three main roles: high-level control, interprocessor communication and routing, and low-level, point-of-sensing control. We also talk about how we achieved the physical modularity that is necessary to the degree of flexibility we desire. Finally, in section 4.3, we discuss each sensor or actuator subsystem that is part of one or more of our robots. We describe the functionality of each subsystem at a high level, and in some cases give a more detailed analysis of the subsystem.

We wanted to design mobile robots that would permit us to perform experiments on a variety of mobile robotic topics. We didn't want to specify what those topics were in advance, though, so we wanted robots that we could change easily. Since we knew that many people would be using these robots, we also wanted robots that would work reliably over an extended period of time. Our design goal, therefore, was to build robots that would be robust, flexible, and easy to use. We realized that goal with our modular robots. Our robots have physical modularity, in that it is straightforward to interchange, add, remove, and move components. They also have software modularity, making it easy to add new sensors and actuators. The

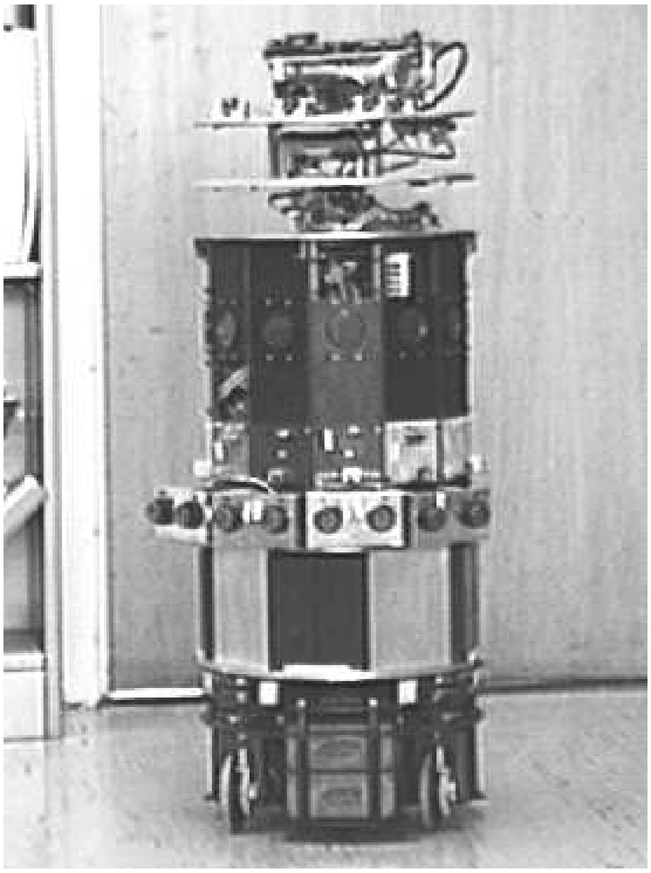


Figure 1.4: LILY, one of the Cornell Mobile Robots

fact that each module is a separate entity makes modules easier to design, build, and maintain. The open-ended I/O structure of the robots' microcontrollers makes it possible to add many different kinds of sensors and actuators, usually without affecting the performance of other subsystems.

We have built four mobile robots in our laboratory, TOMMY, LILY, ROSEMARY, and CAMEL. All four of them have 68xxx-based microcomputers that we program in a dialect of SCHEME ([AS85,CR92] called Mobot SCHEME, which is designed for use on mobile robots. All four are equipped with microcontroller boards that control their sensors and actuators. Each has a modular enclosure equipped with hinged doors and interchangeable mounting plates that allow us to add, remove, and interchange sensors, actuators, and their controlling electronics easily and flexibly. Each has an array of 12 Polaroid-style sonar range sensors, 12 infrared proximity-detectors/communicators, a set of pushbutton bumpers, and buttons and lights for user-robot communication. TOMMY has a speech synthesizer that lets him output state verbally, and a mount that allows a video camera to be attached to his head for remote-controlled active vision experiments. LILY has a laser rangefinder mounted on a motorized pan/tilt platform, and a stage equipped with extra batteries, allowing her to perform longer autonomous experiments and bear more electronics. ROSEMARY also has an extra battery stage, plus two individually servoable pan/tilt heads and a DSP/frame-grabber card, enabling her to do on-board computer vision experiments. TOMMY, LILY, and ROSEMARY have Real World Interfaces B12 synchro-drive wheelbases, which have small control and sensing errors. CAMEL has a larger, treaded base, which allows him to move larger objects, but with greater control and sensing errors. In section 4.3 of this chapter, we will discuss each of the subsystems of our robots in more detail. We discuss the Generic Controller and the Three-Beam Laser Rangefinder in sections 4.2.1.2 and 4.3.2.2; in-depth discussions of these subsystems can be found in Appendices A and B.

1.1.5 Distributed Manipulation by Cooperating Mobile Robots

We are interested in cooperative manipulation tasks/protocols that can be executed by real robots (in particular the ones we have access to). Furthermore, we want to study tasks/protocols that are not readily serializable; i. e., that can be more efficiently executed by multiple robots than by a single robot. Other factors we would like to minimize are communication and global knowledge among the robots. We have designed a set of protocols for leading objects along trajectories using a model of manipulation which we call the *pusher-steerer* model. Each of the two robots in this model behaves as a point mass with wheel kinematics (zero frictional force parallel to the direction of travel, unless driven, and infinite frictional force perpendicular to the direction of travel). The *steerer* robot knows the path along which the object is to move, but provides no motive force to the object (or to its own wheels, either for motion or for braking). Instead, it follows the programmed path as it is pushed along, determining direction of travel as a function of distance travelled along the programmed path. The *pusher* robot, on the other hand, knows nothing about the path to be travelled. Its program is simply to apply pushing force, and to turn as necessary to keep its forward face against and aligned to the pushed object. To move an object with the pusher-steerer model, we place it between the pusher and the steerer and engage the pusher. As the pusher moves forward, it pushes the box along in front of it. The box pushes the steering robot along in front of *it*, causing the steering robot to move forward along its programmed trajectory.

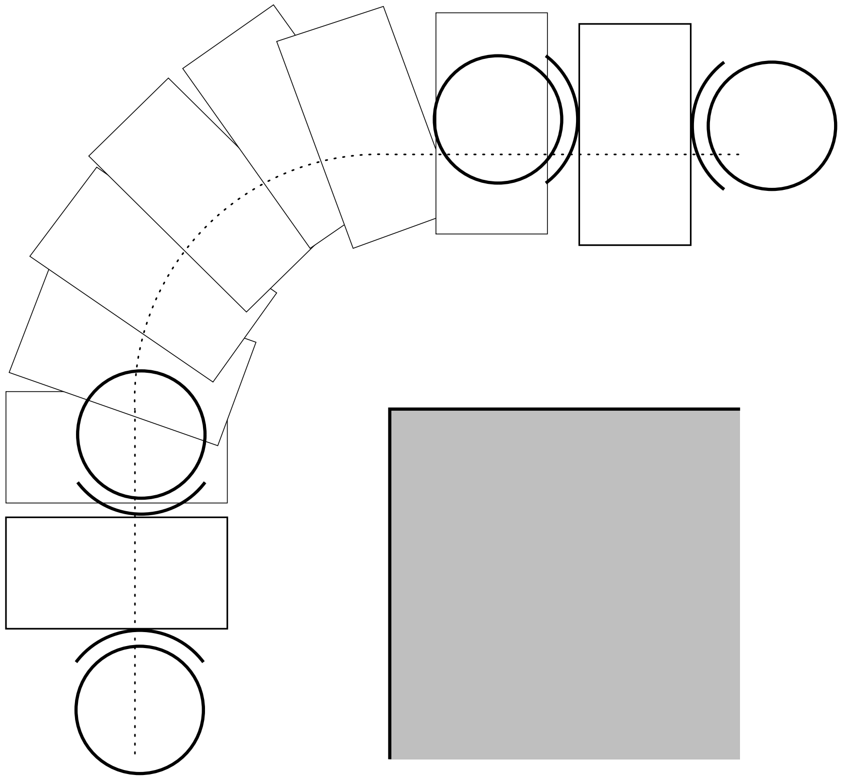


Figure 1.5: Two robots moving a box around a corner using the pusher/steerer model

In chapter 5, we describe the pusher-steerer model in more detail and present code that implements the PUSHER and STEERER portions of a pusher-steerer manipulation system. We present mechanical analysis of how a pusher-steerer system manipulates objects along straight-line trajectories and along circular arcs. One aspect of the pusher-steerer idea that we find exciting is that there is **no direct communication** between the pusher and the steerer. The only communication between them is information transmitted indirectly through the object being manipulated. Also, the low information requirement of protocols PUSHER and STEERER is exciting: The pusher needs no knowledge about the path the steerer is following, and no knowledge about the object being manipulated, except to know that it has a flat face on which to push. The steerer needs no knowledge about the object except that it has a flat front face parallel to its back face, and needs no knowledge about the pusher. The most exciting aspect of all, however, is our claim that the pusher-steerer model of manipulation effectively reduces the problem of trajectory-planning for large-scale manipulation to the problem of nonholonomic motion planning. Since Lynch and Mason ([LM94]) and Barraquand and Latombe ([BL93]) have recently published practical nonholonomic path-planners (for single-robot pushers and train-like robots, respectively), it would appear that the technology for planning and executing complex manipulation trajectories is now available!

1.1.6 Experimental Work

In chapter 6, we will discuss the results of some experiments we have performed with our mobots. We chose these experiments to allow us to assess the performance of our algorithms. These tests also served as a small sample of the kinds of experiments our mobots are capable of performing. We show maps of a small office and part of a large lab made using our laser rangefinder and the mapmaking algorithms described in chapter 3. We also show results of running the localization algorithms from chapter 2 from locations within these environments, using laser rangefinder data and those maps.

The second part of Chapter 6 explores the capabilities of the pusher/steerer protocols described in chapter 5. We show the results of experiments designed to test the ability of two of our robots to manipulate objects through complex paths. We experimentally estimate the minimum turning radius of a system comprised by two robots and a given box for a number of boxes. We do this by programming the robots to lead their box through an arc of a given radius (varied over a range of one-to-several meters) and measuring how far the robots-and-box system can follow an arc of that radius, and also by allowing the steering robot to select a turning radius automatically, based on contact between it and the box.

1.2 Related Work

1.2.1 Navigation, Localization, and Mapmaking

A number of papers have been written in the last eight years or so about localization (or pose-estimation). Most authors, including we, appear to be using roughly the same informal definition of pose estimation, but there is considerable difference in the sensor and world models under which the various authors perform their pose estimation.

At the theory level, the primary work to which we wish to compare our localization algorithms is that of Guibas, et al ([GMR91]). They present the localization problem at a theoretical level: The input is a world polygon and a visibility polygon. The visibility polygon is a star-shaped polygon representing the output of a swept rangefinder. They describe an algorithm which preprocesses the world polygon so that a localization query (in the form of a visibility polygon) can be answered in optimal time, $O(m + \log n + A)$, where m and n are the number of vertices in the visibility polygon and the world polygon, respectively, and A is the size of the output. The output, in this case, is a list of points at which the visibility polygon can be made to match the world polygon (i. e., feasible poses). Preprocessing the world polygon takes time $O(n^5 \log n)$ (worst case – expected case is apparently more like $O(n^2 \log n)$). A single-shot query (with no preprocessing) can be answered in time $O(mn)$. The main *lacunae* of this result are (i) there is no attempt to deal with uncertainty, (ii) the preprocessing assumes a static world. If the world model changes at all, the preprocessing price must be repaid, and (iii) the model of a swept rangefinder as producing a visibility polygon is difficult to realize on a real robot.

Many authors have described localization algorithms that make use of *beacons* or *landmarks*. These are typically features of the environment which it is easy for the robot to recognize. *Active beacons* are objects in the environment that emit a signal which the robot can sense; *passive beacons* are natural features in the environment, such as corners, straight line segments, and vertical edges, that the robot has a good chance of identifying. [MR88] describes a localization system consisting of a directional infrared detector system and a set of beacons that emit modulated infrared signals. If the robot can detect the angles to three beacons at known locations, it can use trigonometry to solve for its own position. [Kle92] achieves a similar result using active sonar beacons – elapsed times between the receptions of chirps from the series of known-location emitters enable the robot to compute its location; in fact, [Kle92] uses an Iterated Extended Kalman Filter ([Jaz70]) to merge pose estimates based on the active beacons with that based on dead reckoning. [LDWC90] defines a *geometric beacon* as "a special class of target which can be reliably observed in successive sensor measurements (a beacon), and which can be accurately described in terms of a concise geometric parameterization." The beacons they use are typically walls, corners, and cylinders. They use a Kalman Filter to combine the "measured" location of nearby beacons with the "expected" location (based on odometry and the robot's map) to compute new pose estimates. [KK91] is an example of outdoor beacon-based navigation. Their beacons are such features as "peaks", "pits", "ridges", and "ravines"; these are appropriate to navigation in rough natural terrains.

Several attempts have been made to use computer vision to detect and locate beacons ([CC92], [AH91], and [RWA$^+$92]). [CC92] use an Extended Kalman Filter to estimate position from odometry, sonar data, and the location of visually detected landmarks. They select their landmarks by hand, choosing objects with strong vertical edges. This work is similar to that of [MR88] and [Kle92], except that the landmarks here are extracted from an image. They support their work with rudimentary experimental evidence. [AH91] presents an algorithm that determines both the correspondence between observed landmarks (vertical edges in the environment) and an *a priori* map, and the location of the robot from that correspondence. They detect and locate their visual landmarks using stereo vision techniques. They present results of extensive experimentation in an uncluttered (empty) interior environment, indicating that their system works very reliably in this context. [RWA+92] uses images and odometry in a very novel way: They compute *controlled hallucinations*, which are mockup 3D renderings of their robot's immediate environment, based on an *a priori* map and a dead-reckoning estimate of pose. They then use a matching algorithm to match real image edges to model edges, and update the pose estimate based on the model-to-image transformation.

Drumheller ([Dru87]) did some of the early work in sonar-based localization. He obtains a large number of sonar readings, each with the transducer rotated a few degrees from the previous, then takes the polygon formed by the endpoints of these sonar "rays", and extracts straight line segments from it. He then uses the *interpretation-tree*-based two-dimensional matching algorithm of Lozano-Pérez and Grimson ([GLP85]) and an *a priori* line-drawn map of the environment to produce *interpretations* (lists of possible (sonar segment/wall) pairings). Each interpretation corresponds to a pose for the robot. He then uses the *sonar barrier test*, which he defines as an additional constraint on sonar: that "an admissible robot configuration must not imply that any sonar ray penetrates a solid object." His algorithm returns the interpretation that passes the sonar barrier test and has the greatest amount of sonar contour in contact with the walls. [GSO92] uses a similar approach, except that they use maps made by the robot (both a map made of line segments and a map of cells), use a laser rangefinder to obtain their sensory data, and use an iterative algorithm to match their maps against their sensory data. [HMB92] use sonar data to perform localization: They extract "edge", "wall", "corner", and "unknown" features from this data, then find the transformation that maps each sensed feature onto a map feature of the same type. They then plot the transformations and look for clusters of similar transforms. This is similar to several matching algorithms based on the Hough Transform. See, for example, [SHD86].

Beveridge and Riseman ([BR91]) use a 3D, perspective-based computer vision matching algorithm to track a robot's progress as it navigates in hallways. Because of the physical and visual unclutteredness of hallway scenes, they are able to use the motion of detected edges to detect the robot's position and motion relative to the walls, corners, and doorways in its environment.

[MDW93] present a sonar sensor which is capable of tracking environment features. This sensor uses a non-colocated transmitter-receiver pair to track *regions of constant depth*. They use this sensor as part of a localization algorithm for a moving mobot in an indoor environment.

A related problem to that of robot self-localization is that of object localization by the robot. This has primarily been studied in the context of manipulator arms. [SKS88] provide a solution to the problem of determining the position and orientation of an object through measurement from multiple, heterogeneous sensors.

Moravec and Elfes introduced the *statistical occupancy grid* method of mapmaking in [Elf87,Mor89,ME85] as a way of making detailed geometric maps using noisy sensors (sonar rangefinders, in their case). A statistical occupancy grid is an array, where the contents of each cell is based on the confidence that the part of the environment corresponding to that cell is occupied. The main idea behind the statistical occupancy grid is that, since range data is inexact, the contents of each cell should be a probability: a cell containing a high probability is likely to be occupied, while a cell with low probability is likely empty. Moravec and Elfes use an update rule based on Bayesian probability to maintain the contents of each cell based on range readings that impinge upon that cell.

[SF93] examine the problem of mapmaking using cooperating mobile robots. They use a decentralized, on-line approach using cooperation and communication between robots to implement an occupancy-grid mapmaking algorithm. They present results from simulation.

Much research has been done on motion planning and execution for mobile robots. Takeda and Latombe ([TL92]), for example, approach the problem of motion planning with uncertainty by computing a *sensory uncertainty field* for each configuration q in the robot's configuration space. The sensory uncertainty field estimates the distribution of possible errors in the robot's pose estimate when it is in configuration q. This information is used by a path planner to generate paths that minimize expected errors; the paths so generated are easier to execute reliably. Ratering and Gini ([RG93]) describe a *hybrid potential field*, which combines a global, *a priori* potential field, generated from the robot's map with a local potential field generated from instantaneous sensory data. This hybrid potential field is designed to enable a mobile robot to execute a motion plan in the presence of unknown, moving obstacles.

1.2.2 Mobile Robot Systems

A truly vast amount has been done and written about designing and building mobile robots. Much of this work has concentrated on the design of a particular part of a mobile robot. For example, [KP92],[WA92], and [IRST92], among many others, have engaged in the design of various types of mobile platforms for purposes requiring the ability to move holonomically, or over rough terrain, or something similar. Many other researchers have designed sensors of various sorts for use on mobile robots. Except for certain works pertaining to optical rangefinding and special-purpose mobot vision, we won't discuss work concentrating on a specific platform or sensor, since we want primarily to discuss those related works which pertain directly to the idea of designing a mobot system that will be useful as a general mobot research vehicle.

HILARE ([GCV83]) was one of the early research mobile robots. HILARE was built on a three-wheeled platform, where the rear two wheels were stepper-motor driven and the front wheel was a passive wheel. It was equipped with a video camera, a laser rangefinder, an array of sonar sensors, an infrared sensor (used to determine position when the robot was in an area marked with fixed beacons), and shaft encoders on the rear wheels. Six microprocessors controlled the on-board functions. This network of processors was radio-linked to a minicomputer. HILARE's higher-level functions were written in a rule-based language coded in LISP. At the time (1979), HILARE was a highly sophisticated system which enabled its builders to do research on machine learning and mobile robot navigation, and appears to have remained in use at least as late as 1988 ([GRDSV89]), although it is unclear whether in original or modified form.

MARS ([KTB87]) was built in the mid-eighties at Stanford. MARS had a three-wheel base whose wheels were configured in an equilateral triangle; each wheel had six passive rollers around their circumferences, allowing them to be drive sideways by the other wheels; this was one of the early designs allowing for holonomic motion in a mobile robot. Computationally, MARS had a 16-bit microcomputer and three 8-bit microcontrollers; each of the 8-bit microcontrollers controlled one of the wheels. In addition, it had an array of sonars, an array of bumpers, and a servoable pan/tilt head bearing a camera hooked to a wireless transmitter. MARS built hierarchical maps that were graphs with nodes such as "building", "room", "wall", and "door". Off-board computation included a Symbolics LISP Machine and a VAX.

Blanche ([Cox88]), built in the late eighties, was an attempt at an inexpensive, flexible mobot. Blanche had a three-wheeled base also, but in this case, the front wheel is driven, and the rear wheels passive. Originally, Blanche only had two sensors: shaft encoders on the wheels and a laser rangefinder. This rangefinder also functioned as a barcode reader for beacons and a wide-band data link with an offboard computer. Blanche was later equipped with sonar sensors for research performed in [LDWC90].

HERMIES III ([RJB+91]), also built in the late eighties, was a much larger-scale attempt at a research mobot, in a number of ways. Physically, HERMIES III was much larger than most mobile robots: it weighed 1230Kg (with batteries) and measured 1.6m by 1.3m by 1.9m. It was also a computational heavyweight, with a 16-node NCUBE hypercube on an IBM 7532 host, and a VME backplane with 5 M68020 processors, all on-board. Off-board compute was provided by an SGI IRIS-4D and a Micro-Vax-II. All of these computers were connected by Ethernet. Sensors included were three CCD cameras, 32 sonars, a laser rangefinder, and shaft-encoders. HERMIES III had an extensive software system running on its various processors – for their demo task, DEMO89, they ran 19 different processes concurrently. This software was developed in cooperation by four different universities and Oak Ridge National Laboratory. HERMIES III apparently failed to perform to its builders' satisfaction, as they say "In general, our experience has been negative. Each component of the computer system that is different from the others introduces its own collection of idiosyncrasies and increases the time and effort required to build the system. . . . Thus, the use of heterogeneous systems reduces the sophistication of the solution."

MODRO ([VTGAS93]) has recently been developed by the Institute of Robotics, ETH. Zurich; MODRO is a modular mobile robot developed as a test platform for indoor applications. MODRO has a tricycle wheel configuration with suspension designed to allow it to traverse small discontinuities in the floor. It is equipped with two sets of bumpers, an ultrasonic "virtual bumper", and a LIDAR (Light Intensity Detection and Ranging) laser rangefinder. They use a modular, layered software system which uses, among other things, artificial neural networks, fuzzy logic, and a combination of the two, to implement its layers.

1.2.3 Optical Rangefinders and Special-Purpose Mobot Vision

A variety of ranging devices exist which incorporate laser beams. [Bes88] provides a survey of a large number of optical range imaging sensors of various types. He categorizes these sensors as using one or more of the following optical principles: radar, optical triangulation, moiré, holographic interferometry, focusing, and diffraction.

The survey provides background on optical ranging, and also gives some basis for comparison of different types of range sensors. We refer readers to [Bes88] for a much more comprehensive survey of optical rangefinders designed prior to 1988.

The longest-range optical rangefinders are laser radars. These rangers use essentially a time of flight measurement to obtain the range value. These rangers can be accurate over long ranges (tens of kilometers, in some cases), but have high cost, and, at least over longer ranges, require a reflector at the other end of the measured distance. For more details, see [Deu82].

Another popular laser ranging technique is the one known as *optical triangulation*, or "light-striping". With light striping, the process of acquiring a depth map of a scene consists of taking a number of scans which have the following form: A laser beam is passed through a cylindrical lens to transform it into a plane of light. This light strikes objects in the world, causing a linear contour. A camera located slightly away from the lens takes an image of the scene with the laser line in place. Image processing is used to extract the location of the line in the image, and triangulation is used to calculate depth. This is typically done several times at closely spaced intervals to obtain a contour of a three dimensional object; it could easily be done a single time with a horizontal plane of light to obtain a contour at a single height. Kanade presents a VLSI based light striping sensor in [KGC91] which relies on analog electronics and a very simple approach to looking for the laser beam to provide a fast and effective light striping range sensor.

Optical triangulation is the term used for laser rangefinders which employ a light projector and an imaging device separated by a known baseline to calculate range distance. These systems typically project a line or plane of light into the world and use a camera or other photodetector to detect the intersection of the light with the world. Since the baseline and relative angle between the laser and the camera is known, a simple trigonometric formula can be used to calculate depth from the position of the light in the image. Our ranger is a special case of optical triangulation, in which the image plane is perpendicular to the projected light. A typical difficulty with optical-triangulation systems is sensitivity to ambient light: since the ranger is looking for bright spots in the image, bright light from other sources can detract from the performance of the ranger.[4]

[BRD91] describes an optical-triangulation-based ranger which uses a camera with a double pinhole aperture. This aperture causes the image of the line pro-

[4] Our ranger is also susceptible to this problem. Since we designed our ranger for indoor use, we were able to solve this problem adequately using an optical filter to eliminate light not in the same frequency range as our laser.

jected by the laser to appear twice in the image, at separations which are related to distance. This system apparently gives robust performance with respect to performance under high ambient light conditions, due to the fact that any line appearing in the image which cannot be matched up with another, similar line coming from the other pinhole, can be treated as spurious data and disregarded. Another optical-triangulation ranger which operates reliably under high ambient lighting conditions is that described in [YH88]. This ranger achieves high accuracy over a 1-5m range under direct sunlight conditions using a dual-camera system. Each camera's image is processed using what they call "dual signal extraction with delay and difference"; this eliminates most of the ambient light (the constant-level ambient light). They then use the two processed images to eliminate what is left (typically transients and pulses from other lighting sources).

Most optical triangulation rangefinders use off-the-shelf CCD video cameras. This necessitates designing electronics to process and digitize an RS170 video signal. [IDD90] avoids this by using an Optic RAM. An Optic RAM is a standard dynamic RAM equipped with a transparent window over the die; they use a 64 kilobit RAM. For each image frame, each memory cell is a single bit which is set if the light falling on that cell is over a certain threshold. Since the RAM can be read directly (like any other memory), any digitizing and thresholding operations that would be needed with a CCD camera are eliminated. The threshold is controlled by the exposure time.

[PP93] describes a laser ranger which does not use a video camera at all. This system, designed at Oxford University, uses a Lateral Effect Photodiode (LEP) and a laser diode to perform optical triangulation. The LEP returns signals from which the lateral position of the centroid of the light hitting it can be extracted; by modulating the outgoing laser and filtering the LEP signals, an accurate estimate of the distance to the nearest object can be obtained. Using mirrors, they can scan at 10 Hz and obtain 256 range readings per scan over a range from 0.4m to 2.5m.

Cameras can also be used to determine depth passively, using only the ambient light in the environment. The most common method for doing this is stereo vision, where matching of features in the left and right images enables computation of depth. General stereo is not very useful for mobile robots, because it is an expensive procedure and doesn't work very well. Some success has been achieved using stereo under restricted circumstances or in combination with other techniques. For example, [KK88] achieves high accuracy depth perception using stereo together with servoed focus control; the two depth-computation methods serve as checks for each other. [IST90] uses a single camera to compute 3D structure by feature tracking:

they automatically select a feature in the camera's image and track that feature from frame to frame as the robot moves. They compute depth from the feature's motion, and build a local map of the area around that feature.

Another line of special purpose mobot vision research pursues applications of panoramic vision. The idea of panoramic vision is to obtain images which are topologically cylindrical, and which provide the mobot with a 360-degree view of its environment. [Sar89] obtains a cylindrical picture by extracting the central column from each of a set of images taken as the robot pans around a full circle; she uses this information to detect features such as walls and doors. [IYT91] and [HTP$^+$91] place a spherical mirror above an upward-facing camera to obtain a panoramic view in a single image. [YNY92] use a similar tactic, but instead use a conical mirror. One of the advantages of this approach is that the intersections of nearby obstacles with the floor are easy to detect; several of these researchers use qualitative analysis of these panoramic images to perform navigation.

[Hor93] subsamples images from inexpensive, low-resolution cameras to resolutions of 64×48 and 16×12 to perform qualitative navigation and scene recognition computations on his robot, Polly. Using a TMS320-based DSP system, he is able to navigate an interior environment at roughly walking speeds.

1.2.4 Sensor Design and Analysis

Some researchers have looked at the problems of modeling sensor characteristics and sensor confidence and designing sensor specifications for given problems. Donald and Jennings, in [DJ91] present the concept of *perceptual equivalence classes*, which allow them to partition the robot's world into sets of configurations that the robot cannot tell apart. They discuss the idea of building *virtual sensors* that sense information about the robot's state by building and evaluating predicates from lower-level information provided by other sensors, both virtual and concrete. Hughes and Ranganathan ([HR93]) consider the problem of determining the reliability of individual sensors in a multi-sensor robotic system in an unknown environment; they provide a model for assessing sensor reliability in a multi-sensor system by determining a measure of confidence from comparisons of data from the different sensors. Erdmann, in [Erd93], attacks the sensor requirements/sensor characteristics issue from the other end, by providing a framework for designing sensor specifications from task specifications.

1.2.5 Cooperative Mobot Manipulation

Cooperative mobot manipulation, as a research area, stems from two earlier research areas, robotic manipulation and cooperating robots. Our cooperative mobot manipulation research draws directly from both areas.

[MS85] provides extensive mathematical analysis of grasping and pushing operations under a quasistatic model. Mason analyzes the behavior of a number of different manipulator/object contact modes, paying attention to physical properties such as the center-of-friction of the object and friction between the object and the manipulator. Since it is assumed throughout that the manipulator is an arm, whose mass is much larger than that of the object, it is assumed that the dynamics of the object is negligible. Along these same lines, [Bro85] and [RG92] have analyzed the geometry and mechanics of quasi-static pushing and squeeze-grasping of planar objects with parallel jaw grippers. Balorda, in [Bal93], presents an algorithm for automatic planning of robot pushing operations. His goal is to acquire a planar object with a two-finger gripper and bring the object into a specified configuration; his algorithm produces a plan from its inputs, which are the part description and initial configuration and error bounds on the initial configuration.

A number of authors have begun exploring the issues involved in getting multiple mobile robots to cooperate, with some sort of manipulation task as an eventual goal. [SB93] look at the problem of manipulating pallets using many small robots; they present the results of their simulations. [Mat93] presents theory and experimental results that study group behaviors such as aggregation, dispersion,and flocking. [ABN93] examine the relative performance of simulated foraging agents with and without communication.

Some work has been done on large-scale manipulation using a single mobile robot. [OY92] built a mobot with a hinged plate at the end of a fixed arm for pushing operations. The plate was chosen to minimize sliding contact. They analyzed the mechanics of their robot pushing a box and synthesized a control law for moving a box to a goal location. [Lyn92] and [LM94] analyze the mechanics of planar pushing with line contact (a robot with a fixed flat blade pusher) and design a path planner which maintains pushing contact. The paths produced by this planner specify stable pushing motions (those which maintain contact with the box), but also specify contact changes, in which the robot breaks contact with the box and repositions to another face, permitting sudden changes in the direction of pushing.

Cameron, et al ([CMW+93]), and Yamamoto and Yun ([YY93]) have examined the problem of controlling mobile manipulators (i. e., mobots equipped with manipulator arms). [CMW+93] present tools for modeling integrated arm-vehicle kinematics and dynamics, and presents simulation results for the reactive control algorithms they describe. [YY93] present a control algorithm for a mobile manipulator following an acquired object; their approach is applicable, for example, to cooperating mobile manipulators transporting a common object (in their case, the common object is picked up off of the floor, rather than manipulated while in contact with the floor, as in our case). They present experimental results where a human guides their robot's manipulator while the platform works to maintain a desirable contact mode.

The main works which have combined cooperation with mobot manipulation are [Nor93], [Par94], and [DJR94a,DJR93,Jen95]. Each of these works demonstrates the manipulation of a box or other large object using two mobile robots. [Nor93] uses cooperation to achieve the task, but not for the actual manipulation: His demonstration task uses one robot to push the target box and another robot which makes sure that the first robot's task is possible by pushing obstacle boxes out of the way. Separating the task in this way allows the robots to do less overall motion, but does not, apparently, make possible any manipulation task which is not possible for a mobot acting alone. [Par94] describes an architecture (L-ALLIANCE) designed to achieve fault-tolerant cooperation within teams of heterogeneous mobile robots and applies that architecture to a number of tasks (some in simulation) including "hazardous waste cleanup", in which several mobots cooperate to pick up a number of pucks and take them to a designated site, and a box-pushing task. The puck cleanup task demonstrates the speedup achievable using multiple robots, but that speedup is sublinear: m robots cleaning up n pucks will spend more than $\frac{1}{m}$ times the time required for one robot to clean up n pucks. Note that for computation problems, sublinear speedup is accepted as a given, but this is not true for manipulation tasks. Consider, for example, two people moving a stack of long boards. One person can only carry one or two boards at a time, but two people working together can carry many boards at once (with one person at each end, the rotational freedom of the boards is constrained more completely, so the strength of the lifters can be applied almost entirely to lifting). The box pushing task is broken into a push-left operation and a push-right operation, which are distributed over the two pushing robots: Each robot pushes its end, then waits for the other robot to push, then pushes again, until the task is accomplished. It is claimed that this protocol can be executed by a single robot in the event of failure of the second (the remaining robot executes its push, eventually decides that the other robot has timed out, moves to the other end of the box, and pretends to be the other robot), but this will only work for some boxes.

Chapter 2

The Mobile Robot Localization Algorithm

2.1 Introduction

Localization is the process of determining the robot's location within the environment. More precisely, it is a procedure that takes as input a map, an estimate of the robot's current pose, and a set of sensor readings (any of these may be incomplete, and all are toleranced by error bounds), and produces as output a new estimate of the robot's current pose. By *pose*, we mean either (a), the position and orientation of the robot in the world or (b), the translation and rotation necessary to make a set of sensor readings (or a local map, built from those readings) best match an *a priori*, global map. In this chapter, we will describe in detail our approach to localization, culminating in a family of algorithms for localization by mobile robots.

Before diving into formal, mathematical descriptions of localization algorithms, let's define and "type" some of the terms and items we will be using in this chapter. When we refer to a *map* in this chapter, we mean an *a priori* description of the environment, which will typically be either a collection of polygons supplied as lists of edges or a bitmap (a two-dimensional binary array). A *range probe* or *range vector* is a vector whose length is the value returned by a rangefinder (a laser rangefinder or other point-and-shoot distance measuring sensor, typically) and whose orientation is the rangefinder's heading relative to the robot's coordinate frame.

Why do we need a localization algorithm? In the domain of robotic manipulator arms, position sensing is a sensing primitive — it is typically possible to get a position reading that is within some (small) error bound of the actual position of the manipulator. Furthermore, the error bound is a constant; it does not grow over time and applies over all configurations of the arm. With mobile robots, this assumption does not hold; odometry from wheel shaft encoders is fraught with error. One method for tracking the position of a moving object is *dead reckoning*: starting from a known position and computing current position by integrating velocity over time. This works poorly in practice: control and sensing uncertainty integrated over time make the dead reckoning estimate of position extremely inaccurate. Some work has been done toward developing localization techniques and algorithms that operate by comparing geometric representations of the robot's environment with instantaneous sensory data, in such a way as to provide estimates of the robot's pose within its map. One recent computational geometry algorithm provides a theoretical approach to robot localization that illustrates this technique. Guibas, et al [GMR91], present the localization problem as a matching problem between a world polygon and a visibility polygon. The visibility polygon represents the output of a swept rangefinder. This algorithm provides a list of poses within the world polygon from which the view is consistent with the range-finder polygon. The two main *lacunæ* of this result are (i) there is no attempt to deal with uncertainty, and (ii) this result assumes a static environment—the algorithm is optimized with preprocessing to allow quick processing of queries. Note that these difficulties are not insurmountable, since the algorithm may well be adapted to compensate for these problems, but these extensions have yet to be done. A more fundamental difficulty with this algorithm is that, in general, obtaining the visibility polygon, as a list of vertices and edges, is extremely difficult with any physically realizable sensor.

The main concept underlying the localization algorithms presented in this thesis is that of the *feasible pose*. A feasible pose is one that is consistent with the available range and map information: our algorithms partition the robot's configuration space into poses that conflict with available range information and poses that do not; these latter are the feasible poses. Poses are feasible or infeasible relative to several parameters: a pose is feasible with respect to a map and a range vector if that pose places the robot such that the range vector terminates at an obstacle boundary and is otherwise unobstructed. A pose is feasible with respect to a map and a *set* of range vectors if it is feasible with the map and each of those range vectors, individually. We can determine feasibility within an error bound: to do this, we allow the length of a range vector to vary within an uncertainty bound and determine the poses that are consistent with some pose plus-or-minus the uncertainty bound. We can extend limited-error-bound feasibility to multiple range readings. We can also determine the minimum error bound for which there exists a feasible pose. Finally, with multiple

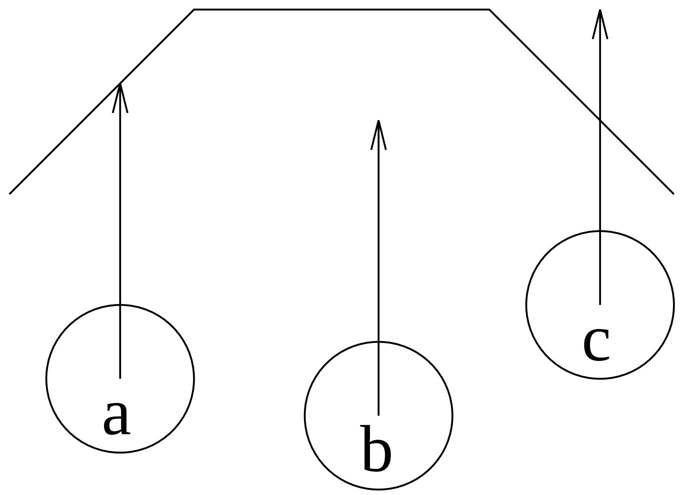


Figure 2.1: A range probe is a vector with its tail at the robot and its head at an obstacle boundary. Pose (a) is feasible under the shown range vector, but poses (b) and (c) are not.

range readings, we can count how many range readings are feasible with a given pose and map, and look for poses that are feasible with a maximal number of the supplied range readings.

2.2 Localization Algorithms

The main results for this section comprise algorithms to compute *feasible pose sets.* A feasible pose set is the set of all poses in the robot's configuration space, C, which are feasible with the given inputs. Primarily, we consider the case where C is $\mathbb{R}^2$, and the robot has perfect orientation sensing, but no position sensing. We also consider an $\mathbb{R}^2 \times S^1$ configuration space, where the robot has no knowledge or only limited knowledge about its orientation with respect to the world. In this section, we will foreground combinatorially precise computational geometric algorithms; we will discuss versions of these algorithms that can be made to run efficiently in section 2.3.

2.2.1 Definition of the Localization Operation: Computing Feasible Poses

Let us represent a range reading as a vector with the tail at the robot. In order for a given position within a geometric map to be consistent with a particular range reading, the following statement must be true: If we place the tail of the range vector at the given position, then (i), The head of the range vector must lie on an obstacle but (ii), The body of the vector must not intersect any obstacles. This comes from the following interpretatation of range reading:"a reading of 57cm in direction θ means that there is nothing *within* 57cm of the rangefinder in direction θ, but there is something *at* 57cm." Possible positions of the robot, given the positions of the obstacles, meet the following condition: any place the tail of the vector can be positioned such that the head of the vector intersects an obstacle but no part of the body of the vector lies on an obstacle, represents a place where the robot can be and get that range reading. Figure 2.1 demonstrates this interpretation. Given a range vector z, then, we can compute the possible locations of the robot. Before we can define feasible pose precisely, we need a few definitions: The Minkowski sum of two sets, $X \oplus Y$ (as described in [LP83]) is the vector sum of all points in X with all points in Y:

$$X \oplus Y = \{x + y \mid x \in X,\, y \in Y\}.$$

$X \ominus Y$ is the vector sum of X with the 180 degree rotation of Y:

$$X \ominus Y = \{x - y \mid x \in X,\, y \in Y\}.$$

$\oplus$ is also called *convolution*; the Minkowski sum is mathematically closely coupled with the convolution integral used in signal analysis. If x and y are points, $\ell(x, y)$ denotes the open segment between (but not including) x and y. Let ∂X denotes the boundary of the set X. Let M be a polygon representing the free space in the robot's map. Let z be the vector representing a given range probe. We can express the feasible poses for model M and probe z:

$$FP(M, z) = \partial(M \ominus z) - (M \ominus \ell(0, z)) \tag{2.1}$$

where "$-$"denotes set difference. In fact, we can dispense with the ∂, because the interior of $M \ominus z$ is contained in $M \ominus \ell(0, z)$. This gives us

$$FP(M, z) = (M \ominus z) - (M \ominus \ell(0, z)) \tag{2.2}$$

which is an equivalent definition, but that extends more easily to the case with uncertainty. For purposes of explication, we define two more sets, $D(M, z)$ and $E(M, z)$:

$$D(M, z) = M \ominus z$$

$$E(M, z) = M \ominus \ell(0, z)$$

Since $FP(M, z) = D(M, z) - E(M, z)$, $D(M, z)$ is, in some sense, the set of points "added in" by the head of a given range probe, while $E(M, z)$ is the set of points "subtracted out" by its body. Figure 2.2 illustrates an example calculation of $FP(M, z)$. Precise localization will, in general, take multiple range probes. Given a set of range probes Z, we define

$$FP(M, Z) = \bigcap_{z \in Z} FP(M, z) \tag{2.3}$$

We deal with translational uncertainty by one of two mechanisms: the *consistent reading* method and the *varying epsilon* method. The consistent reading method says, "if $FP(M, Z)$ is empty (there are no points in the intersection of the results of all range readings), what is the set of all poses that are contained in a maximal number of $FP(M, z)$, for $z \in Z$?" In other words, if there are 24 range readings taken (say, one every 15 degrees), and no point in the map is in all 24 feasible pose sets, then look for the set of poses contained in any 23 of the sets, or any 22, and so on (this can be done without increased complexity, as explained below). This introduces a quality measure to the result: a point that is contained in all 24 point sets is a better match than one that is only contained in 23, or 22. This tactic enables the algorithm to be robust in the presence of bad range readings. It also can handle the case where a movable obstacle in the robot's vicinity occludes a stationary obstacle, causing one or more range readings to be different than expected (see, for example, figure 2.3). This idea can be formalized as follows: If a set X is a subset of ambient space $\mathcal{U}$, then the *characteristic function* for X is a function $\chi_X : \mathcal{U} \to \{0, 1\}$ such that, for all $u \in \mathcal{U}$,

$$\chi_X(u) = \begin{cases} 1 & \text{if } u \in X \\ 0 & \text{otherwise.} \end{cases}$$

Using this notation, the function that maps a point $p \in \mathrm{C}$ onto $\{0, 1\}$ based on whether $m \in FP(M, z)$ is $\chi_{FP(M,z)}$ Given a set of m range readings, Z, we define $\mathrm{rnk}(M, Z, p)$ as the function from C to $\{0, 1, \ldots, m\}$ that, for each $p \in \mathrm{C}$, counts how many $FP(M, z)$ contain it. Precisely,

$$\forall p \in C, \mathrm{rnk}(M, Z, p) = \sum_{z \in Z} \chi_{FP(M,z)}(p)$$

We can now define the set of points in C that are contained in a maximal number of $FP(M, z)$. Let $\kappa = \sup_{p \in C}(M, Z, \cdot)$. Then define

$$\mathrm{loc}(M, Z) = \{p \mid \mathrm{rnk}(M, Z, p) = \kappa\}.$$

The varying-epsilon method for coping with uncertainty uses epsilon balls (we refer to an uncertainty ball about the origin with radius ϵ as B_ϵ. B_ϵ will be an L_2

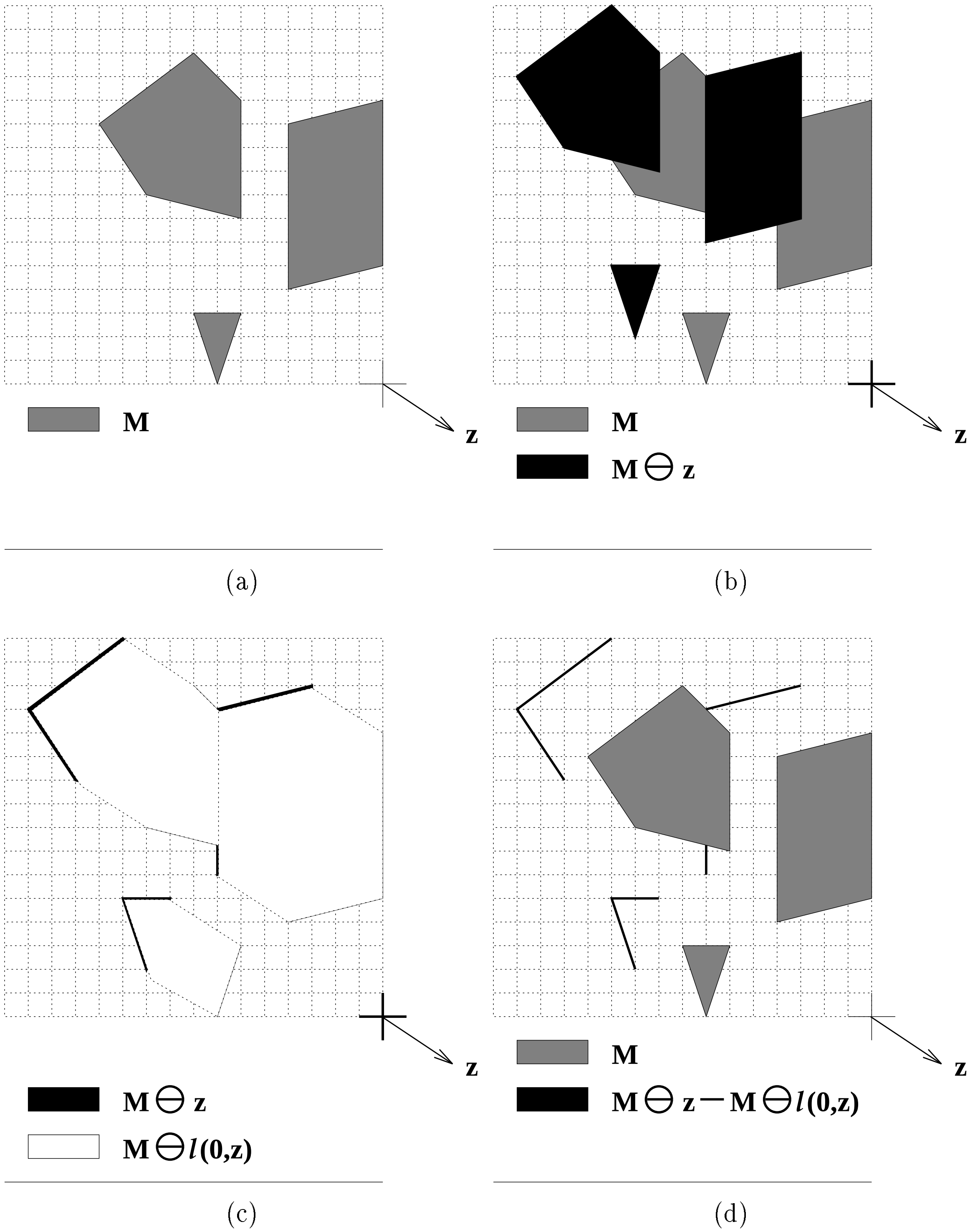
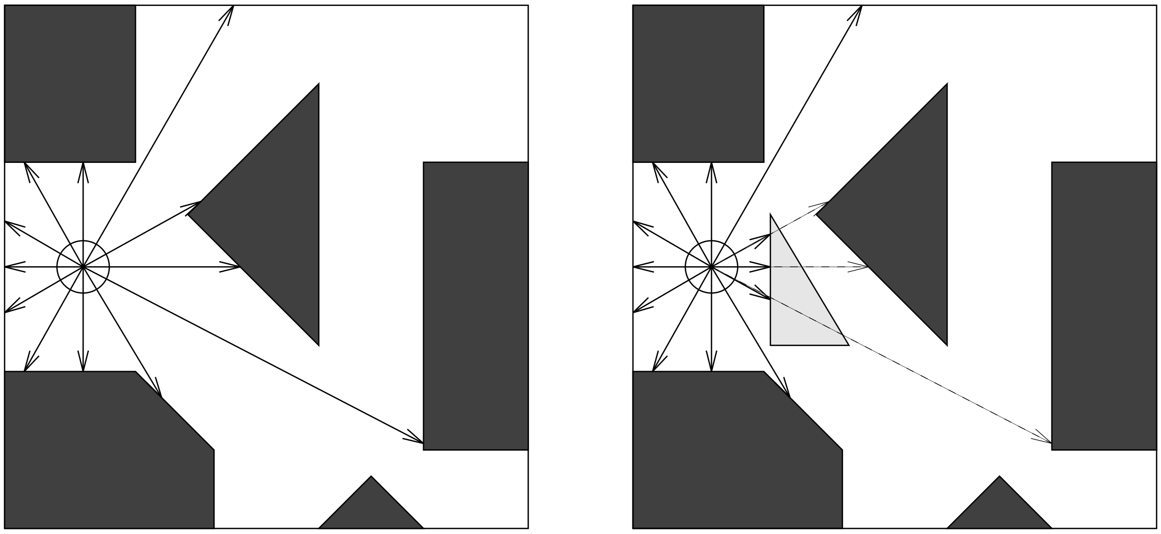


Figure 2.2: (a). A set of polygons, M, and a range vector z. (b). M with $D(M,z)$ overlaid. (c). $D(M,z)$ with $E(M,z)$ overlaid. (d). $FP(M,z) = D(M,z) - E(M,z)$ shown with the original M.



a b

Figure 2.3: The varying epsilon method can handle new obstacles: if the robot (the circle) has a feasible pose under the twelve range probes shown in (a), it will still have a feasible pose meeting nine of the twelve range probes, when three of the range probes are changed by the introduction of a new obstacle.

---

disk (Euclidean distance metric) unless otherwise specified). Given a map, M, and a range reading $z \in \mathbb{R}^2$, we want all poses p where $p \oplus z \oplus B_\epsilon$ intersects an obstacle, but the portion of $\ell(p, p \oplus z)$ lying outside $p \oplus z \oplus B_\epsilon$ does not. We denote the feasible poses for M and z with an uncertainty ball of radius ϵ by $FP_\epsilon(M, z)$:

$$FP_\epsilon(M, z) = (M \ominus (z \oplus B_\epsilon)) - \left(M \ominus \ell\Big(0, z\Big(1 - \frac{\epsilon}{\|z\|}\Big)\Big)\right). \qquad (2.4)$$

Again, we define the "additive" and "subtractive" parts of the formula:

$$D_\epsilon(M, z) = M \ominus z \oplus B_\epsilon$$

$$E_\epsilon(M, z) = M \ominus \ell\Big(0, z\Big(1 - \frac{\epsilon}{\|z\|}\Big)\Big)$$

so that $FP_\epsilon(M, z) = D_\epsilon(M, z) - E_\epsilon(M, z)$. We can now treat the range reading as having the form, say, "57cm plus or minus 2cm". While $FP(M, z)$ are typically sets of line segments, $FP_\epsilon(M, z)$ is typically a set of 2D "bands" that are, in effect, the line segments "grown" by ϵ. The intersection of two or more $FP_\epsilon(M, z_i)$ is also now typically a set of 2D regions, rather than a set of points or segments. We now measure the quality of a match by saying, "how large do we need to make ϵ in order to find a point that is in all of the feasible pose sets?" We can also combine the

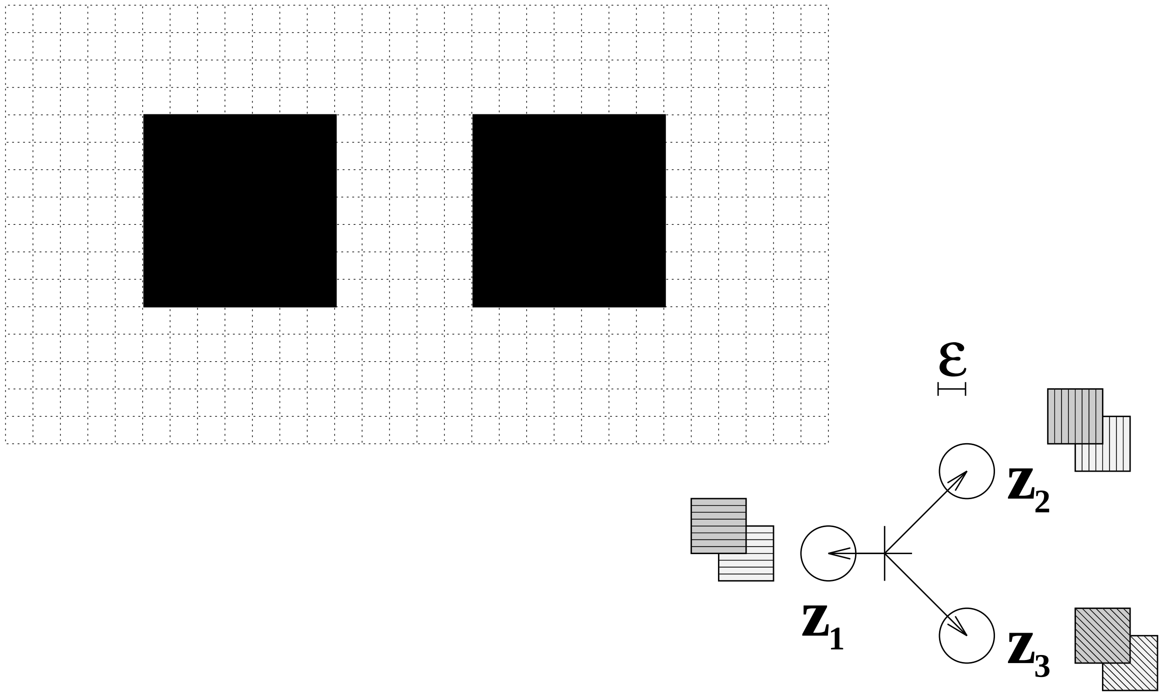


Figure 2.4: A localization example: A map (the black squares), three range probes ($z1$, $z2$, and $z3$), and an uncertainty ball of radius ϵ. For figures 2.4 through 2.8, the small, partially occluded grey squares key the shading of $D(M, z_i)$. The darker, unoccluded grey squares key the shading of $E(M, z_i)$.

---

consistent reading method and the varying epsilon method by asking, “How large do we need to make epsilon so that there exists a point that is in, say, 90% of the feasible pose sets?” (If we have m range probes, what is the minimum epsilon such that there exists a point that is contained within $\geq 0.9m$ of the $FP_\epsilon(M, z)$?) Figures 2.4 through 2.8 show an example of localization with three range probes. The first figure depicts the map, M and the three uncertain range probes, z_1 through z_3; the next three show the D_ϵ and E_ϵ sets for the three range probes. The fifth figure shows $FP_\epsilon(M, Z)$, the region that is consistent with all three range probes, overlaid with the original M and the three D_ϵ.

2.2.2 Exact Algorithms for Localization in $\mathbb{R}^2$

Given localization operations that we have just defined, we can now define algorithms to compute the various feasible pose sets. Our first set of algorithms are exact combinatorial algorithms to perform localization when the map is a set of polygons in the plane. Initially, we define the geometric framework for these exact algorithms and provide a rough sketch of a brute-force algorithm to perform the

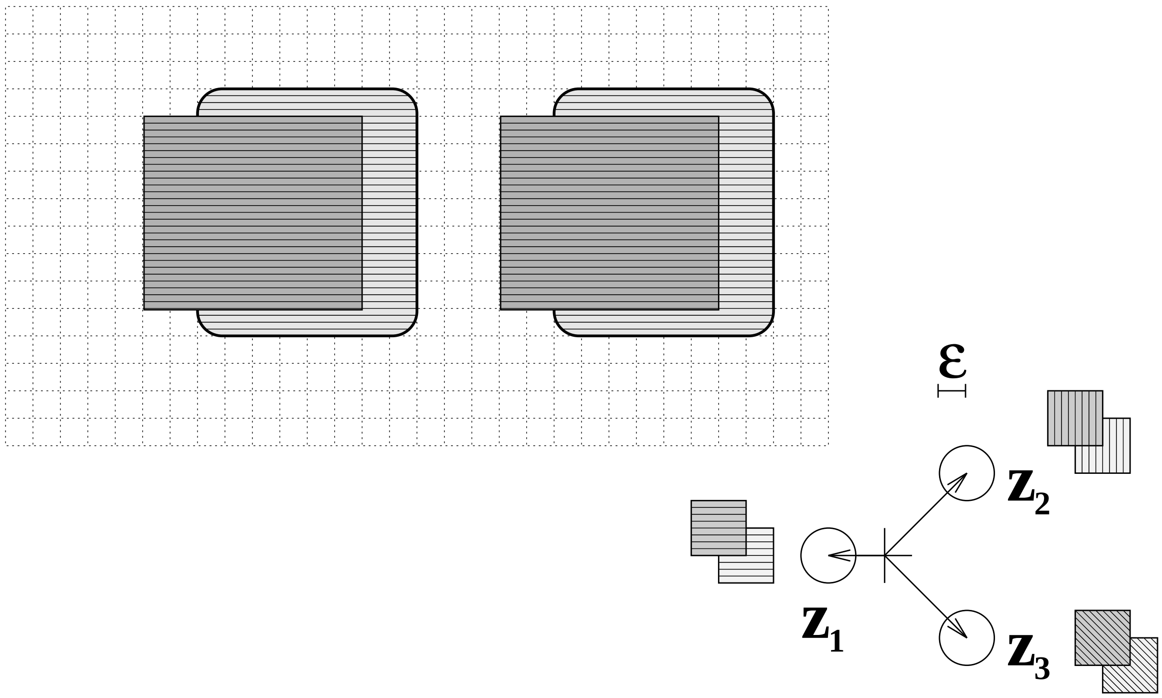


Figure 2.5: $D(M, z_1)$ and $E(M, z_1)$

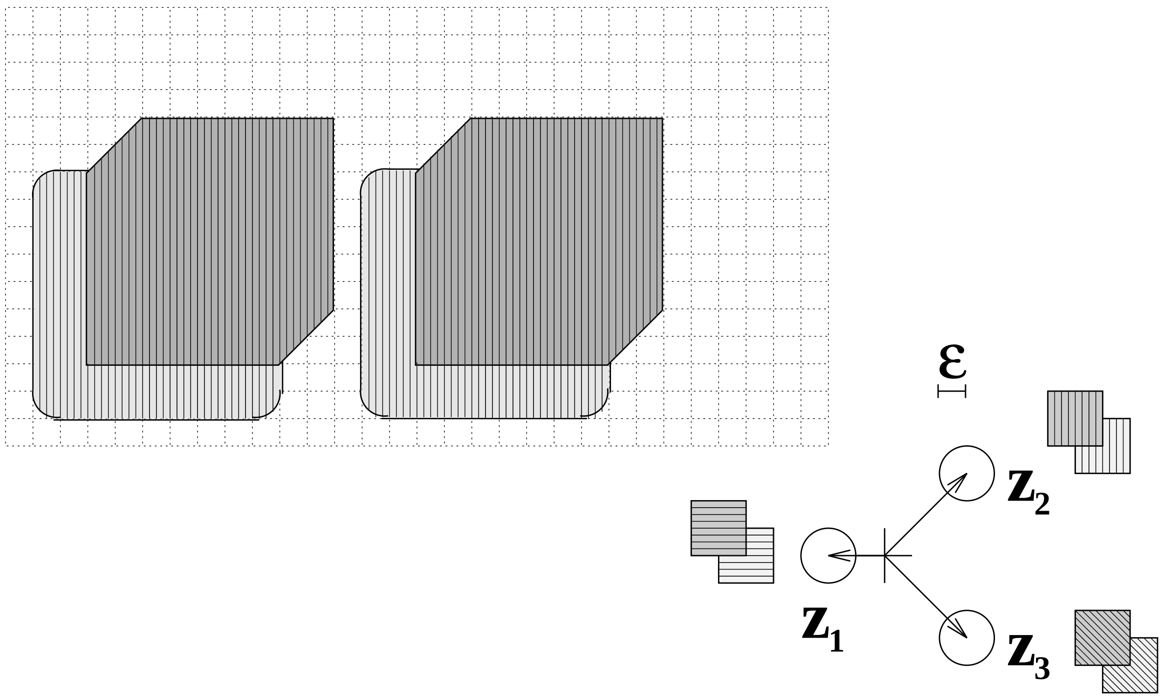


Figure 2.6: $D(M, z_2)$ and $E(M, z_2)$

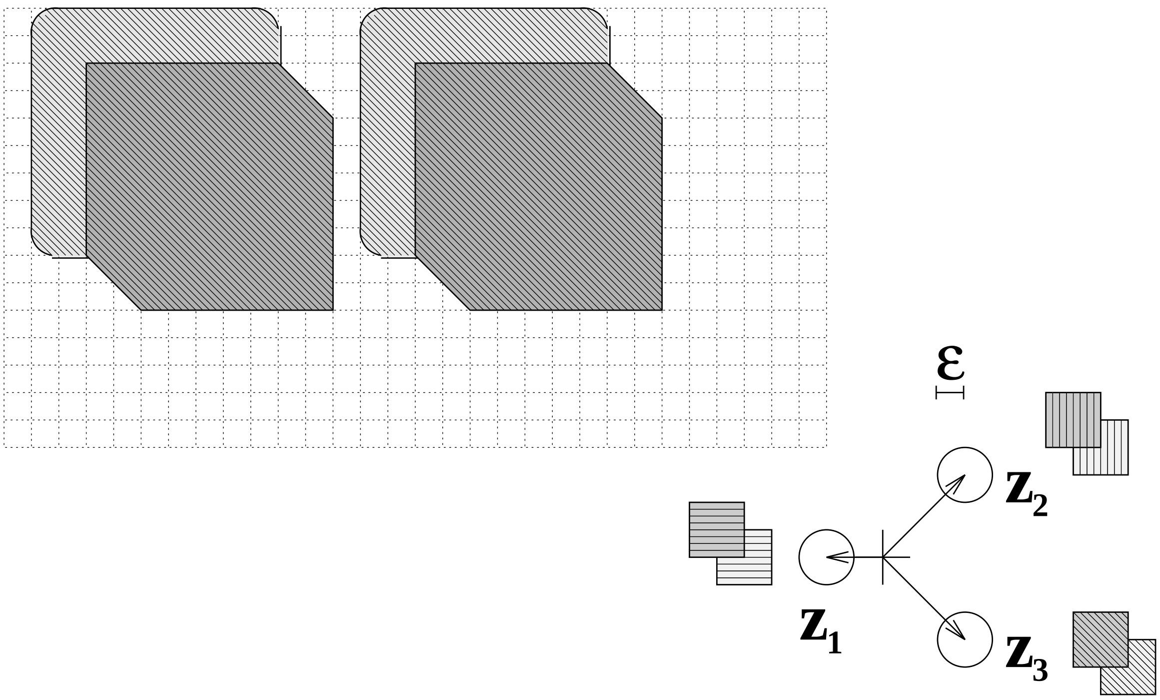


Figure 2.7: $D(M, z_3)$ and $E(M, z_3)$

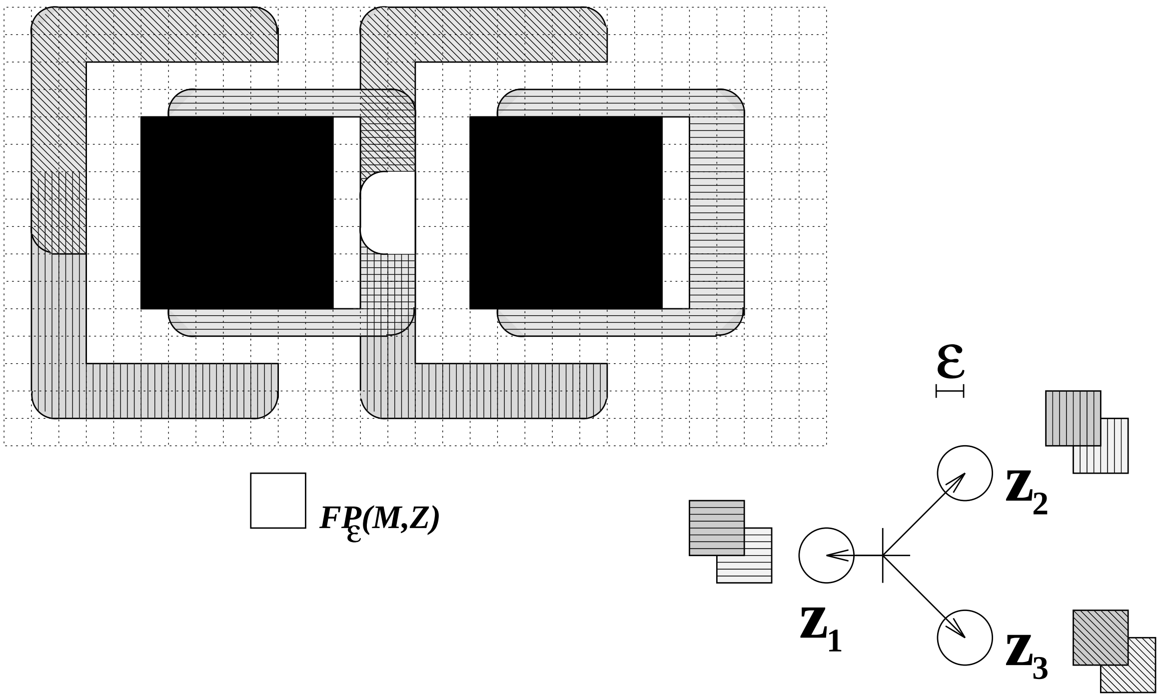


Figure 2.8: Here, we show the three $FP(M, z_i) = D(M, z_i) - E(M, z_i)$ overlaid. The white rectangle with two curved corners in the center is the intersection, $FP(M, \{z1, z2, z3\})$.

localization operation. We then describe a family of geometric algorithms known as "plane-sweep" algorithms that have better asymptotic behavior for set intersection problems than the brute-force method, and then use the plane-sweep for localization.

Equation 2.2 said that for map M and probe z:

$$FP(M, z) = M \ominus z - M \ominus \ell(0, z).$$

This is the set-theoretic definition of the feasible pose set for a given map and probe. We can develop an algorithm for computing feasible pose sets directly from this. Suppose M is a set of convex polygons with n vertices and edges. We can compute $D = M \ominus z$ in time $O(n)$. We can also compute $E = M \ominus \ell(0, z)$ in linear time: [LP83] showed that $B \ominus A$, for polygons B and A with b and a vertices, respectively, can be computed in time $\Theta(a + b)$. This time bound is achieved by building the result set directly by merging the edge lists of A and B in order of increasing orientation relative to the coordinate system. If we treat the closure of $\ell(0, z)$ as the boundary of a two-sided polygon, then we can apply this result to our problem. Note that D is simply M translated by $-z$, and is therefore still a set of polygons. E is a set of polygonal regions, but each region is an open set. Taking the set difference can be done in a naïve fashion in $O(n^2)$ time by taking the set difference between each region P_+ in D and each region P_- in E, as this is exactly $FP(M, z)$. Note that it is important in the exact case that we treat the regions in E as open, since the boundary of a region in E includes the boundary of the corresponding region in D (this follows from the definition), so if we don't treat the E objects as open sets, we will obtain no feasible poses.

We can adapt this algorithm to the varying-epsilon method in a straightforward fashion: D_ϵ is computed by convolving M with $z \oplus B_\epsilon$, E_ϵ by convolving M with $\ell\left(0, z\left(1 - \frac{\epsilon}{\|z\|}\right)\right)$, as per equation (2.4). The cost of computing D_ϵ and E_ϵ is still linear. The brute-force algorithm we just described for the zero-uncertainty case can still be used here, except for two changes. First, we no longer have to worry about the boundaries of the E objects, and can treat all regions in both D and E as compact. Second, if our uncertainty region is an L_2 disk, then we have to allow for circular arcs in the boundaries of our objects. While this creates additional implementation details, it has no effect on asymptotic cost.

For a set Z of m different range probes, we can compute $FP_\epsilon(M, Z)$ most straightforwardly by computing each $FP_\epsilon(M, z)$ and computing the intersection of the m sets. In general, a set of k segments and circular arcs in the plane can have $O(k^2)$ intersections. Thus, if M has n edges, each $FP_\epsilon(M, z)$ has complexity $O(n^2)$, so the union of all of the $FP_\epsilon(M, z)$ has complexity $O(mn^2)$. Hence, we can compute the intersections of the union in time $O(n^4m^2)$. A property of each individual $FP(M, z)$, however, allows us a tighter bound:

Lemma 2.1 *Suppose Z contains m range probes. Let U be the union*

$$\bigcup_{z \in Z} FP(M, z).$$

Then (i) the complexity of U is $O(mn^2)$, and (ii) the number of intersections between elements of U is $O(m^2n^2)$.

Proof. We have just shown *(i)*: if M has n edges, each $FP_\epsilon(M, z)$ has complexity $O(n^2)$, so the union of all of the $FP_\epsilon(M, z)$ has complexity $O(mn^2)$. To see *(ii)*: although there are $O(n^2)$ segments in an individual $FP(M, z)$, they all lie on $\partial(D(M, z))$, which is just a translation of M. The segments, therefore, lie on $O(n)$ lines. In addition, these segments do not overlap. Thus, there can only be $O(n^2)$ intersections between them. Analogously, when we take the union of m $FP(M, z)$, the $O(mn^2)$ segments lie on $O(mn)$ lines. U, therefore, can only have $O(m^2n^2)$ intersections. $\square$

In fact, these bounds are tight:

Lemma 2.2 *(i) The complexity of U is $\Theta(mn^2)$, and (ii) the number of intersections between elements of U is $\Theta(m^2n^2)$.*

Proof. (i): Figure 2.9 shows a set of polygons, M, which has n edges. It also shows $FP(M, z)$ for a sample range probe, z. If M has $n/8$ vertical rectangles and $n/8$ horizontal rectangles, there are $\Omega(n^2)$ of the open squares between the rectangles. Each open square contains part of $FP(M, z)$, so $FP(M, z)$ must have size $\Omega(n^2)$. *(ii)*:if we have m different range probes, $\{z\}$, with each z shorter than the size of those open squares, each $FP(M, z)$ will have that same complexity. U, therefore, for this M and Z, has $\Omega(mn^2)$ edges. Furthermore, we can select the $\{z\}$ such that each $FP(M, z)$ intersects every other within each of the open squares. In that case, each square contains $\Omega(m^2)$ intersections, leading to $\Omega(m^2n^2)$ intersections overall. Lemma 2.1 gives us upper bounds equal to these lower bounds, allowing us to conclude that these bounds are Θ-tight. $\square$

Observation 2.3 *We have not required that the polygons in M be disjoint. If we make that restriction, we conjecture that the bound for the complexity of U becomes $\Theta(mn)$. The bound on number of intersections remains unchanged at $\Theta(m^2n^2)$.*

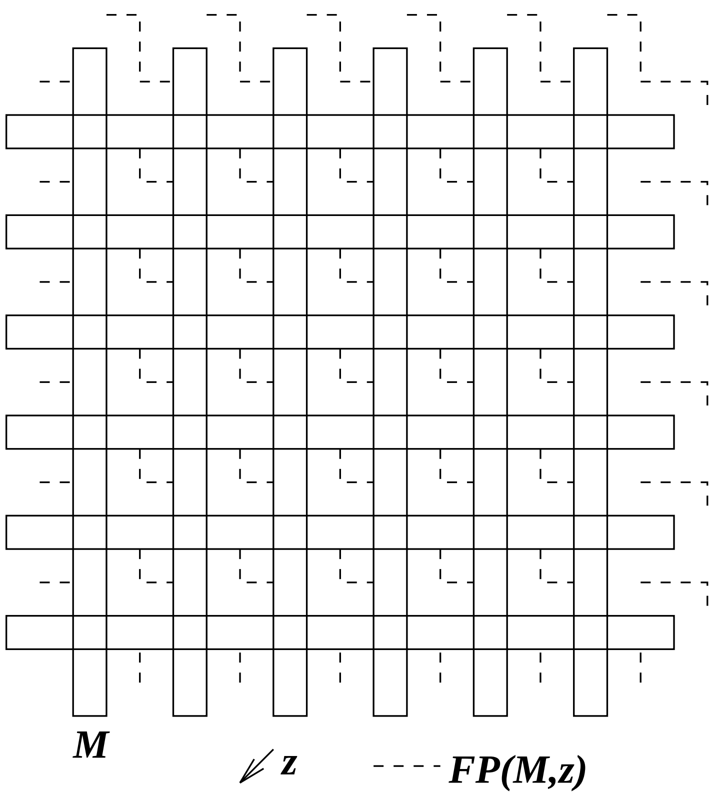


Figure 2.9: A lower bound: M has n edges, but $FP(M,z)$ has $O(n^2)$ components.

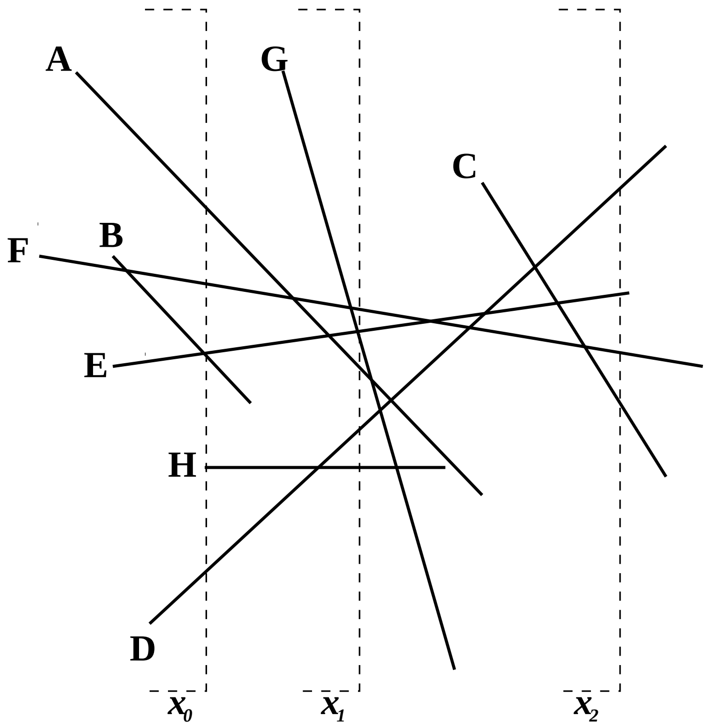
2.2.2.1 The Plane Sweep

We will now look at the family of algorithms known as "plane sweeps". The following discussion is a condensed treatment of material found in, among other places, [PS85], [NP82], and [Lat92]. Suppose we have a set of n line segments in the plane, and we want to determine all of the intersections between them. The obvious way to do this is to go through all $(n)(n-1)/2$ pairs of segments, determine whether they intersect, and if so, where, and output all of the intersections. The cost of doing this is $O(n^2)$, as it takes constant time to process each pair. But this cost is completely independent of the actual number of intersections; we'd like an algorithm that is cheaper than $O(n^2)$ if there are significantly fewer than $O(n^2)$ intersections.

The plane sweep segment-intersection algorithm meets that cost condition. The idea behind this algorithm is that we have a vertical line that we sweep across the plane, stopping whenever we encounter a significant event. We keep two data structures, the sweep-line and the event queue, which contain information regarding the intersection of the sweep-line with objects in the plane, and upcoming events in the sweep, respectively. The event queue, E, contains scheduled future events, in this case, left and right endpoints of line segments and intersections of segments, supports the following operations: FIRST(E), INSERT(x, E), and MEMBER(x, E). By using a balanced-tree implementation, we can make these operations require at most log-time. The sweep line, S contains a description of the intersection of the sweep line with the geometric structure being swept. We need S to support the operations INSERT(x, S), DELETE(x, S), and LEFT- and RIGHT-NEIGHBOR(x, S). In this case, the main information contained in S is the vertical position of all segments that currently cross the sweep line. Figure 2.10 shows a sample set of line segments and the position of the sweep line at three of the events that occur during the sweep. The chart at the right shows the list of segments that the sweep-line encounters in top-to-bottom order for each event, as well as the event(s) that occur at that sweep position. Initially, we set the event queue to contain the x-coordinate of all segment endpoints, and set the sweep-line to be empty. At each event, we have to update both S and E. At any event, we delete that event from E and use the FIRST operator to determine the x-coordinate of the next event. At the left end of a segment ℓ, we (i) insert ℓ's y-coordinate and slope into S, and (ii) compute and insert into E the x-coordinate of the intersection(s) of ℓ with its neighbor segments in S. At the right end of ℓ, we delete ℓ from S. At the crossing of ℓ and another segment, q, we (i) delete ℓ and q from S, (ii) insert them into S again in the opposite order, and (iii) compute and insert into E the x-coordinate of any intersections between ℓ or q and their new neighbors.

The initialization cost of the algorithm is $O(n \log n)$: we must insert all of the left and right endpoints onto E. All of the event-driven updates take time $O(\log k)$ when there are k items in S or E. Since there are only n segments, the size of S will always be $O(n)$. E will likewise have $O(n)$ elements, since there are at most the n left endpoints, the n right endpoints, and a number of intersections that is no larger than the number of elements in S. Thus all events will take time $O(\log n)$. If there are c intersections overall, then the cost of running the plane sweep will be $O((n + c) \log n)$. If c is $\Omega(n^2/\log n)$, then the plane sweep will be at least as expensive as the brute-force approach, but in most cases arising in practice, the plane sweep will be less expensive.

Another major advantage of the plane sweep approach is that we can do a lot more than just find intersections, without paying an asymptotic time penalty. Figure 2.11 shows a collection of overlapping polygons with the areas of overlap shaded proportionally to the depth of coverage. We can use a plane sweep algorithm to compute all of the areas of overlap of a set of polygons, along with the depths of coverage. The output of this algorithm is known as the *arrangement* of the input

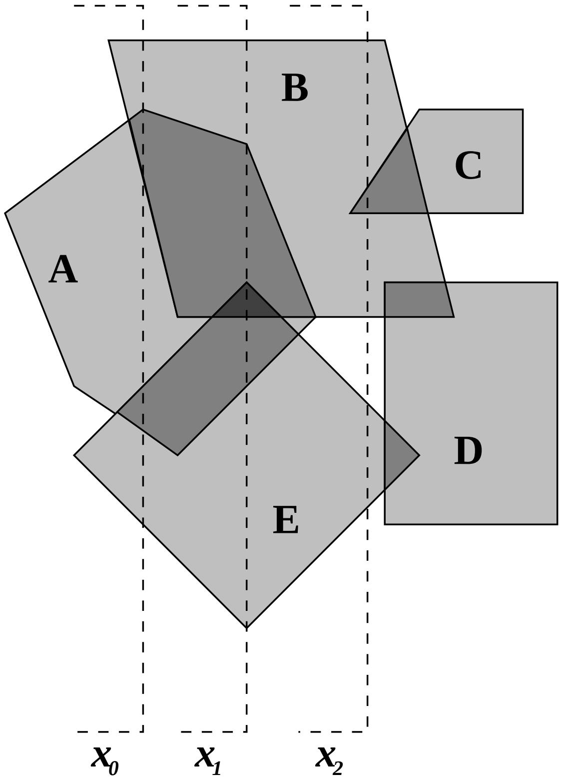


Sweep Line Position		
x_0	x_1	x_2
A	F	C
F	G/E	D
B/E	A	E
H	D	F
D	H	
Begin		
H		
End		
		E
Cross		
B,E	G,E	

Figure 2.10: An example of the segment-intersecting plane-sweep in operation

polygons. For this sweep, each entry in the sweep-line structure contains an integer that is the depth of coverage at the associated segment of the sweep-line (0 for freespace, 1 when inside one polygon, 2 when in two, and so on) and also a pointer to the list of edges that describe the left-of-sweep portion of the polygonal region that that segment of the sweep-line passes through. When an intersection occurs, if it closes off a polygonal region, we output that region's boundary and depth. This doesn't increase the cost of the algorithm, since we can amortize the cost of outputting the region's boundary down to $O(1)$ time per vertex of that boundary. The chart on the right of figure 2.11 shows the state of the sweep-line at three x-coordinates (not necessarily the x-coordinates of events).

We can compute more sophisticated coverage information using almost the same algorithm. The sweep algorithm can be used with various sorts of generalized polygons. Of particular interest to us here is the class of generalized polygons whose edges are line segments and circular arcs. The only fundamental change we must make to the algorithm is to insert any x-coordinate where a circular arc is tangent to the sweep-line; since there are only a constant number of these per arc (≤ 2), the performance analysis doesn't change.



Sweep Line Position					
x_0		x_1		x_2	
Polygon Transition/Coverage					
	0		0		0
B+		B+		B+	
	1		1		1
A+		A+		C+	
	2		2		2
B-		E+		C-	
	1		3		1
E+		B-		B-	
	2		2		0
A-		A-		E+	
	1		1		1
E-		E-		E-	
	0		0		0

Figure 2.11: Computing polygon depth coverage with a plane sweep.

We can also compute more specific arrangements. For example, in figure 2.11, suppose that polygons **A**, **C**, and **D** are colored red, while polygons **B** and **E** are colored blue. We can compute both red coverage and blue coverage at once, in the same manner as we compute overall coverage, by keeping an array of color depths in each sweep-line entry. This can still be done with the same time bounds as before, if we use some care about how we track and output coverage information. The ability to compute multicolored arrangements in this fashion is a key part of the localization algorithms we will describe next. The plane sweep algorithm is a key subpart of many computational geometric algorithms. For other instances of applications using the plane-sweep see, for example, [PS85,Don89,Lat92]).

2.2.2.2 Plane Sweep Localization Algorithms

We now explain how to construct plane sweep algorithms to compute the various types of feasible pose sets we've defined and to analyze these plane sweeps to determine their asymptotic behavior.

Theorem 2.4 *Given a set of polygons M and a range vector z, we can compute $FP(M,z)$ in time $O((n+s)\log n)$, where n is the number of edges in M and s is the number of intersections between M and $M \ominus z$.*

Proof. We can compute $D(M,z)$ and the union of $E(M,z)$ with its boundary each in time $O(n)$. The standard arrangement-computing plane-sweep algorithm runs on this collection of polygons (which has a total of $2n$ edges) in time $O((n+s)\log n)$, as shown by [NP82]. We modify the sweep-line data structure so that each segment on the sweep-line knows how many polygons of $D(M,z)$ (call these the *red* polygons) and how many polygons of $\partial(E(M,z))$ (call these the *blue* polygons) cover it. Since we have a constant number of colors to keep track of, we can make this change without altering the asymptotic cost of processing each event. As we build up chains that, upon closing, are output as regions of the arrangement, we decorate each edge with the *red* and *blue* coverage depths of the region on each side of that edge; this information gets output with the edge list for each region. Once the arrangement has been computed, we make a list of all the edges (order is unimportant). This list is $O(n+s)$ long. We go through this list and output all of the edges that have $(red, blue)$ coverage of (j,k) on one side (for $j > 0$ and $k > 0$) and $(0,0)$ on the other. These are the segments of $D(M,z)$ that are not covered by the *interior* of any polygon of $E(M,z)$, and that therefore lie in $D(M,z) - E(M,z)$. This step takes time proportional to the length of the list to perform. The overall work done therefore to compute the annotated arrangement and filter out the segments that are part of $FP(M,z)$ takes time $O((n+s)\log n)$. □

Theorem 2.5 *Given a set of polygons M, a range vector z, and an uncertainty value ϵ, we can compute $FP_\epsilon(M,z)$ in time $O((n+s)\log n)$.*

Proof. As noted previously, $FP_\epsilon(M,z)$, for $\epsilon \neq 0$ is a set of generalized polygons (line segments and circular arc segments). The introduction of circular arc segments does not change the asymptotic behavior of the plane sweep algorithm. For this problem, we again use the arrangement-generating plane-sweep with two colors, red and blue. Since the feasible poses are now, in general, regions of nonzero area, we only need to output for each region what its *red* and *blue* coverage depths are. We then filter the list of output regions, keeping all regions with $(red, blue)$ coverage of $(k,0)$, for any $k > 0$. In this case, the filtering step takes time proportional to the number of regions, which is bounded by the number of edges in the arrangement. Since both the arrangement computation and the filtering computation are no more complex than in the zero-uncertainty case, we conclude that we can compute $FP_\epsilon(M,z)$ in time $O((n+s)\log n)$. □

The analysis for multiple range probes goes similarly. The main difference in the analysis is that we must establish that we can perform all the work necessary to keep track of $2m$ colors without increasing the asymptotic cost of the algorithm. Fortunately, we don't have to perform a general $2m$-colored arrangement, Theorem 2.6 explains how lemma 2.1 allows us to extend theorems 2.4 and 2.5 efficiently to the computation of feasible poses with multiple range probes:

Theorem 2.6 *Given a set of polygons, M and a set of range probes, z, we can compute $FP(M, Z)$ in time $O((mn^2 + s)\log(mn))$, where n is the number of edges in M, m is the number of range probes, and s is the number of intersections between edges of*

$$\bigcup_{z \in Z} D(M, z) \cup E(M, \ell(0, z)).$$

Furthermore, s is $O(m^2n^2)$ in the worst case.

Proof. If we can get each $FP(M, z)$ to be nonoverlapping (no two regions intersect), then we can compute $FP(M, Z)$ by computing the intersection of the m $FP(M, z_i)$ by a straightforward arrangement algorithm: we run a plane sweep algorithm over the union of the $FP(M, z_i)$ and extract the regions with coverage depth m. Each of the $FP(M, z_i)$ can have complexity $O(n^2)$, so we would expect computing the arrangement of the m $FP(M, z_i)$ to take time $O(m^2n^4 \log(mn))$. Lemma 2.1, however, says that this arrangement will only have $O(m^2n^2)$ intersections; so its computation will take time only $O(m^2n^2 \log(mn))$ in the worst case. Therefore, we can use the (conceptually simple) method of taking the m individual arrangements and computing the intersection of those, and still meet our time bound. □

Corollary 2.7 *We can compute* $\mathrm{loc}(M, Z)$ *in time $O((mn^2 + s)\log(mn))$.*

Proof. Since we have to examine each region boundary of the arrangement in any event, we can count the number of range probes each region boundary is consistent with at no extra asymptotic cost, since addition is no more expensive than ANDing. By outputting coverage information with each edge or vertex, we increase the size of the arrangement by a factor of $O(\log m)$, but that leaves the time to output the arrangement no longer than the time to compute it. Thus, we can output the count-annotated arrangement in the same time as we can output the coverage-annotated arrangement. □

Theorem 2.8 *Given a set of polygons, M, a set of range probes, Z, and an uncertainty value ϵ, we can compute $FP_\epsilon(M, Z)$ in time $O((mn^2 + s)\log(mn))$, where n*

is the number of edges in M, m is the number of range probes, and s is the number of intersections between edges of

$$\bigcup_{z \in Z} D_\epsilon(M, z) \cup E_\epsilon(M, \ell(0, z))$$

(which is $O(m^2n^2)$ in the worst case).

Corollary 2.9 *We can compute* $\mathrm{loc}_\epsilon(M, Z)$ *in time $O((mn^2 + s)\log(mn))$.*

Theorem 2.8 follows from theorem 2.5 and lemma 2.1 in the same way that theorem 2.6 does. Corollary 2.9 follows from theorem 2.8 using the same arguments used to get from theorem 2.6 to corollary 2.7.

Finally, we wish to consider the problem of determining the minimum uncertainty, ϵ, such that $FP_\epsilon(M, Z)$ is nonempty. Megiddo, in [Meg83], provides a mechanism for transforming decision procedures into minimization procedures, for some problems. This mechanism, known as *parametric search*, applies to decision procedures that have a distinguished quantitative input, d (for example, a decision procedure $P(A, B, d)$ that determines whether $f(a, b) \leq d$, for a given function f of two arguments, A and B), such that there exists a critical value d^* for any instance of A, and B, such that for $d < d^*$, $P(A, B, d)$ is "*no*", and for $d \geq d^*$, $P(A, B, d)$ is "*yes*". The minimization problem $P_{min}(A, B)$ is to find the value of d^*. Parametric search typically takes a serial decision procedure that operates in time $O(T(n))$ and a parallel decision algorithm that operates in parallel time $O(P(n))$, and produces a minimization procedure that operates in time $O(T(n)\log^k(P(n)))$ for some positive integer k. [AST92] shows how this mechanism can be applied to finding the minimum hausdorff distance ([HK90] between two point sets. We can use a similar approach to develop an algorithm to compute the smallest uncertainty ball, ϵ_{min}, such that $FP_{\epsilon_{min}}(M, Z)$ is nonempty. [AST92] take a decision procedure with worst case time $O(m^2n^2\log(mn))$ and are able to obtain a minimization procedure that takes worst case time $O(m^2n^2\log^3(mn))$. We conjecture that we can do likewise with our problem.

2.2.3 Allowing Rotational Uncertainty

Until now, we have assumed that the robot has perfect orientation information; i. e., that it always correctly knows which direction it is facing. This may not be a valid assumption. The robot may have a compass, but that compass will have only limited resolution and accuracy; the error may not be small enough for us to ignore it. The robot may not have a compass; in this case, even if the orientational odometry is very good, there will arise occasions, such as initialization-time or after a sliding motion, when we will need to determine the robot's orientation by comparing sensory data to previously gathered information.

2.2.3.1 Extending Localization to $\mathbb{R}^2 \times \mathbf{S}^1$

Consider extending the definitions we used for translation-only localization for $\mathbb{R}^2 \times \mathrm{S}^1$. Given a map M that is a collection of planar geometric objects in $\mathbb{R}^2$, we define M_r, a map in $\mathbb{R}^2 \times \mathrm{S}^1$, so that $M_r(x, y, \theta)$ is $M(x, y)$ rotated by θ around M's origin . A planar slice of M_r at θ_0 is equivalent to the planar map in which a robot would perform localization operations most correctly, in the case where the robot has an orientation error of θ_0. The idea here is to keep the robot's coordinate system constant, and rotate the world around it. M_r is an object that occupies $\mathbb{R}^2 \times \mathrm{S}^1$ such that a slice at a given θ is equivalent to the original planar map rotated by θ. Suppose that we want to localize a robot in this new, higher dimensional object. One can think of performing the $\mathbb{R}^2$ localization procedure for any fixed θ easily enough; what we now want to do is to perform this procedure simultaneously for all θ values. Let's now consider what the geometric constructs for this three-dimensional localization are.

Because we are computing $M_r(\theta)$ by rotating M by θ around its origin, the polygons in M become solids in M_r whose edges are spirals and whose faces are helices. A vertex $v = (x, y) \equiv \langle r, \rho \rangle$ in M maps to a spiral $v(\theta) = \langle r, \rho + \theta, \theta \rangle \equiv (r\cos(\rho + \theta), r\sin(\rho + \theta), \theta)$. An edge connecting v_a and v_b, which can be described using the parametric equations $x_v(t) = x_a + (x_b - x_a)t$ and $y_v = y_a + (y_b - y_a)t$, for $0 \leq t \leq 1$ becomes a swept surface with equations: $x_v(t, \theta) = x_a(\theta) + (x_b(\theta) - x_a(\theta))t$ and $y_v(t, \theta) = y_a(\theta) + (y_b(\theta) - y_a(\theta))t$. Since we are rotating the world around the robot, instead of the robot within the world, the cross section of a range probe, $z(\theta) =< r, \phi >$ is the same over all θ: If we forget for a moment that M_r lives in $\mathbb{R}^2 \times \mathrm{S}^1$, and pretend that it lives in $\mathbb{R}^3$, we can think of z as a vertical segment of height 2π located at $(z\cos\phi, z\sin\phi, \theta)$, and $\ell(0, z)$ as a vertical rectangle of height 2π, width $\|z\|$, and orientation ϕ with respect to the x axis. We can now define D as before and also E, except that now these sets live in $\mathbb{R}^2 \times \mathrm{S}^1$ instead of $\mathbb{R}^2$; i. e., D is the convolution of a vertical line (actually a circle) with a set of extruded helical objects, and E is the convolution of a vertical rectangle with those same objects. If we allow uncertainty, D becomes the convolution of a vertical cylinder (actually a torus) with the extruded helical objects, while E has the same geometric structure as before, but with a narrower vertical rectangle. If the point (x, y, θ) is in

$FP(M, z)$, then the point (x, y) rotated around the origin by $-\theta$ is a feasible pose, if the robot has an orientation error of θ.

We can use a space-sweep algorithm (the 3D analog to a plane-sweep algorithm) to compute these feasible pose sets. While we have neither detailed analyses nor complexity bounds for the sweep algorithm needed for this particular problem, we believe the cost to be high (on the order of $O((m + n)^6 \log^k(mn))$ for some k), as the complexity of the problem seems closely related to that of finding the minimum Hausdorff distance between two sets of points, segments, and polygons in the plane under Euclidean motion (translation and rotation). The current best algorithm for this problem (proposed by [CGH$^+$93]) has an upper time bound of $O(m^3 n^3 \log^2(mn))$, for two sets P with complexity m and Q with complexity n. This bound appears to be close to optimal (within $O(\log^k(mn))$). [Ruc93]

2.2.3.2 Theta-Slice Approximating $\mathbb{R}^2 \times \mathbf{S}^1$ Localization

Perhaps a more practical method of handling rotational uncertainty in the localization process is to approximate the procedure just outlined by performing translation-only localization at a set of discrete rotations. We can compute M_{k_r} for $k = 0, 1, \ldots, p - 1$ in $O(np)$ time. If we can perform the translation-only localization at each $k\Delta\theta$ in time $T(n, m)$ for an environment M of size n and m range probes Z, then we can determine which of the $k\Delta\theta$ angle offsets is(are) most compatible with M and Z in time $O(pT(n, m))$.

2.2.3.3 Separating Orientation Localization from Translation Localization

We may not want to extend the localization algorithm to handle rotational uncertainty at all. Under some robot and environment models, it makes more sense to perform rotational localization separately from translational localization. Among the models to which this applies are (i) robots equipped with bounded-error compasses or gyroscopes, and (ii) robots operating in environments such as uncluttered office environments where a large majority of the vertical surfaces in the environment have orientation along a set of cardinal axes. In cases like this it may make more sense to use some sort of sensory alignment procedure to reduce orientation uncertainty; using such a procedure in coordination with a translation-only localization algorithm may be more efficient than using an $\mathbb{R}^2 \times \mathrm{S}^1$ localization algorithm. In chapter 3 we will discuss some techniques we have used for reducing rotational uncertainty.

2.3 Using Rasterization for Practical Localization Algorithms

For performance reasons, we use rasterized algorithms when we implement algorithms involving geometry on the robot. A *rasterized algorithm* employs the following technique: given a geometric description (for example, lists of edges or vertices), imbed that description in an array by digitizing the primitives (in essence, "plotting" geometric primitives from the description on an initially blank bitmap). Given an algorithm that runs on geometric descriptions, convert (by hand) the algorithm to perform on a discretized grid. We believe that algorithms operating on discretized arrays are useful, in that they are often easier to implement than the original combinatorial algorithms, and they often run faster. See [Lat92], [BL91], [LP83], [LRDG90], [HK93], [CK92] for examples of how rasterization can be applied to robotics algorithms. In this section, we define rasterized algorithms, and define some basic building blocks from which rasterized algorithms can be built. We then present rasterized versions of the translation-only ($\mathbb{R}^2$) localization algorithms presented in section 2.2.2.

Another reason for preferring rasterized algorithms for on-robot implementations lies in the type of data that is actually available to the robot. Many results in theoretical robotics are built on the data type *set of polygonal obstacles* (polyhedral obstacles, in three dimensions). It is impossible, however, to make a map that is a good polygonal/polyhedral representation of an unstructured environment. Part of this is due to limited sensor accuracy, but there are more fundamental problems: For one thing, the world is not made up of polyhedral objects. Objects in the world, such as piles of cables or hockey bags, can often only be represented approximately using polygonal/polyhedral shapes. The cost of representing such an object can be high, as it may require a large number of segments or faces. In addition, obtaining that representation will likely require arbitrary application of line- and curve-fitting procedures. Given a specification of required accuracy for our maps, we believe it is simpler to select a minimum feature size (based on that specification), and to make our maps using that feature size as their resolution. Thus, we feel rasterized maps are the most straightforward way to represent the real-space world our robot encounters. We want, therefore, to use rasterized computational geometry algorithms for manipulating our maps, since using standard computational geometry algorithms would require frequent conversion between rasterized and polygonal representations, leading to loss of accuracy.

2.3.1 Introduction to Rasterized Computational Geometry

Rasterized algorithms are not new. Many existing rasterized algorithms were originally developed by researchers in computer vision and computer graphics. Computer vision researchers often use rasterized algorithms, largely because computer vision algorithms operate, at least at low levels, upon images obtained from video cameras or scanners, which invariably *provide* rectangular arrays of intensity and/or color values. Computer graphics researchers design algorithms that create images for display upon video screens, which almost always *expect* rectangular arrays of intensity and/or color values. A video image has limited resolution, causing features below a certain scale to disappear and the locations of larger features to be known only approximately (usually to the nearest pixel). Thus, it makes sense to design computer vision and computer graphics algorithms that will operate well on rectangular arrays of limited resolution data. For example, there is often little advantage to a hidden-surface removal algorithm employing exact (rational) arithmetic and numbers for precision, if the output is to be displayed on a screen with 1024 pixel by 768 pixel resolution.

2.3.2 Some Building Blocks for Rasterized Algorithms

The fundamental data type of rasterized algorithms is the *grid*. A grid is divided up into *cells*, each of which can take on an integer value. Most operations fall into two categories: grid-level and cell-level. Basic grid-level operations include cell-wise arithmetic and logic (for example, given two equal-sized grids, A and B, $(A+B)[i,j] = A[i,j] + B[i,j]$) and transformation operations (e.g. x-y-translation ($A[i,j] = B[i+p, k+q]$) and reflection ($A[i,j] = B[p-i, q-j]$)). Cell-level operations typically assign a value to each cell as a function of its neighbor cells. More complex functions are built up from sequential applications of these basic operations.

2.3.2.1 Discrete Convolution

The Minkowski sum operation, defined in section 2.2.1 as

$$P \oplus Q = \{p + q \mid p \in P, q \in Q\}$$

can be applied directly in the rasterized domain. In the continuous domain, if P and Q are finite sets of discrete points, we can compute $P \oplus Q$ by a simple iteration: for each $p \in P$, iterate over all $q \in Q$, outputting $p + q$. In the rasterized domain, there are no continuous point sets: curves and areas in the continuous domain are

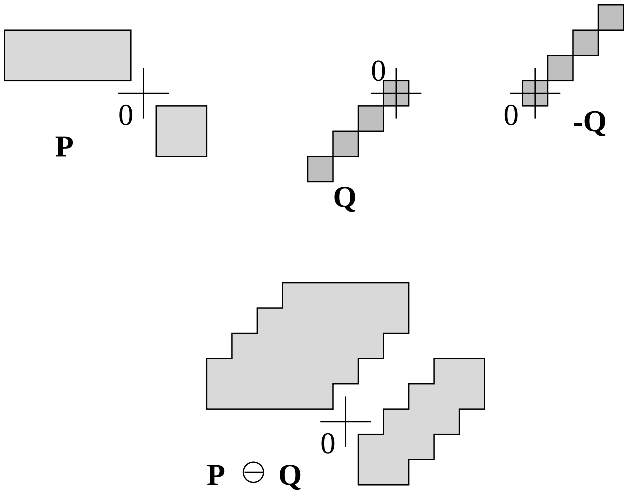


Figure 2.12: Discretized Convolution: $P \ominus Q$

transformed into contiguous sets of pixels. Since all finite sets of discrete points have finite cardinality, we can compute $P \oplus Q$ for any pair of finite rasterized sets, P and Q. The operation denoted $\oplus$, applied to a rasterized domain, is exactly the operation that is known in digital signal processing [OW83,OS89], image processing, and computer vision circles [MH79,Mar82] as convolution. In digital filter design, for example, a common application of convolution is the Finite Impulse Response filter, whose output is computed by convolving a linear or rectangular region with a sampled input. For example, many of the "smoothing" filters and edge-detectors in common use today rely on this operation. Since it is a very common, convolution is an operation that that many processors (particularly those designed specifically for digital signal processing) are optimized to perform. Since specialized signal/image processing hardware is capable of performing this operation very quickly, discrete convolution is a very useful operation for rasterized-algorithms. Figure 2.12 shows a sample rasterized convolution.

2.3.2.2 Floodfill/Brushfire Algorithms

Breadth First Search[AHU83] is an often-used order for visiting nodes in a graph, in which we visit/examine/compute upon the nodes adjacent to the start node first, then those that are at distance two from the start, then those at distance three, and so on until all nodes have been visited. In the context of rasterized algorithms, this order is known as a "floodfill" or "brushfire" algorithm. In its simplest form, the procedure to floodfill an array can be described as (i) number all source cells 0 and place them on the floodfill queue. (ii) while the queue is nonempty, dequeue

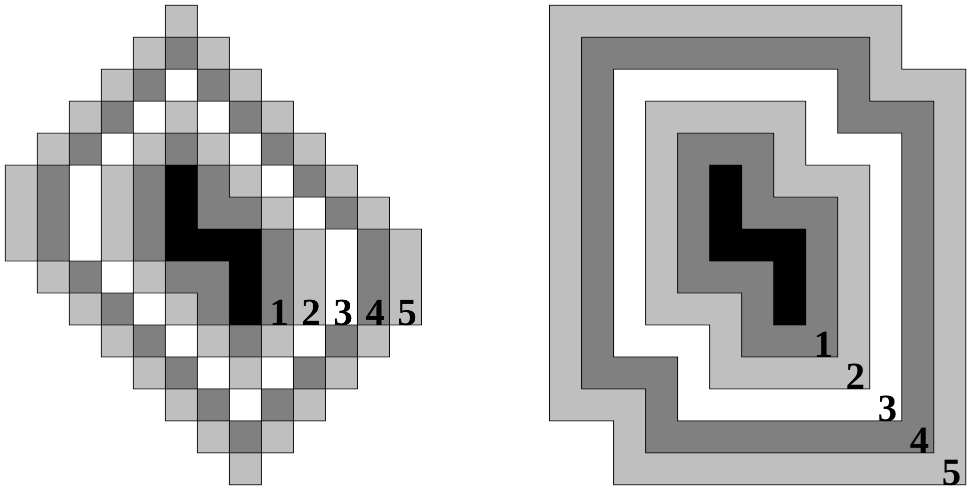


Figure 2.13: A pixelized shape shown with flooded L_1 and L_∞ contours.

a cell c; if c bears the number i, number all of its unnumbered neighbor cells $i+1$ and enqueue them. This algorithm floods from a set of source cells in contours that are determined by the definition of *neighbor*. If we define neighbors to be those cells that share a side (resp. side or corner) with the current cell, then we flood in diamond (square) shapes so that the number of a given cell is its distance to the nearest source cell under an L_1 (L_∞) distance metric. Figure 2.13 shows the shape of these contours for a sample pixelized shape.

We can define *obstacle cells* in the array as those that cannot be flooded. If we do this, then the contours define the distance to the nearest source cell via a path that does not cross any obstacle cell. In fact, [LRDG90] makes use of this property to design a very fast robot path planner that computes the shortest path from anywhere in the robot's configuration space to a specified goal. Once the countours are in place, the path from a given start point to that goal can be determined in real time, provided it exists; if there is no path to the goal, this can be determined instantaneously. We use a variant of this planner on Lily and will discuss it further in chapter 3.

Sometimes we want to be able to flood from a set of source points only out to a specified distance. For example, consider the C-space obstacles generated by a path-planner for a circular robot. These C-space obstacles are the real obstacles grown out by the diameter of the robot in all directions. Thus, if we are interested in the grown-obstacle-map of a diamond- or square-shaped (i. e., L_1 or L_∞ circular) robot with radius r, we can easily compute its C-space by the following modified floodfill algorithm: (i) number all obstacle cells 0 and place them on the floodfill queue. (ii) while the queue is nonempty, dequeue a cell; if its number is less than r, number all of its unnumbered 4-connected neighbor cells (resp. 8-connected neighbor cells) one higher and enqueue them. This algorithm floods from a set of source cells in contours that are circular under an L_1 (resp. L_∞) distance metric. It only floods out to a specific distance, however, so that the set of all numbered cells after the algorithm is run is the same set of cells as obtained by taking the Minkowski sum of the obstacle map and an L_1 (L_∞) disk of the radius of the robot. Since we never enqueue a cell more than once, the cost is only proportional to the area of the map with the flood-fill case, as opposed to the map-area times disk-area cost of the Minkowski sum. Unfortunately, it is not possible to flood in this same fashion using an L_2 disk. It is, however, possible to approximate a small L_2 disk (one with radius less than 20 cells) with an octagonal region (an "$L_{octagon}$" disk) whose boundary is never more than one cell-size away from the boundary of an L_2 disk of the same radius. This "$L_{octagon}$ disk" is obtained by alternating in a pre-specified pattern between 4-neighbor and 8-neighbor growth. Figure 2.14 shows the distance contours associated with $L_{octagon}$ growth, along with the L_2 circles they approximate. Which set of neighbors we use for a particular cell depends solely on the potential value given to that cell. Given that we ordinarily use a map whose cell-size is within an order of magnitude of the robot size, this method of determining the grown-obstacle-space is adequate, and in practice much faster than taking the Minkowski sum.

It is also useful to be able to "flood" in a specific direction. A sample application here is computing the grown-obstacle-space of a rod-shaped robot at various orientations. As before, we could perform this operation by taking the Minkowski sum of the original grid contents and the pixelized vector representing the direction and distance of growth. A faster way to do this is to extract from the pixelized vector V_r a list of "neighbor numbers", which map from the pixels of V_r to the integer range $[0-7]$. $neighbor[i]$ represents the direction from the i^{th} pixel of V_r to the $i+1^{st}$ (its neighbor going away from the origin): the pixel to the right has neighbor number 0; the pixel above and to the right has neighbor number 1; the rest of the neighbors are numbered in counterclockwise order from there. Now, when we do the floodfill, if we take a cell numbered i off of the queue, we only number and enqueue the one neighbor cell that is indicated by $neighbor[i]$. This way, we flood from source cells only in the direction of the pixelized vector. In fact, making this approach work properly is slightly more complicated: Since the grid really only recognizes eight distinct directions at a local level, it is possible for the vector that should stem from a source pixel to be partially or wholly omitted, in the case where the surface tangent of an obstacle at that source pixel is close to the direction of

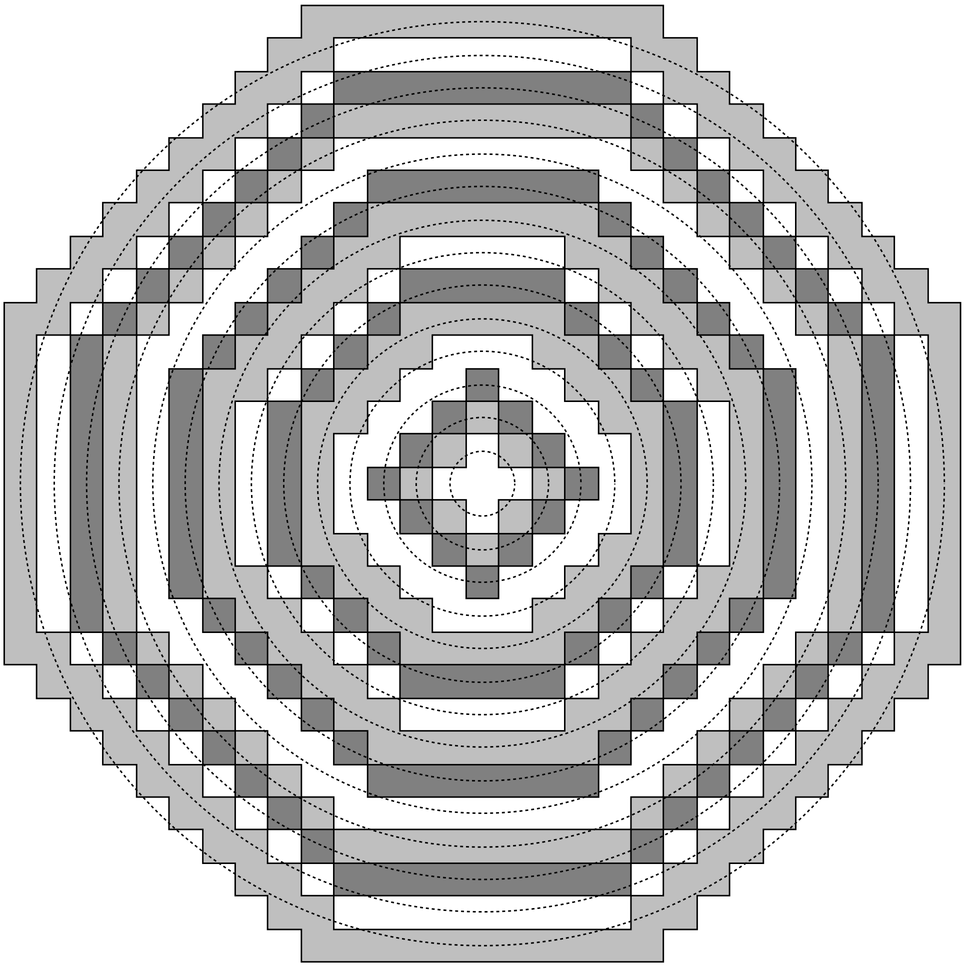


Figure 2.14: $L_{octagon}$ contours flooded out from a point source. L_2 circles are shown for reference.

---

the range vector. This happens when the 8-directional pixelized vector points locally into the obstacle, even though the original vector points out of the obstacle. Figure 2.15a shows an example of this problem. The vector shown is sourced from the starred cell; the first three cells of the pixelized vector lie within the obstacle, even though the original vector points out of the obstacle when sourced from the obstacle's upper boundary. We address this difficulty using a lookahead scheme that causes the flood to work correctly without affecting the algorithm's asymptotic performance. In figure 2.15b, we've added the pixelized vector sourced at the cell marked '+'. Hashed cells lie on the pixelized vectors sourced at either '*' or '+', while cross-hashed cells are those free-space cells that lie on both pixelized vectors. The cells that are forward-hashed, but not back-hashed are those that would not get numbered if the '*' vector were eliminated but the '+' vector remained. To ensure

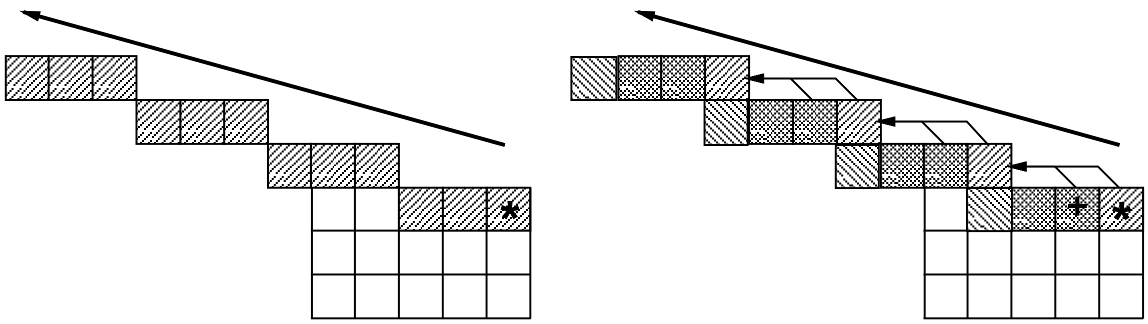


Figure 2.15: (a). A pixelized vector may overlap a pixelized object, even though the original (unpixelized) vector points out of the object. (b). To avoid this and some other difficulties, we use a lookahead scheme based on the way the pixelized vector overlaps a translated copy of itself.

---

that these cells get numbered properly, at each step the floodfill procedure looks ahead to two cells: the cell indicated by the neighbor number of the current cell and the cell that is in the position of the nearest higher-numbered cell that *wouldn't* get marked if the neighbor cell already bore a lower number. Which cell that is for a given potential can be precomputed once for each cell on the pixelized vector at the beginning of the fill, by examining how the pixelized vector overlaps a translated copy of itself (translated by one pixel in the dominant direction). Essentially, we create a picture like the one in figure 2.15b, for the particular vector being used, and look at the overlap.

The directional floodfill algorithm has an additional small performance advantage over omnidirectional floodfill procedures: Since we know in advance the direction in which growth will be taking place, it is not in fact necessary to use a queue-based implementation to perform directional flooding. It is sufficient to make a single ordered sweep across the grid, with the direction of the sweep determined by the orientation of the growth vector. In an omnidirectional fill, we must be certain that all cells at a distance less than k from the source cells have been processed before beginning to process the cells at distance k; otherwise, some cells may get a larger than necessary potential number (in fact, we can prevent this, but only by increasing the cost of the procedure due to the necessity of processing some cells multiple times). In the directional case, we only need to ensure that the lower-numbered cells that contribute to the numbering of a cell at distance k be numbered. For this reason, we can sweep the grid in a direction that will cause cells on a straight line from a given source cell to be processed in order of increasing distance. This does not affect the asymptotic behavior of the flooding procedure, but may allow significant reduction in the constants pertaining to the performance of a particular implementation of the procedure.

2.3.3 Rasterized Localization in $\mathbb{R}^2$

2.3.3.1 Localization Using Discrete Convolution

A simple implementation of the algorithm to compute the feasible pose set for a given rasterized range vector, map, and uncertainty follows directly from the definition of $FP_\epsilon(M, z)$ in equation (2.4): suppose we have an inverted range vector z, uncertainty ϵ, and map M (figure 2.16). Create a set Z_r that is the set of integer gridpoints that intersect the vector z at a distance greater than ϵ from the head of z. Create also a set Z_ϵ that is the set of integer gridpoints with a Euclidean distance of ϵ of the head of z. Regard the map M conceptually as a set of points, but maintain it as a two-dimensional array. Compute D (which can be regarded as the set of all cells that are within ϵ of a cell that is itself the vector sum of z with some obstacle cell) by convolving M with Z_ϵ. Remember that convolution of a bitmap with a point set is equivalent to offsetting the bitmap by the coordinates of each member of the point set, and computing the union of each of those offset bitmaps by ORing them all together. Next, we compute an intermediate set E (the set of all cells that are within $r - \epsilon$ of an obstacle cell in the direction of z) by convolving M with Z_r. Now, FP is just the set difference $D - E$

The complexity analysis is straightforward. Given a map with size $n \times n$, a vector that discretizes into r cells, and an uncertainty ball of radius ϵ, then the time required can be expressed $O(n^2(r+\epsilon^2))$. This is the number of cells in the map times the number of times that an offset-and-copy operation has to be performed on the map. With multiple range probes, the cost is linear in the number of such probes: we can intersect the m feasible pose sets $FP(M, z_i)$ in time $O(n^2 m)$, which does not increase the asymptotic cost beyond the $O(n^2(r + \epsilon^2)m)$ total cost for computing the m individual $FP(M, z_i)$. Determining the coverage of each cell in M can be done in the exact same time complexity by counting the number of $FP(M, z_i)$ that contain that cell instead of ANDing the cells together. In the discretized case, we can determine the optimal ϵ for a given coverage at a cost increase of $O(\log n)$ by a simple binary search tactic: since we can only select integer values for ϵ, we need not resort to parametric search. This gives us an overall time complexity for determining the smallest nonempty $FP(M, Z)$ of $O(n^2 m \log n(r + \epsilon^2))$.

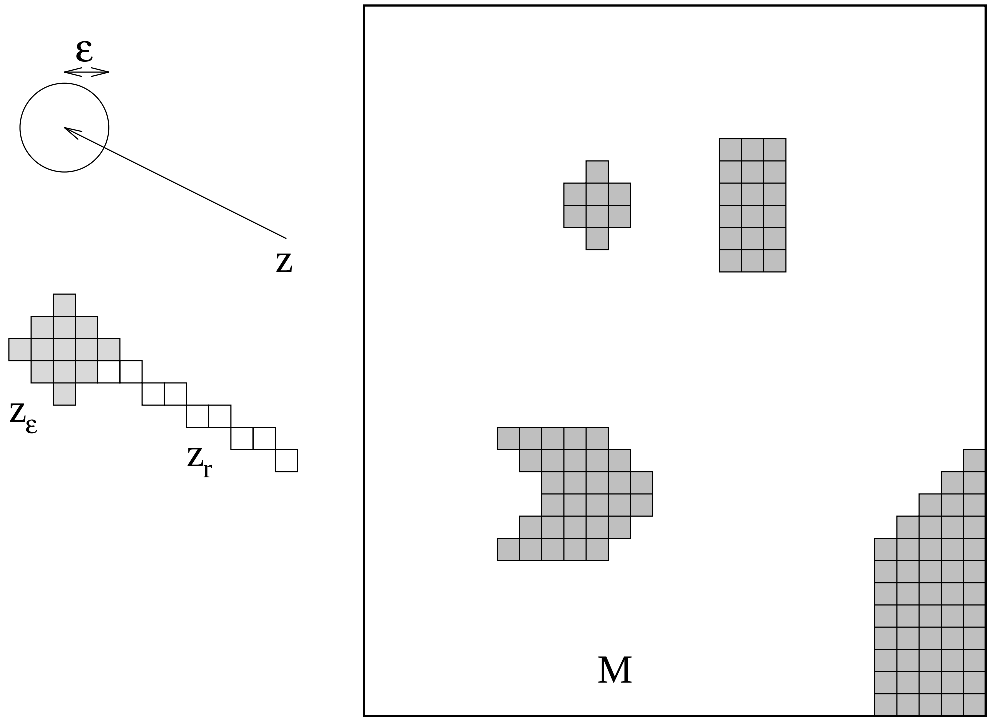


Figure 2.16: M flooded in the direction of z by distance $||z|| - \epsilon$

2.3.3.2 Speeding up the Rasterized Localization Algorithm

The algorithm of section 2.3.3.1 does redundant work. For a single range probe z, each cell in the map M is copied and operated upon $O(r + \epsilon^2)$ times, for r the length of the range probe and ϵ the radius of the uncertainty ball. This is true even of those cells that are inside an obstacle, or out in the middle of freespace. We can reduce the cost of computing an individual $FP_\epsilon(M, z)$ to $O(n^2)$, independent of r and ϵ. This means we can compute the smallest nonempty $FP_\epsilon(M, Z)$ in time $O(n^2 m \log n)$.

We can reduce the cost of computing $FP_\epsilon(M, Z)$ to $O(mn^2 r)$ by approximating the error ball so that we can use a flood-fill-based method to compute D. We do this using much the same method as used to compute the C-space obstacles: We translate the map by $-z$, then grow it by an "$L_{octagon}$" disk of radius ϵ, using a limited range flood-fill, as described in 2.3.2.2. This is the approximate D set, which is almost the same as the D computed by the more expensive convolution method. It is, in fact, identical for $\epsilon \leq 4$ cells. For ϵ between 5 and about 20, the boundary of the $L_{octagon}$ disk of radius ϵ lies within one cell-size of the L_2 disk of the same radius. Figure 2.17 shows a sample rasterized M along with the D set. Translated

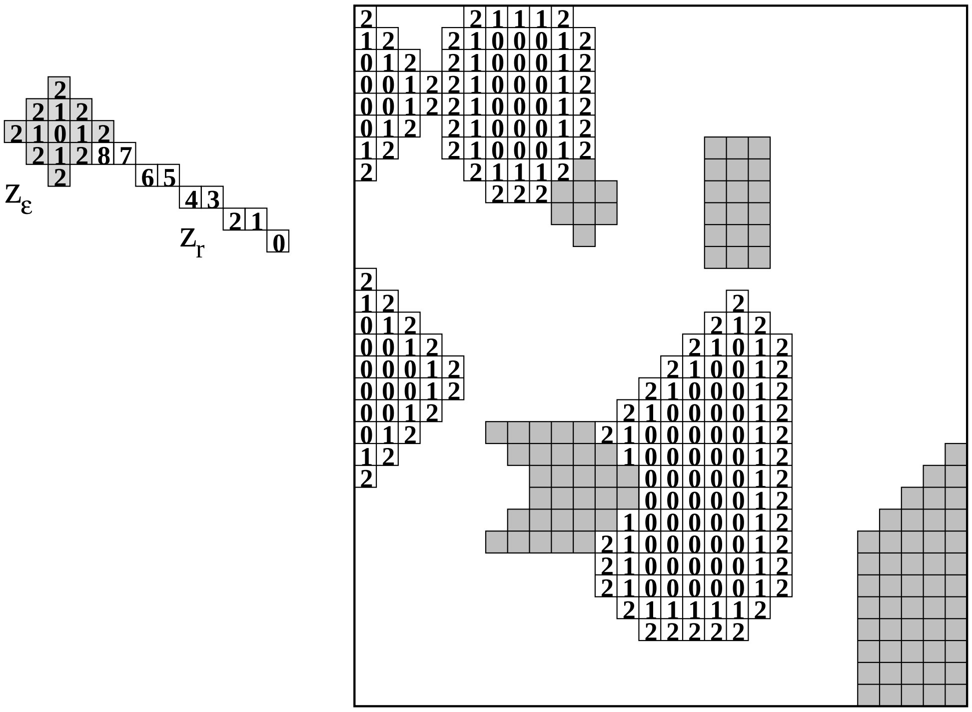


Figure 2.17: $M \ominus z$ flooded by an ϵ of 2, shown with the original M

M pixels are numbered 0; pixels at distances 1 and 2 away from 0-numbered pixels are numbered 1 and 2.

We further reduce the cost to $O(mn^2)$ by using the directional floodfill of section 2.3.2.2 to compute E. For a range vector at angle θ of length r (resp. $r - \epsilon$, if the uncertainty value is nonzero), we do this in the following way: we start with the obstacle pixels in place and numbered zero, and perform a directional floodfill out to distance r (resp. $r - \epsilon$). Finally, we compute $FP_\epsilon(M, z)$ by merging the two arrays by the following algorithm: (i), set all numbered cells in $D(M, z)$ to a distinguished negative value k_D. (ii), If a cell in $E(M, z)$ has a numbered, we copy that number into the corresponding cell in $D(M, z)$. All cells that still bear the value k_D are feasible pose cells for map M and range probe z. Figure 2.18 shows a directional flood from M in the direction of $-z$. Figure 2.19 shows $FP_\epsilon(M, z)$: numbered cells are in $FP_\epsilon(M, z)$. The white regions are cells covered by D_ϵ, and gray regions are the original M.

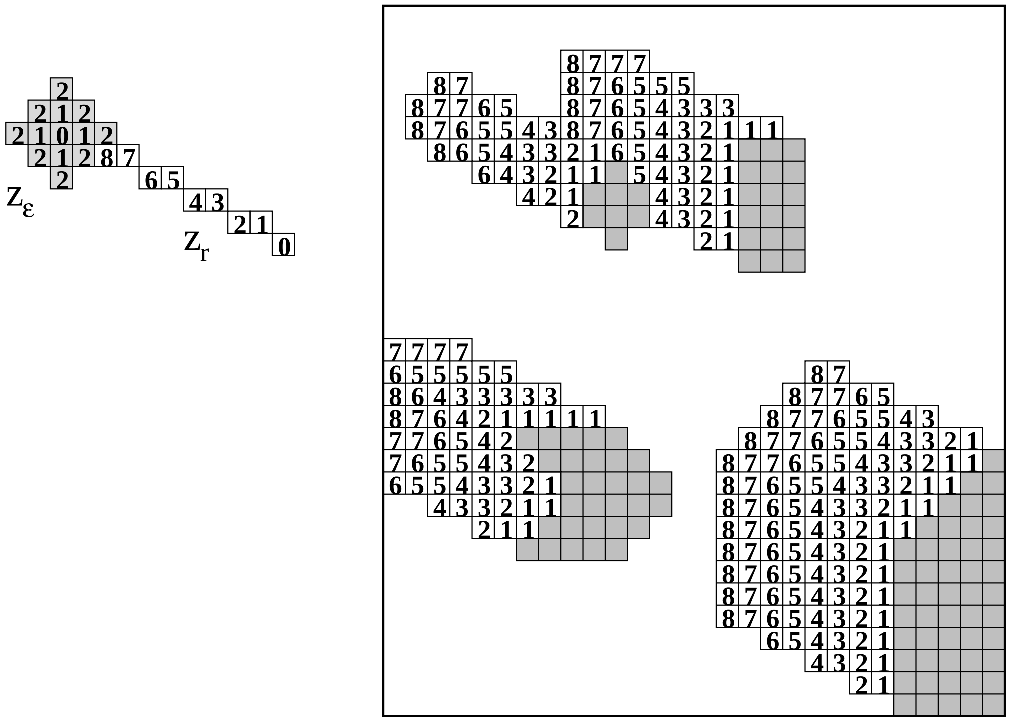


Figure 2.18: M flooded in the direction of z by distance $||z|| - \epsilon$

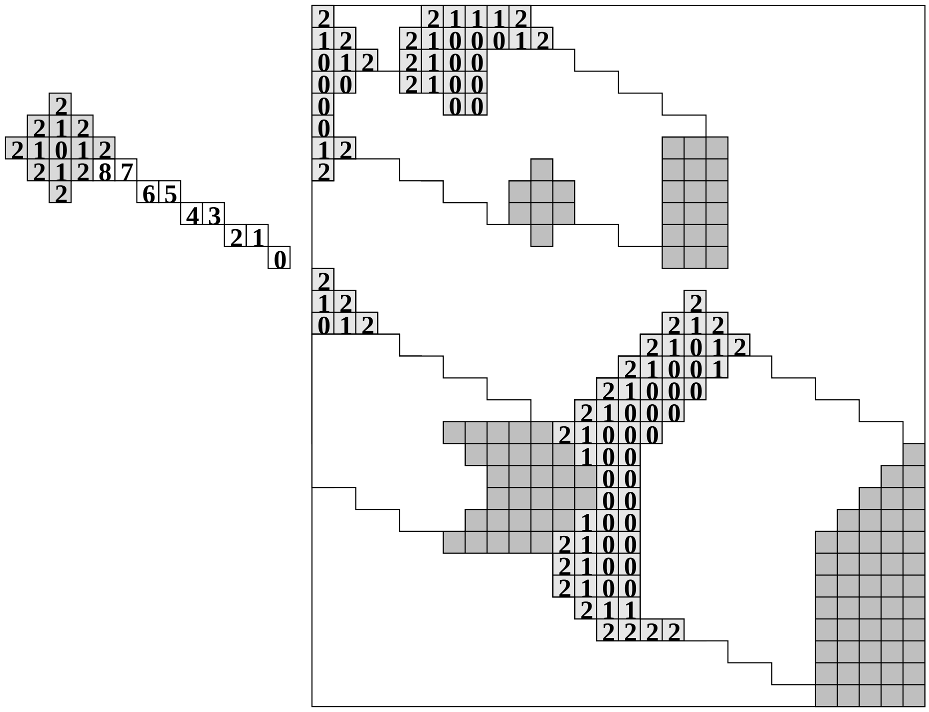


Figure 2.19: $FP_\epsilon(M, z)$: the shaded and numbered cells are those from D_ϵ that do not lie under E_ϵ. The original M is included for reference.

2.4 Summary

In this chapter, we defined a model of mobile robot localization that builds upon the concept of *feasible poses*, which are poses for a mobile robot that are consistent with available range and map data. We model a point-and-shoot rangefinder as a sensor that returns the distance to the nearest object in the direction of its sensitivity, toleranced by an error bound. A feasible pose relative to a map and a range vector is essentially a pose from which a rangefinder could return the given range vector, in a world characterized by the map. We provided formal definitions for feasible poses relative to single range probes and relative to sets of range probes, both with and without uncertainty in the range data. We gave exact computational-geometric algorithms to compute feasible poses. Finally, we defined feasible poses in a rasterized framework, and provided algorithms to compute feasible poses using *rasterized computational geometry*. One of our robots, LILY uses a laser rangefinder and these rasterized algorithms to perform self localization by computation of feasible poses. In chapter 6, we show examples of our localization system in operation.

Chapter 3

Implementing Lily's Mapmaking and Localization System

3.1 Introduction

To test our ideas about mobile robot localization (and the appropriateness of our robots and sensors to navigation using our model of localization), we implemented a navigation system supporting several navigation functions, including mapmaking, localization, and path planning and execution. We wanted to answer the question, "could we implement a system around the localization algorithms described in chapter 2 that would be capable of navigating around office environments in a robust fashion?" We have implemented a navigation system on our mobile robot LILY. This chapter describes several aspects of that system. The remainder of this section provides a rough description of the various components of the system (except for the localization component, which has already been discussed at length), and a description of the environment in which it was implemented. Subsequent sections of this chapter provide more detailed descriptions of the system components. All of the program code for procedures described in this chapter can be found in Appendix C. It is important to note that many of the algorithms and techniques described in this chapter are based on and very similar to existing work; the main purpose of this chapter is to describe support functions added to LILY's software system to allow us to test the localization algorithms described in chapter 2 and the cooperative large-scale manipulation protocols described in chapter 5. We do describe in this chapter, however, some minor enhancements that we have added to existing algorithms. For example, section 3.2.2 describes one of our techniques for enabling the robot to cope with changes in its environment.

The mapmaking technique we chose for this system is a variant of the *statistical occupancy grid*. Moravec and Elfes introduced this method of mapmaking in [Elf87, Mor89,ME85] as a way of making detailed geometric maps using noisy sensors (sonar rangefinders, in their case). A simple occupancy grid is a bitmap, where ones represent occupied cells and zeroes vacant cells. The main idea behind the statistical occupancy grid is that, since range data is inexact, the contents of each cell should be a probability: a cell containing a high probability is likely to be occupied, while a cell with low probability is likely empty. Moravec and Elfes use an update rule based on Bayesian probability to maintain the contents of each cell based on range readings that impinge upon that cell. We chose to adapt their work for a number of reasons, but primarily because a grid-based mapmaking technique fit best with our ideas about rasterized computational geometry for robotics (as discussed in chapter 2). Our method differs somewhat from theirs: we use a different structure for maintaining occupancy data, and we use an update rule that, while keeping track of the probability of occupancy for each cell, integrates absolute change in probability. This way, frequently changing cells can be distinguished from those whose values seldom change. These ideas will be discussed further in section 3.2.

The path-planning module in this system is also a grid-based/rasterized technique. We've chosen to implement a simplified version of the C-space potential field path planner of [LRDG90]. They implemented a fast $\mathbb{R}^2 \times \mathrm{S}^1$ planner using specialized graphics hardware to compute the configuration space of a planar obstacle map and robot and to grow a potential field from the specified goal, such that from any start position, the robot need only follow the descending gradient of the potential field to reach the global minimum in the potential field, which occurs at the goal. We instead implement an $\mathbb{R}^2$ planner by computing a planar C-space and growing the potential field in that space. This is sufficient for our purposes, because our robots are rotationally symmetric. More details on this part of the system are found in section 3.3 of this chapter, which draws on techniques described in section 2.3.2.2.

Finally, we discuss a number of minor functions that make use of the laser rangefinder to perform various navigation-related tasks. We have implemented several primitive-recognition routines: they identify features such as convex and concave corners and long, straight edges in specified portions of the environment. These functions can be used as subroutines to the localization procedure: detecting nearby corners is useful for determining which range probes are most useful for the localization procedures outlined in chapter 2. In addition, the ability to detect long edges in the environment is useful for rotational localization. Several of these minor parts of the system, along with their applications, are discussed in section 3.4.

3.1.1 Programming the Robot

The high-level environment we use for programming our robots is a SCHEME byte-code interpreter running on the robot, that can be tied to a byte-code compiler and symbolic debugger running on a SPARCstation. This system, implemented by Jonathan Rees on top of a SCHEME-48 virtual machine (and described in detail in [RD92]), allows the user to establish a remote read-eval-print loop on the robot (across a detachable RS-232 serial line). This enables the user to define functions, download programs, and evaluate forms on the robot, with the output displayed on the SPARCstation. It allows the user to execute programs on the robot *without* the SPARCstation, simply by removing the serial line. The built-in library to Mobot SCHEME allows the user to access all of the sensors and actuators on the robot through SCHEME functions, and to define arbitrary actions to be executed upon certain events; for example, each robot has several push-buttons that can be bound to execute an arbitrary zero-argument lambda expression. Mobot SCHEME is multi-threaded, allowing the user to define light-weight threads that execute asynchronously.

In programming our robots for mapmaking and localization tasks, we have found Mobot SCHEME to be a wonderfully flexible system that is very easy to program and to use. We encountered some performance shortfalls, however, when we tried to run the rasterized localization algorithms described in chapter 2, as well as the mapmaking system that is detailed later in this chapter. The main difficulties are the following:

- Mobot SCHEME's thread package relies on a timer-based scheduler: Every time there is a timer interrupt, SCHEME code is executed that determines when it's time to change threads. This scheduler has a relatively high overhead cost, including enough consing to guarantee several garbage collections per minute.

- Amplifying the cost of these garbage collections is the fact that almost all of our limited memory is taken up by large two-dimensional arrays: most available memory (the number of available storage cells in one of SCHEME's two semispaces[1]) is occupied by large, permanent objects. The net effect is that, every few seconds, large blocks of largely unchanged data get copied from one semispace to another; this happens frequently because there is little

[1] SCHEME-48 divides its memory up into two evenly sized semispaces. At a given time, only one semispace is actively used. When that semispace fills up, a garbage collection occurs that copies all active storage elements (those holding reachable information) into the other semispace. The active and inactive semispaces then trade roles, and execution continues.

"volatile" (unallocated) memory available. The version of SCHEME-48 we use does not provide for noncopying memory, so there is no way to avoid this behavior in this version (but a version of SCHEME-48 has recently become available that will remedy this problem when a new Mobot SCHEME is built).

- Since the user's SCHEME code is byte-code interpreted, there is a large cost for byte-wise operations on large arrays. Since we don't have the ability to compile SCHEME code directly into optimized machine code, this is an unavoidable penalty.

One solution to these difficulties is to use C instead of SCHEME. Instead, and rather than throwing away the flexibility that Mobot SCHEME gives us, we had Jonathan Rees add a foreign function interface to the Mobot SCHEME implementation. This provides us the ability to add primitives to the language, where the new primitives are C functions provided by the user. Figure 3.1 shows both ends of this interface. The upper part of the figure is SCHEME code that defines the SCHEME call to a C subroutine. The lower part is C code that extracts C-style arguments from the SCHEME argument list, calls a C function, and returns a value that SCHEME can recognize. Also provided by the interface are the ability to run C code before invoking the SCHEME virtual machine and the ability to define "SCHEME memory" and "C memory". C memory is reserved space that SCHEME cannot access, and that is not garbage collected. By storing, for example, our large arrays in this space, we avoid frequent copying.

3.2 Mapmaking

3.2.1 Occupancy Grids

The statistical occupancy grid, as introduced in [Mor89,Elf87,ME85], is one way of fusing the information returned by many range probes into one coherent picture of the world. The main concept behind the statistical occupancy grid is that a range reading should be regarded as statistical evidence about the state of the world: a range probe of one meter in a given direction says that there *probably* is nothing *closer* than one meter to the ranger, and that there *probably* is something *at* one meter from the ranger. This means that a particular range probe should be fused into the overall map according to the following guidelines: all cells through which the range probe passes should have their probability of occupancy adjusted downward, while the cells in which the range probe terminates should have their probability of occupancy adjusted upward. Moravec and Elfes maintain separate "occupied" and

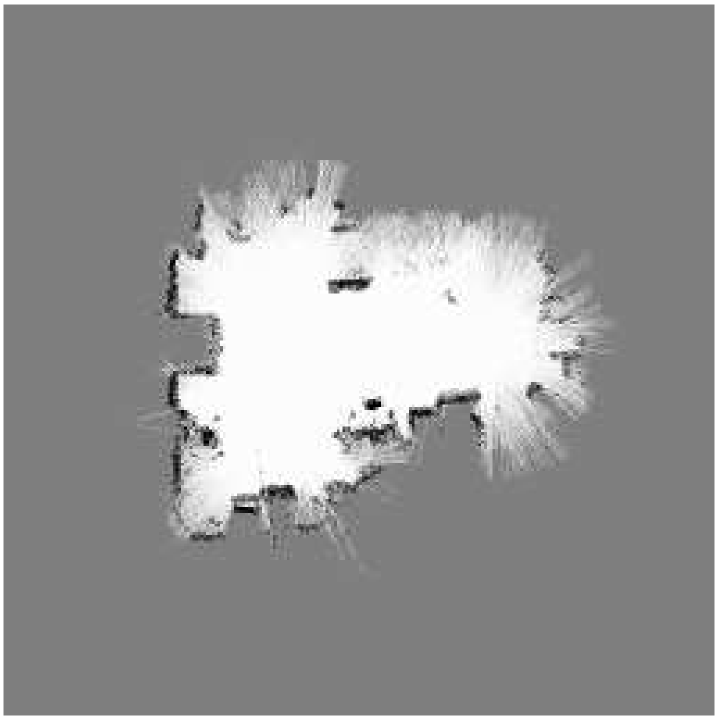
```
; SCHEME code defining the interface call from SCHEME to a C function
(define *add-vector-localization-addr* #x00801cb6)

(define sch-add-vector-localization
                  (address->external *add-vector-localization-addr*))

(define (add-vector-localization side range theta epsilon)
    (external-call sch-add-vector-localization
                   side range theta epsilon))
```

```
/* a C function that can be called from SCHEME, provided that SCHEME
   has the necessary definitions made:  A SCHEME-callable C function
   has return type ``scheme_value'', and a calling convention that
   allows for a variable number of arguments:  The first argument is
   the number of SCHEME arguments passed in, while the second is a
   list of scheme_value-typed objects that are the SCHEME arguments
   in right-to-left order. */

scheme_value sch_add_vector_localization(nargs, args)

long nargs;
scheme_value *args;
{
  int epsilon, range, theta, side;
  point range_vector;

  side = EXTRACT_FIXNUM(args[3]);    /* arguments are presented */
  range = EXTRACT_FIXNUM(args[2]);   /* in right-to-left order  */
  theta = EXTRACT_FIXNUM(args[1]);
  epsilon = EXTRACT_FIXNUM(args[0]);
  if (side == RIGHT)
    theta -= beam_angle_offset(range);
  else
    theta += beam_angle_offset(range);
  range_vector.x = r_cos_theta(range,theta);
  range_vector.y = r_sin_theta(range,theta);
  add_vector_localization(threshmap,localization_map,
                                    epsilon,range_vector);
  return(SCHTRUE); /* SCHTRUE is the predefined SCHEME value #t. */
}
```

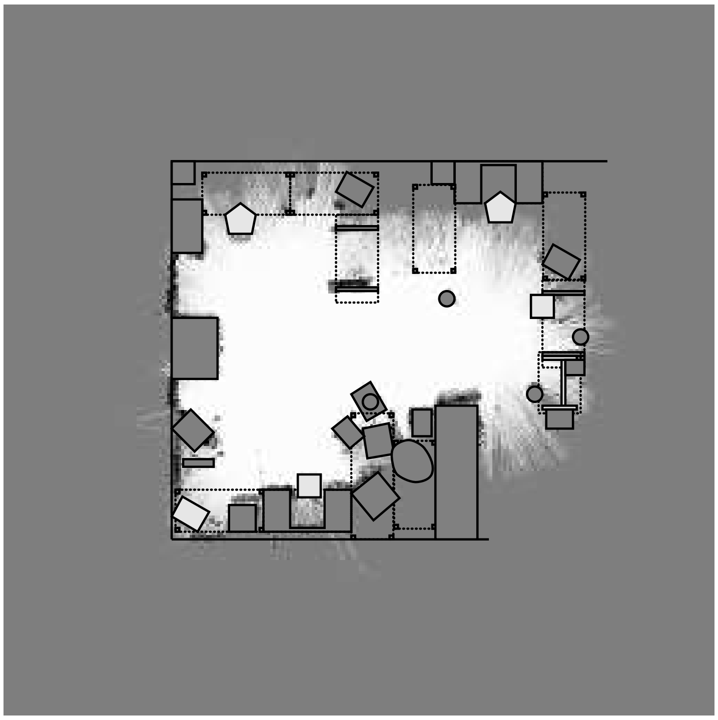
Figure 3.1: Scheme code enabling a Scheme call to a C subroutine and a C subroutine that can be called from Scheme.

"empty" grids and update the grids using a Bayesian probability rule. We prefer to maintain a single grid where all cells start out at 50 percent (no knowledge about whether the cell tends toward full or empty), and adjust the probabilities upward and downward between 0 and 1. In fact, we maintain the grid as a two-dimensional array of bytes, and give each cell a probability value between 0 and 255. One advantage of this structure is that our map is structurally equivalent to a gray-scale video image, allowing us to use computer vision primitives such as smoothing and edge detection on the map. This is useful on those occasions when we want to extract features (such as edges and corners) from the map. Another advantage of the unified structure is that it makes it easy to convert the statistical map to a bitmap. We can threshold the map at any certainty level we like; what threshold is appropriate depends on whether the particular application requires a map that is conservative about obstacles or about freespace. Figure 3.2 shows a sample map of our lab, made using our statistical occupancy grid implementation and the three-beam laser rangefinder. In this picture, the lighter a cell is, the more it is considered to be empty; conversely, the darker a cell is, the more certainly it is considered to be occupied. The medium gray regions around the periphery of the map are areas in the map corresponding to parts of the world to which the robot has not been. Those cells still have the original 50 percent occupancy value that can be considered to mean "no information". Figure 3.3 shows the same map (a) smoothed using a gaussian filter and (b) edge detected using the Canny edge-detection algorithm ([Can86]).

The implemention of our mapmaking package makes it easy to change the rules used to update the map. We have tried a number of update rules at different times including a simple binary update, where occupancy probability is either 0 or 1, and a linear update, where occupancy probability for a cell varies between 0 and 1 by units of $\frac{1}{k}$, increasing or decreasing by one unit, appropriately, every time that cell is updated. When we tried this mapmaking system with wide-angle sonar sensors (like those used by Moravec and Elfes), we also used update rules that changed the contents of a cell proportionally to its angular distance from the center line of the sonar's sensitivity region and to its linear distance from the sonar. The update rule that we have found to work best is an exponential-decay update rule: probability is raised by the rule $p_{new}(x) = 1-(\alpha(1-p_{old}(x)))$ and lowered by the rule $p_{new}(x) = \alpha p_{old}(x)$. In practice, we use distinct α_{raise} and α_{lower} values, for raising and lowering probabilities, respectively. Typical values are 0.3 for α_{raise} and 0.15 for α_{lower}. Also, on the mobot, since we implement the occupancy grid as an array

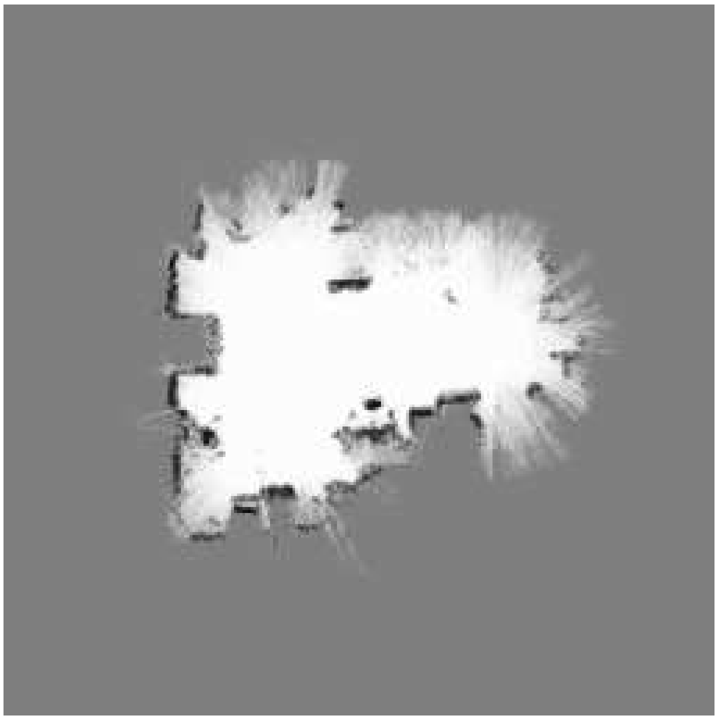


(a)

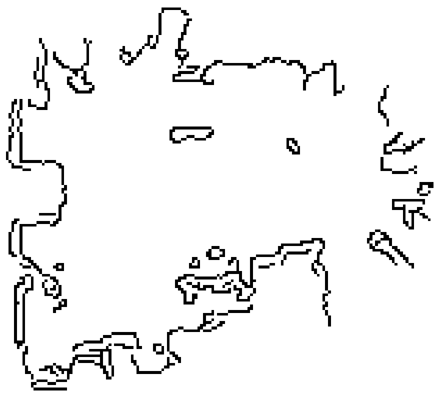


(b)

Figure 3.2: (a) An occupancy grid representing part of our lab, built using the three-beam laser rangefinder. (b) The grid is shown with a hand-drawn map overlaid.



(a)



(b)

Figure 3.3: The same map (a) smoothed with a Gaussian filter, and (b) edge-detected using the Canny edge-detection algorithm.

of bytes, it is adequate to implement the update rules as 256-entry lookup-tables, considerably improving the performance of the update procedures.

3.2.2 Maps that Track Time-Variability of Locations

This section concentrates on an enhancement of the map-making process that allows the robot access to information about the time-varying behavior of various parts of its environment. In other words, the robot gathers information about what parts of its environment are likely to change (for example, doors and chairs, as opposed to walls). The enhancement is presented as a modification of the statistical occupancy grid algorithm (since that is what we actually use in our system), but the idea of annotating map features with time-variability values could be applied to many different world representations.

We propose adding the following to the grid maintenance algorithm: For each cell in the grid, keep track not only of probability of occupancy, but of the derivative, with respect to time, of that probability. To be more precise, let ρ be the statistical occupancy grid representing a given environment. $\rho(x, y, t)$ is the probability of occupancy of cell (x, y) at time t. We want to keep track of the time derivative of this function, $\dot{\rho} = \frac{\partial \rho}{\partial t}$. $\dot{\rho}(x, y, t)$ is the rate at which cell (x, y) is changing at time t. Now, define Γ to be the time integral of the absolute value of $\dot{\rho}$:

$$\Gamma(x, y, t) = \int_{t-t_0}^{t} \mid \dot{\rho}(x, y, \tau) \mid d\tau \tag{3.1}$$

$\Gamma(x, y, t)$ is a measure of the total amount of change in cell (x, y) over some specified time interval t_0. In practice, we approximate (3.1) using finite differences. If we were to compare Γ for cells in a grid to the "real world" areas they represent, we would expect to find the following: Cabinets and walls (that seldom move) would have very small values of Γ; the same should be true for open spaces that are seldom occupied. On the other hand, a group of cells with large values for Γ, would likely be doors or chairs, for example. One can configure the robot's planner to treat spaces that, at a current time t_1 have low values for $\rho(x, y, t_1)$ but high values for $\Gamma(x, y, t_1)$ as dangerous. Unless information on such areas is very recent or the robot can directly sense that those areas are, in fact, open, the planner should avoid them. This information is very useful to a navigational planner; the planner can plan its strategy for navigating from one place to another based on the current values of Γ: If, in the map that represents the current value of ρ, there is an open path on which all cells have low Γ values, then the navigator can choose to follow that path, with reasonable confidence that it will be able to achieve its objective (this can be

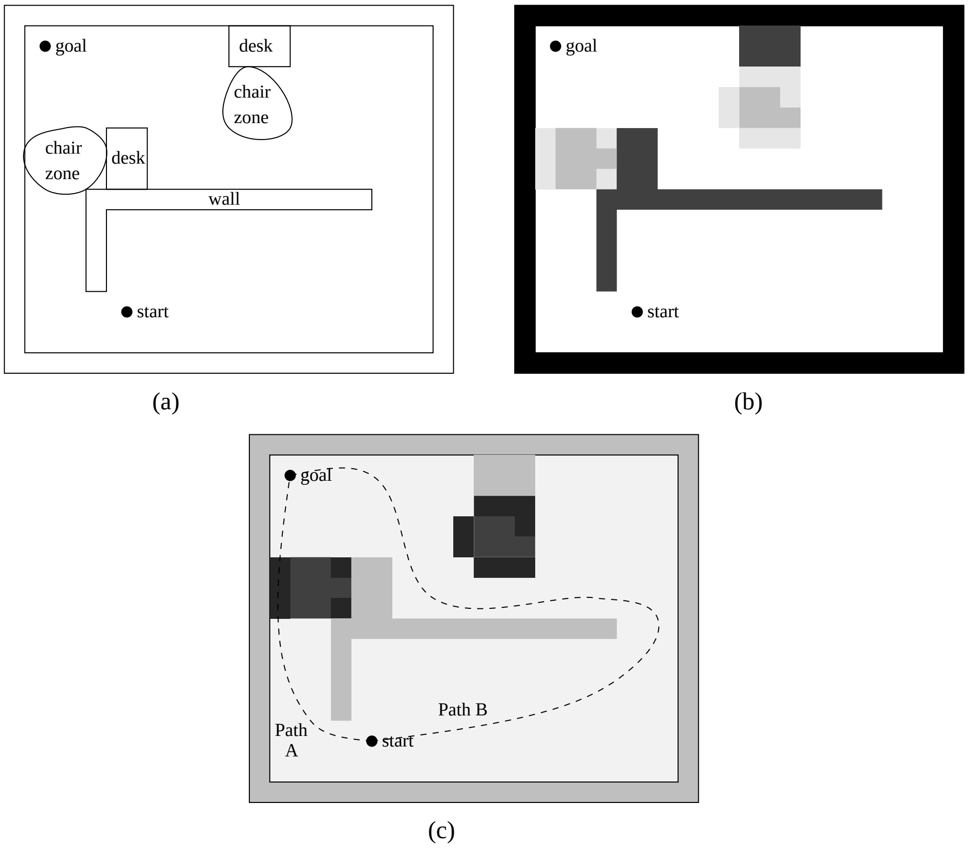


Figure 3.4: Time-variability information is very useful to a navigational planner. In (a), the "chair zones" are large regions that contain small desk chairs. Since the chairs move within the zones, the time-variability of the occupancy grid in those regions is high. (b) shows the occupancy grid corresponding to the map in (a). (c) shows time-variability information for the occupancy grid in (b). Path A is a possible start-to-goal path, but it goes through a highly time-variable region. Path B is a longer path, but stays in regions for which we have high confidence, not only that they *were* open, but also that they still *are* open.

implemented easily by thresholding on the value of Γ). If, on the other hand, the shortest path to the goal passes through cells with high Γ values, the planner will need to plan an experiment to determine whether those cells are, in fact, currently open. If the cells are currently open in the world, the robot can proceed; otherwise it must plan another path to its goal. Figure 3.4 illustrates this concept.

3.2.3 Tactical Considerations: Making Maps using a Laser Rangefinder Mounted on a Wheeled Base

3.2.3.1 Using the Three-Beam Ranger to Make Maps

Given that we're using a real sensor that has limited range and that lacks hard error bounds, we'd like to use the sensor in a way that allows it to be as reliable and accurate as possible. We would like the ranger's dominant failure mode to be the production of "clearly impossible" readings; furthermore, we'd like as large a portion of the sensor's incorrect readings as possible to fall into that category. In other words, since we can't guarantee correct readings, we'd like to maximize the percentage of readings that are either correct or "no information". The justification is that we don't want to make decisions based on believed-correct range values that are, in fact, erroneous. Our motivation is similar to that used by Donald in [Don89] for developing Error-Detection and Recovery plans: it may be impossible to generate guaranteed plans (sensor readings), but we may be able to generate plans that are guarantee either to succeed (reach the goal) or recognizably fail (reach a failure region that is recognizeable and distinguishable from the goal). If we reach the failure region, we can generate a new plan for reaching the goal from that region.

Our mechanism for achieving this property is to limit the maximum correct range reading. We do this by tilting the ranger toward the floor: we incline the pan-tilt head so that the laser beams strike the floor about two meters from the robot. This has several advantages:

- We know that any range reading over about two meters (the actual distance along the beams will be a little more) is incorrect, and can disregard it.
- Less ambient light, particularly from windows, enters the field of view of the ranger, increasing its ability to detect the laser spots accurately.
- Since the amount of light from each reflected beam enters the camera in an amount that is inversely proportional to the square of the distance from the laser spot to the aperture, range readings at smaller ranges are more likely to be detected correctly. Thus, we reduce the number of overall errors.
- A fourth issue is unrelated to ranger accuracy, but a concern nonetheless: the knowledge that there is uncontained laser light in a room tends to make people nervous. It is helpful, therefore, to keep the beams aimed low, as this reduces the odds that a beam will bother someone.

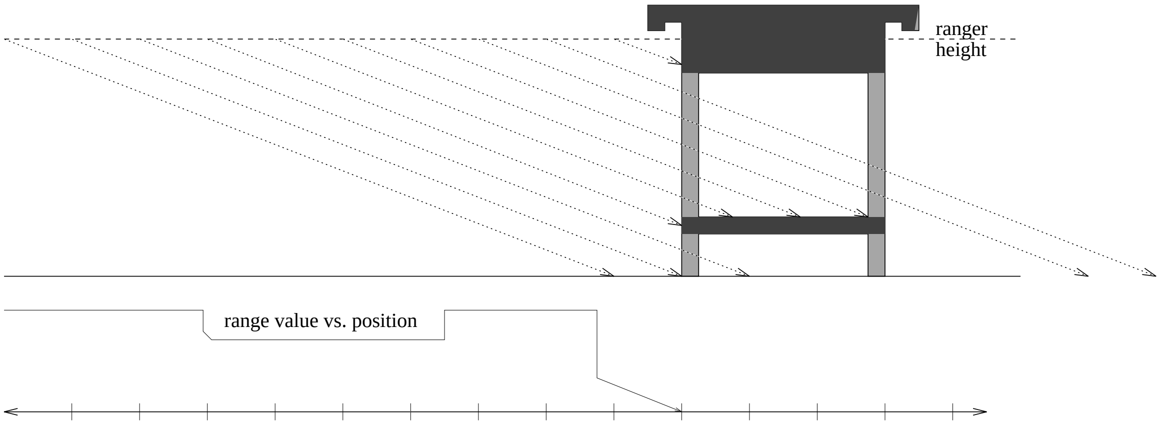


Figure 3.5: One problem with the tilted rangefinder: the world is not planar. A table with a low shelf appears to move as the robot approaches it, because the height where the beam intersects the table changes as the robot moves. (Darker portions of the table are those that the robot can see.)

There are a couple of disadvantages to this tactic. One is that it is possible for the robot to end up "in the middle of nowhere." There can easily be areas in the middle of labs and large offices where there are no detectable features to a ranger with a (hard) maximum range of two meters. A more distressing difficulty is the following: We are regarding the world as planar. Thus, we treat all objects in the world as extruded planar shapes. We can justify this treatment if the ranger is level: if the ranger moves toward something, its beams will intersect the object in the same place, so the distance to the object will shrink linearly with distance. If we tilt the ranger, though, this is no longer the case. Consider a cart with a low shelf (like the one in figure 3.5). At long range, the robot sees the floor beneath the card. As the robot approaches, the apparent distance begins to reduce linearly as the beam climbs the side of the shelf. As the beams travel across the top of the shelf, the apparent distance remains constant. As the robot continues to approach the cart, it may see floor on the other side of the cart, then the edge of the top shelf of the cart, then the top of the cart. Our mapmaking techniques must be robust in the face of this behavior.

3.2.3.2 Minimizing Error Introduced through Odometry

A natural question that will be asked about the way we make maps is, "*If odometry is so bad, then how can you make accurate maps based on it?*" This is a reasonable question, since we need a good map to do localization well, but need good localization to make good maps. The key to this apparent inconsistency lies in the way one uses odometry. A synchro-drive wheelbase is capable of extremely accurate odometry under limited circumstances; ours is good to within a couple of millimeters on a single translational motion, and to within a degree on a single rotational motion. Errors arise from three sources: combined motions, incremental error, and gross slip. When the robot is required to follow a non-straight trajectory, there is an unpredictable interaction between the translation and rotation components of the motion; this partially results from the inevitable small slippage that occurs when the (identically driven) wheels of the base follow different length paths. Incremental error is the sum of two types of error that occur primarily during single-component motions: the robot can't really turn on a point; some translation occurs during any rotation, and the magnitude of that translation depends at least partially on the microfrictional properties of the floor. Similarly, no translation follows a perfectly straight line; some rotation always occurs, due largely to an inevitably imperfect wheel alignment. Finally, the assumption that all motions made by the robot register on its odometers becomes completely untenable if the robot is performing manipulation. If the robot pushes a large box, for example, there is some chance that its wheels will slip due to greater frictional force between the box and the floor than between the robot and the floor. This kind of gross slip introduces large errors into the robot's rotational and translational pose estimates.

We deal with these errors by the following tactics: When it is critical to maintain accurate pose estimates, we avoid actions that promote combined-motion and gross-slip errors. In other words, we allow the robot only to execute pure translations and pure rotations, and we avoid high-force motions that are subject to loss of frictional contact with the floor. To minimize the second source of error, we try, at least initially, to keep the number of separate motions made by the robot as small as we reasonably can. After the robot has built enough map to perform accurate localization, we can allow motions and activities that have higher potential for odometric error, as long as we take care to maximize the quality of our pose estimate whenever we change the map.

3.3 Path Planning and Execution

Making maps and performing localization is not necessarily a valuable robot skill, solely for its own sake. We want maps and the ability to localize so that our robots can perform useful tasks. If the desired task is in the family of tasks "take x from y to z," then the robot will need to be able to navigate to and from locations such as y and z. At some level, therefore, the robot will need some sort of path planner/plan executor system. It is difficult for current robots to recognize abstract concepts such as *room* or *hallway*, either from direct sensing of the world or from analysis of an automatically generated geometric map. We prefer to implement our path-planning algorithms using representations the robot can directly build from its physical observables. For this reason, we've chosen to use planning based directly on the geometry of the robot's automatically generated map.

3.3.1 The Floodfill Path Planner

One frequently used technique for planning paths to a goal is the *potential field*. In general, a potential field is a function over a space that defines for each point in that space how far it is from some defined ideal state or maximally "bad" state. In path planning, two types of potential fields dominate: when the task is obstacle avoidance, the potential for a given configuration is a function of the distance from the nearest obstacle (often $1/(\| d \|)^k$, for some large k). In this case, the objective is to minimize the potential, thereby staying as far away from obstacles as possible. When the task is to achieve a goal state, the potential is a function of the distance from the goal (this may or may not take into account obstacles that might be in the way). [Lat92] devotes a chapter to the history of the use of potential fields in robotics; therefore, we will only mention a few works, some of which have had particular bearing on our system: [BL91] describe a multiscale rasterized potential field planner that does motion planning for high degree-of-freedom robots. *Multiscale*, here, means that they start planning on a coarse grid and lower the granularity as necessary (subject to a maximum) to find a valid plan. [LRDG90] describe a rasterized potential field planner that uses a configuration-space grid and specialized graphics hardware to make fast plans for $\mathbb{R}^2 \times \mathrm{S}^1$ and $\mathbb{R}^{2.5} \times \mathrm{S}^1$ (three-dimensional objects with vertical faces that are able to translate and rotate in one plane) planning problems. Note that both of these systems are rasterized. Other works that discuss potential-field methods include [Kha80,KV88,Hog88,Kho87,Buc86].

[BL91] designed a motion-planning system in which a combination of the "distance from obstacle" potential field and the "distance from goal" potential field is used to compute a potential field that will guide a point robot in real-space into a goal by following the path of lowest potential. They extend this planner to non-point robots by placing *control points* on the robot: a control point is a particular point on the robot that is required to follow the path of lowest potential to the goal. If there are multiple attractors, the path of lowest potential is determined by adding the potentials at each of the attractors on the robot. The main problem here is that once multiple attractors are involved and the robot is no longer a point, the possibility exists that there will be *local minima* in the potential field. Local minimum means the same thing here that it does for any function: a place where the function value is less than it is at any other point in an arbitrarily small region around that point. In this case, it also means a place that a robot following the path of least potential will not be able to leave, because that path ends at this non-goal minimum. [BL91] handle this difficulty by a variety of methods that seem to work extremely well in practice, including Monte Carlo methods, in which the robot is displaced along a random path until it descends into the global minimum at the goal, and also including deterministic methods that, in essence, "fill in" local minima as the robot encounters them.

An alternate means of avoiding local minima is to eliminate them by using a configuration space planner. [LRDG90] describes a motion-planning system that does this by computing the configuration space of the robot and the obstacles, and creating a potential field that has its global minimum at the goal (which in this case is both a position and an orientation). In the $\mathbb{R}^2 \times \mathrm{S}^1$ case, the input to their algorithm is a set of polygonal obstacles, which they immediately convert to a planar rasterized map (a two-dimensional bitmap) and a rasterized representation of the robot. The first step of their planner is to compute an $\mathbb{R}^2 \times \mathrm{S}^1$ rasterized configuration space. For a map of size $n \times n$, a slice of the configuration space is computed at m different theta-slices, yielding a configuration space with resolution $n \times n \times m$. C-space slice θ_i is generated by discrete convolution of the obstacle map with the robot rotated by θ_i around its reference point. Once the C-space is computed, they compute a potential field from every point in freespace to the specified goal. This operation is done using a floodfill algorithm, similar to those discussed in 2.3.2.2. The goal cell is assigned the value 0, its neighbors get the value 1, and so on until there are no cells left to number. Note that cells that are in a different connected component of the configuration space than the goal are never numbered at all. Since a configuration from which there is no path to the goal can be recognized immediately, the path executor can report failure immediately in such a case. The path planning system we use on LILY is based on this planner. Our planner is considerably simplified, but uses the same basic algorithms. Since LILY is rotationally symmetric, we are able to restrict our path planning to $\mathbb{R}^2$, significantly reducing the time and space requirements for the algorithm. We compute the C-space either by convolving the map with a disc or by $L_{octagon}$-flooding the obstacles by the radius of the robot. We then compute the potential field by a floodfill using the same method as [LRDG90], though without any special hardware.

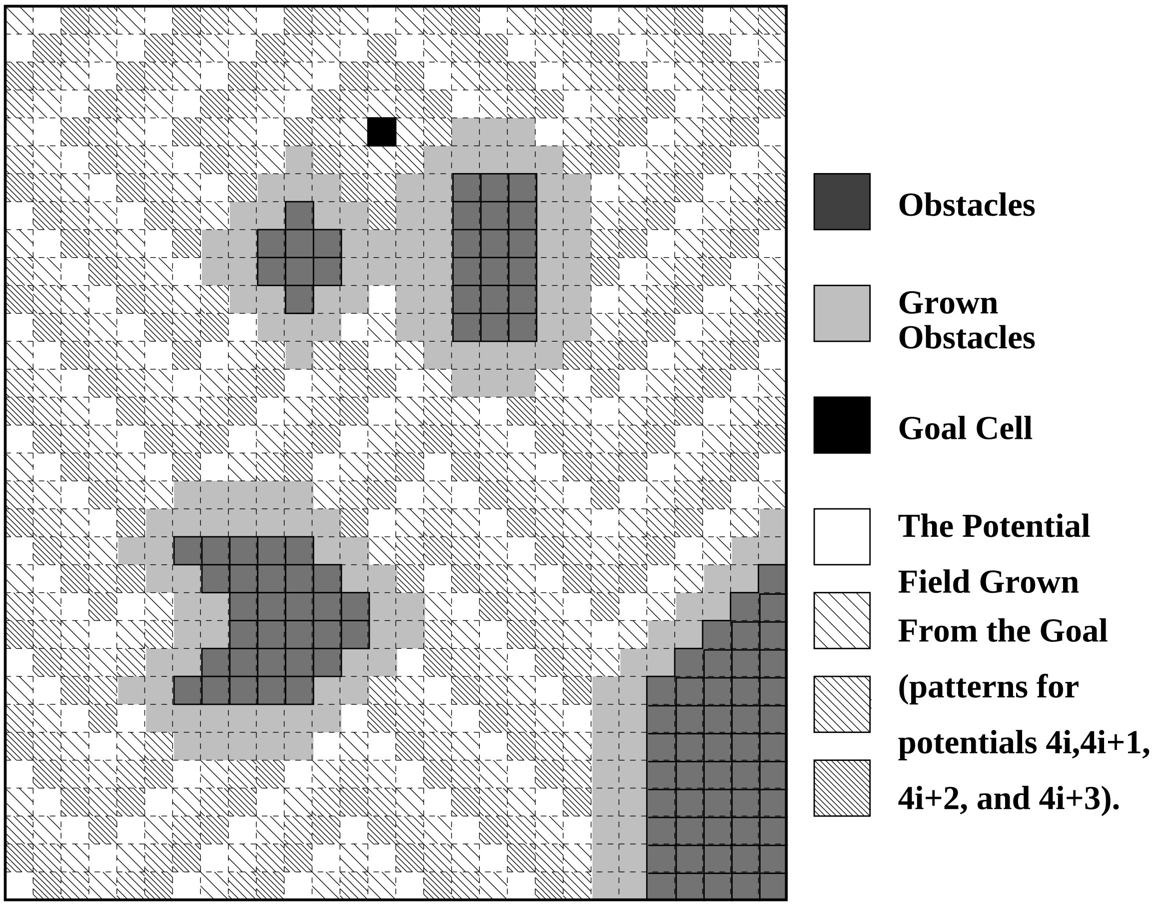


Figure 3.6: A rasterized map, grown by the radius of a robot, shown with the potential field grown from the marked goal. We show the potential values modulo 4 to better display the individual equipotential contours.

Once the potential field has been grown for a given goal, the “plan” to get from a specified start location to that goal is simply to travel from cell to cell, always heading toward the lowest numbered neighbor, until either the goal is reached or a planned motion fails. In figure 3.6, we've shown a potential field grown according to an L_1 distance metric, by flooding from a given cell to its unnumbered 4-connected neighbors. The reason for selecting that metric is that corner-connected neighbors may not always be reachable (since going from a cell to a corner-connected cell may cause the robot to hit an obstacle in one of those cells' common edge-connected neighbors. The drawback to the L_1 potential field is that paths through the field are often more zig-zagged than strictly necessary: This is particularly true of the path to the goal that is “closest” to the Euclidean shortest path. We reduce this tendency by causing the robot to favor motions that reduce its potential while continuing in the direction it already faces. This lets the robot make longer straight line motions and increases the speed and accuracy with which the robot executes its plans.

3.3.2 Recovering from Plan Failure

A mobile robot cannot always follow a plan generated by its path planner. Unexpected obstacles will appear from time to time, due to two main sources: map error and navigation error. Map error (the map does not correctly reflect the world) is probably the easier of the two to handle: If an unexpected obstacle appears, the robot can attempt to go around it in some way (cf. Lumelsky [Lum86]). For the path planner just described, the appropriate action is to modify the potential field by raising the potential of the cell in which the obstacle was encountered to the large (infinite) value assigned to obstacle cells elsewhere in the map. The robot then attempts the following actions: First, the robot attempts to find a cell adjacent to its current one that has a lower potential value than the current cell. If it succeeds, then it continues along this new path. If it cannot, then it updates the obstacle cell in the original map and regenerates the plan to the goal. It then attempts to follow the new plan. This is guaranteed to reach the goal eventually if a path to the goal exists (subject to the resolution of the grid).

Navigation error (the robot's pose estimate is incorrect) cannot be handled by this method: if the robot's pose estimate is off, it won't be able to achieve the goal by following the plan at all – the best it can do is to *think* it has achieved the goal. This is where the localization algorithms described in chapter 2 tie into the system. When evidence is encountered (such as an unexpected obstacle) suggesting error in the system, the robot must consider the hypothesis that there is navigation error. To determine whether there is such an error, we use the localization algorithm: If the robot's current estimate of pose turns out to be one of the most feasible (i.e., it matches most of the range probes with a reasonable uncertainty ball), then the robot can reasonably conclude that its pose is the correct one and that an obstacle has been added to the world. If, however, the most feasible pose is significantly different from the pose estimate the robot held prior to running the localization procedure, then a reasonable conclusion is that the robot's pose estimate was in error. In this case, the proper response is to update the robot's pose estimate and attempt to follow a path to the goal from the robot's current position by following the original potential field from the point in the map that corresponds to the new pose estimate.

We don't necessarily want to wait until we detect an error condition before we perform a localization procedure. There is a strong relationship between the accuracy of the pose estimate and the accuracy of maps made under that estimate. Of course, we can't just run the localization procedure after every motion, for a number of reasons: if we begin localization before sufficient map detail is available, we won't get satisfactory results. In addition, the localization procedure is computationally somewhat expensive to run; we don't want to do it more often than is necessary. We do not yet know the exact relationship between localization quality and map quality; nor do we know how often we should execute the localization procedure. How often a given robot should localize depends on the instantaneous and extended precision and accuracy of its base and its range sensor.

Navigation errors cannot be avoided; moreover, they cannot be anticipated. In addition, we cannot sense them *when* they happen, and often detect them well after their occurrence. This makes it extremely difficult to produce guaranteed strategies for maintaining map correctness. This is somewhat similar to the difficulties considered by Donald in [Don89], whose planner produces plans that cause the robot either to achieve its goal recognizeably or to fail recognizeably. We want to use strategies that allow the robot either to keep a correct map or to fail recognizeably (in which case it can make a new plan). Donald's plans will either achieve the goal or fail recognizeably, assuming a robot with bounded sensor error and control error. In the mobile robotic domain, we cannot use the same strategies, since position error can be unboundedly large. For maximal map correctness, we need to insure that we don't make large changes to our map based on grossly false pose estimates. Since we can't guarantee that this won't happen, we want the ability to undo any damage that we might do to our map as soon as we discover that our pose estimate is incorrect. Even this is beyond current capability, since the exact moment when we began making erroneous map updates is unknown. What we can do is to keep backups of our maps. Just as we make backup copies of program code and important data files, the robot can keep one or more copies of its map, so that it can revert to a previous version when it determines that it has made improper and destructive updates to its current map. We have not implemented this technique on Lily, primarily due to memory limitations, but we have explored methods for determining when the map should be reverted to a backup state because of gross disagreement between the map and the world. One way to determine this level of disagreement is through application of the localization algorithm: if the localization algorithm returns a drastically different pose than our current estimate, then recent map updates are likely to have been in error. Another approach would be to use the time-varying map information described in section 3.2.2, possibly by also tracking second derivative information: if a large number of updates occur over a short period of time in a region that had been previously sensed to be stable over time, this might be evidence that something incorrect was happening. This would not *necessarily* mean that we should throw out the most recent map, but would certainly indicate that a localization pass is needed to determine whether an error condition existed.

3.4 Miscellaneous Feature Detection Operations

There are several ways we can go about detecting features such as corners and straight edges. One way would be to use computer vision/image processing operators on our rasterized maps. There are three main difficulties with using this idea, at least in general: Functions that operate on the entire map are often too expensive to use for everything, particularly when we are interested in primarily local information. We may want information that is of finer granularity than that provided by our rasterized map, since it has fixed resolution. Finally, and most importantly, we want a mechanism for detecting local geometric features that is independent of both the current pose estimate and the current map. This is primarily because we may be using these feature detection primitives to double-check the localization algorithm or as part of its input. For this reason, we've looked at separate procedures for detecting convex and concave corners and for detecting and measuring straight edges in the environment. These primitives are fairly simple, but it is useful to know about them and how they work.

3.4.1 Finding and Tracking Corners

We have incorporated a number of minor functions into the Scheme part of our system; most of these are of the form "find feature x in the range (θ_0, θ_1)," where x is a feature, such as a corner or nearest point to the robot, and (θ_0, θ_1) is a range of angles relative to the robot where it is to look for x. We define the *distance function* to be a function mapping the heading of the robot's rangefinder to the distance value it returns, for a fixed position and at a fixed time. The features we look for are primarily corners: *convex corners*, *concave corners*, and *jump corners*. Convex corners are local minima in the distance function at which large changes in surface normal occur; these are typically the corners of convex objects in the world, such as couches and boxes. Concave corners are local maxima in the distance function at which large changes in surface normal occur; these are usually junctions where two objects are adjacent. A jump corner usually marks an occlusion boundary, where the end of one object covers part of another. Jump corners are usually locations at which the distance function jumps significantly between one reading and the next, either to another finite value, or to "no reading". These features are mostly useful for localization and mapmaking, but can also be used as part of a model acquisition and recognition system such as the one suggested by [JR93], particularly since we have primitives analogous to the `measure-edge` and `find-next-face` that they describe. We will discuss edge measurement in section 3.4.2, as it has applications to rotational localization. First, however, we will discuss the usefulness of these corner-detection primitives.

3.4.1.1 Corner and Object Tracking Applications to Localization and Mapmaking

In chapter 2, our discussion of localization algorithms omitted the following question: how do we select range probes for input to the localization algorithm? The optimal set of range probes is obviously "the set that works the best" (i. e., the set that produces the smallest nonempty set of feasible poses with the smallest uncertainty ball). Like many problems in computer science, however, the optimal result is uncomputable: determining the optimal set of range probes requires trying all possible sets of range probes and selecting the best. From a theoretical standpoint, we'd like to define "best" as "the best we can do with a given number of range probes (where that number may be a function of the environment size)." Because the robot lives in the physical world, this, too, is beyond our capability. From a practical standpoint, the best we can do is to select range probes in a sensible, systematic fashion, and set a course of action for the robot to use when the initially chosen set of range probes fails to produce a satisfactory feasible pose set.

Two approaches are "take a range probe every k degrees around the robot (for some k)," and "identify features that may help reduce the size of the feasible pose set quickly." The first approach is feasible, but may be unnecessarily expensive. If the robot is in an area where there is little within the rangefinder's effective range, this may require slicing the circle up into a large number of pieces, before the localization algorithm can be provided with sufficient usable range probes. More careful selection of probing directions can't improve the quality of localization in such a case but can significantly reduce the cost of obtaining the best estimate for the robot's current pose. In any event, we would like to select probe directions in a more careful way than uniform angular spacing. This leads us to the second approach. The main features we can easily detect with our ranger are minima and maxima in the distance function. These are essentially the convex and concave corners discussed earlier in this section (these features are analogous to the *landmarks* and *beacons* used for navigation by, among others, Mataric ([Mat90]) and Leonard, Durrant-Whyte, and Cox ([LDWC90])). We propose the following algorithm:

1. Perform a low-speed, high-density circular scan of the laser rangefinder and select convex and concave corners from the output of that scan.

2. Eliminate corners that are angularly too close together from the point of view of the robot.

3. Sort remaining features by increasing distance from the robot and discard all but the closest k of them.

4. Perform the localization procedure as outlined in chapter 2 with the range probes to those k features.

5. If the results are not satisfactory, retrieve the eliminated features and localize based on them as well.

3.4.2 Reducing Rotational Uncertainty: Aligning to Environment Edges

In chapter 2, we alluded to a method of rotational localization (determination of absolute heading) that is separate from the localization methods discussed in that chapter. This method, which we now discuss in more detail, uses data from the laser rangefinder to align the robot to particular edges in the environment. If we align the robot to an environment edge and then associate that edge with its representation in the map, then we can establish the robot's orientation. The accuracy of this detected orientation is a function of the accuracy with which we can align to the edge and of the accuracy of the edge's representation in its map.

3.4.2.1 Detecting Long Edges in the Environment

The main data structure we use for edge detection and alignment is the *comb*. A comb is basically a one-dimensional map of part of the environment. Although it is implemented simply as a vector of (position.distance) dotted pairs, it gets its name from the way it is visualized: namely as a series of parallel range probes going from the robot's straight-line trajectory to the nearest objects in a direction normal to that trajectory. Figure 3.7b shows a comb that might be obtained by a robot following the shown trajectory past the set of objects shown in figure 3.7a. We detect edges in a comb using a one-dimensional analogue to the edge-detection algorithms used in computer vision (such as [MH79] and [Can86]). We first filter out of the comb any elements that correspond to "unlikely" readings (those longer than the maximum effective range, and those shorter than the ranger's minimum range) (figure 3.7c). We then differentiate the comb (actually, take a finite difference (figure 3.8d)), and threshold on the difference value, discarding those with large magnitudes (figure 3.8e). Finally, we use the positions and signs of the remaining difference comb entries to detect regions within which there is no large change in distance. We can then take, for example, the longest interval (the one indicated in figure 3.8e), and compute its relative orientation to the robot. We then rotate the robot by the negative of that orientation, bringing the robot into alignment with the environment edge that corresponds to the interval. Using this tactic, it is possible to get alignment accuracy on the close order of one degree.

3.4.2.2 Using Edge Alignment to Reduce Rotational Uncertainty

The ability to find, measure, and align to long edges in the environment has a number of applications to mobile robot localization. An obvious application of this capability is to perform $\mathbb{R}^2 \times \mathrm{S}^1$ localization using techniques borrowed from the model-based recognition literature in computer vision. Model-based recognition is, generally speaking, the attempt to find objects in images using *a priori* models of those objects being looked for. One common mode of model-based recognition attempts to match models whose features are edges against the results of edge-detecting the input image. The most well known example of this sort of algorithm is the *interpretation tree* method ([GLP87]). This method matches a model edge against each image edge, then recursively attempts to match adjacent model edges against image edges in a way that is pairwise consistent (by match, here, we mean "select a transformation that turns the one into the other"). More recent matching methods used for model-based recognition include the *alignment method* ([HU90]), which transforms triads of model vertices to coincide with image vertices, and methods based on the *Minimum Hausdorff Distance* ([HK90,Ruc95]), which transforms model points to lie near image points. We can apply these same algorithms in the context of mobile robot localization, using our current model of mapmaking and sensing: our model consists of edges that the robot can detect (by building combs from laser rangefinder data) in its vicinity, and our edge-map is the output of an edge detector on our rasterized map.

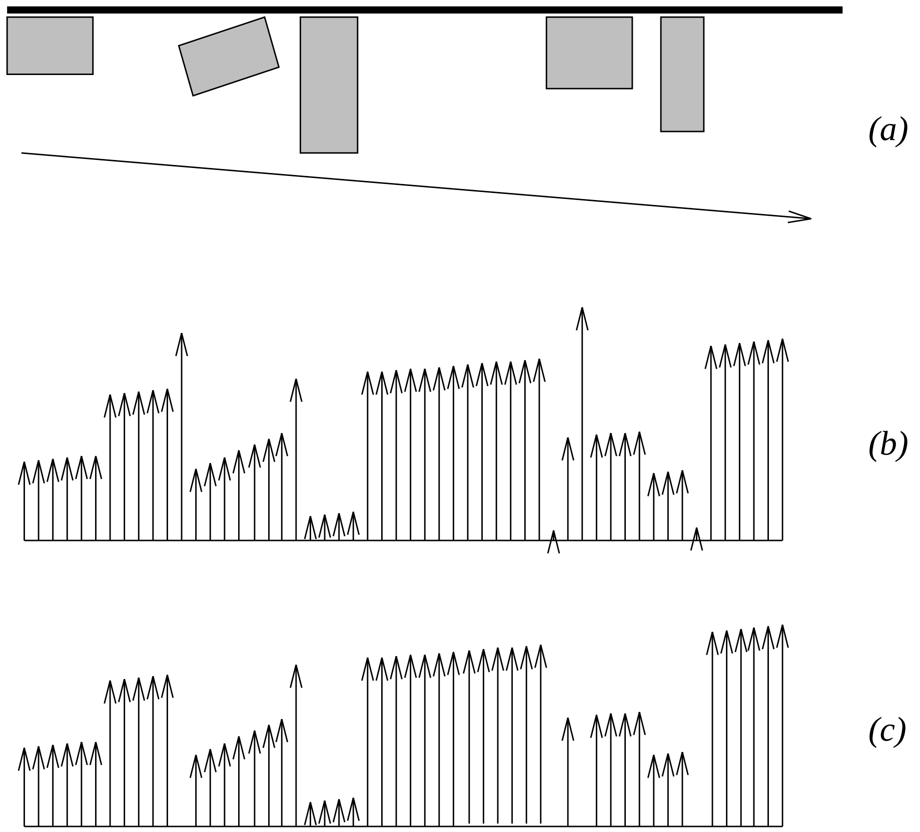


Figure 3.7: (a). The robot travels along the trajectory marked by the arrow, measuring the distance to the objects shown, or to the wall behind them. (b). The comb obtained during the pass. (c). The comb, with spurious values (those too large or too small for the ranger) discarded

We prefer not to depend on a localization procedure of the type just described for a variety of reasons. One reason is that most model-based-recognition algorithms are asymptotically very expensive[2] particularly if we allow for missing edges in the image. Another is that it is very difficult to characterize the behavior of such algorithms in terms of how their behavior degrades with sensor error. Finally, we noted in chapter 2 that the concepts of edges and polygons are not really compatible with the sensor models that we actually have. While the comb technique is a piece of evidence against that assertion, it is an expensive technique to use for general edge measuring (since the robot physically moves and takes a large number of range measurements), and only works for some edges (those corresponding to straight,

[2]Rucklidge, in [Ruc95], discusses a very fast model-based-recognition system based on a rasterized implementation of the Minimum Hausdorff Distance ([HK93]). We regard this as the first practical method for model-based-recognition.

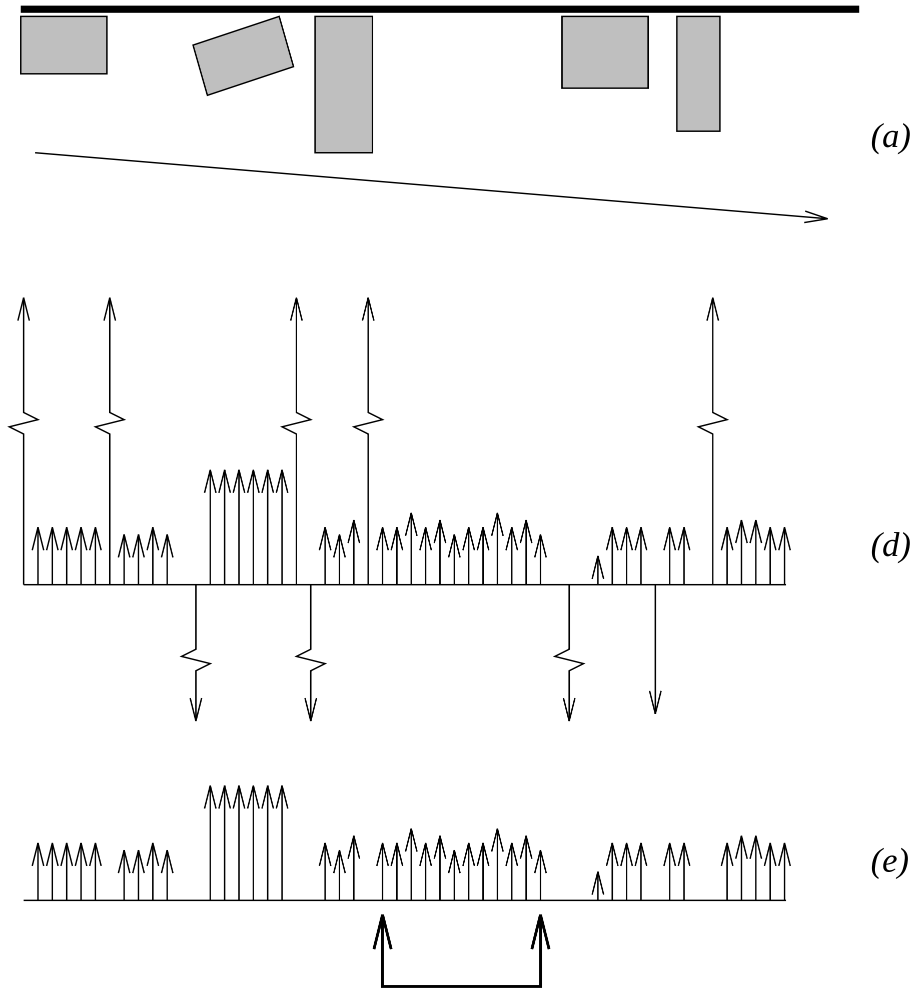


Figure 3.8: (a). The original trajectory (d). We've taken the difference between each element of the comb and its neighbor. The zagged arrows are extremely large differences, such as found at an object boundary. (e). Comb elements are separated into contiguous groups of like-valued differences containing no extremely large difference values. Each group corresponds to an approximately smooth world edge.

---

vertical faces in the environment). It's a reasonable technique to use for applications where only one or two edges need to be measured.

Our preferred application for edge alignment is for rotational localization only. In fact, the simpler the application of edge alignment to rotational localization, the better. The simplest way we can use this tactic only needs a single long edge. In section 2.2.3.2, we introduced the notion of performing $\mathbb{R}^2 \times \mathrm{S}^1$ localization by slicing the unit circle up into discrete intervals and looking for the slice in which the planar localization algorithm required the smallest uncertainty value. The way that one would first think to divide the circle into intervals would be into fixed-size intervals, say every k degrees. This works, but a more efficient way would be to select a set of candidate angular intervals that can be searched for the best $\mathbb{R}^2$ localization. One way to select those candidate intervals is to locate a long straight edge in the environment, measure it, then search the map (or an edge detected version thereof) for straight edges of approximately the same length as the measured edge. The relative orientation of those map edges to the measured edge can be used as offsets to the $\mathbb{R}^2$ localization algorithm: if the robot is aligned to a particular measured edge in the environment, and its current orientation estimate is θ_0, and there is an edge in the map whose orientation is θ_1, with approximately the same length as the measured edge, then it is reasonable to examine how well the localization algorithm runs if we add $\theta_1 - \theta_0$ to the angular offset of each of the range probes. If there is a pose (x_f, y_f) with that offset that is feasible within a small error bound, then we can conclude that the pose (x_f, y_f, θ_1) is a feasible pose in $\mathbb{R}^2 \times \mathrm{S}^1$, and that our original orientation estimate may be off by $\theta_1 - \theta_0$. If there are few long edges in the environment, or if they have a small set of possible orientations, this method of $\mathbb{R}^2 \times \mathrm{S}^1$ localization may be significantly faster than the method of slicing the unit circle up into many small intervals.

3.5 Conclusion

The purpose of this chapter was to discuss Lily's mapmaking and localization system and its subsystems. The system consists of several pieces, some of which are implemented in Jonathan Rees's Mobot Scheme ([RD92]), and some of which are implemented in C and tied into the Scheme system as external calls. This multi-language implementation buys us the flexibility of interpreted Scheme for programming ease and the performance of C for performance-critical parts of the system. We described the rasterized mapmaking system, which is loosely based on Moravec and Elfes's *Statistical Occupancy Grid*, and the rasterized path planner that is based directly on [LRDG90], and related to [BL91]. We discussed how the mapmaker, the localization module, and the path planner interact with each other under the common framework of rasterized computational geometry. Finally, we discussed other functions in the system, which do not depend on the rasterized framework. Part of that section dealt with a mechanism for determining which range probes are most effective for quick and accurate localization; the rest discussed a method for rotational localization using a tactic for accurately measuring and aligning to long straight edges. We will examine the behavior of this system in real situations in chapter 6.

Chapter 4

The Cornell Mobile Robots

4.1 Introduction: Modular Design for Robustness and Flexibility

4.1.1 A Manifesto for Flexible, Easy to Use Mobile Robots

When the Cornell Robotics and Vision Laboratory began working in the field of mobile robots, we made a conscious decision to build robots that were sturdy, reliable, flexible, and easy to use. It is our perception that university-built mobile robots have historically been largely under-specified and under-engineered[1]; they are usually built by researchers who have little experience in designing and building reliable electro-mechanical systems and who, quite reasonably, are more interested in getting their robots to perform immediately needed tasks and to function long enough to run the test demo than in building robots that will provide long-term service to their laboratories. Thus, every robotics researcher has to build their own robot, because every robot available in their laboratory either (1) won't meet cur-

[1]This statement is a generalization and is not intended as a blanket insult. It is also a difficult statement to justify in a scholarly way: If someone's robots work poorly, this fact will typically not be mentioned directly in papers published on work tested on those robots. Our perception is largely formed from direct observations and informal and second hand commentaries; nevertheless, we believe it to accurately reflect the state of the mobile robotics world. Some authors are refreshingly candid: [Par94], for example, says of the IS Robotics robots she used for her research, "The side IR sensors of RED and GREEN (2 of the 3 model R2 robots) have become dysfunctional" (p. 52), "...incessant mechanical failures" (p. 17), and "If the IRs were working, then the gear train would break. If the radio was working, then the gripper motors would break. If the gripper motors worked, then the break beam between them would fail. Perhaps the ... touch sensor between the fingers would break. And, just was you get everything working, the battery supply runs out... The robot loses its program... but the robot's microprocessor refuses to talk to the Macintosh." (p. 205)

rent needs or (2) has broken down and is not documented well enough to enable repair. This is a well-institutionalized bad habit, for a number of reasons:

1. Mobile robot building, for its own sake, is not an encouraged topic for scientific publishing. Mobile robot papers are accepted for publication only if they appear to discuss an interesting scientific principle. This is as it should be, but does make the building of mobile robots a task of secondary interest.

2. Researchers are not ordinarily encouraged to take the months necessary to produce documentation that is sufficiently detailed and correct to allow someone unfamiliar with the mobot to repair, maintain, and in some cases, even to use that mobot.

3. Most university robotics labs cannot afford to hire professional engineers and technicians to design and build their robots; furthermore, few researchers are sufficiently skilled at formal specification and design process to get their needs met by professionals.

4. A researcher designing a mobile robot to test particular theoretical principles often lacks sufficient understanding of those principles and their ramifications to enable them to design a robot with all resources necessary to testing those principles. As they discover new needs, they must meet those needs by retrofits to their original design, making that design more complicated (therefore less reliable) and less crisp and clean (and therefore harder to learn and maintain).

When we began our mobile robotics program, we were aware of these points; we knew furthermore that *we* lacked full specification of our needs. We decided, therefore, to design a system that would be modular enough that we could easily replace individual modules of the mobot systems and add extra modules as required. We knew that certain commercial vendors, notably Real World Interfaces, Nomadix, and (at that time) Denning, manufactured mobile robot bases and were working on complete robotic systems; at the time, however, none of these manufacturers made an affordable system that we were confident would meet our needs. We opted to take advantage of commercial hardware to the extent that such hardware would meet our needs better than what we could build ourselves – the key requirement here was that the commercial hardware meet our flexibility needs.

In many cases, there is an advantage to distributing some of the computation performed by the controllers handling sensors and actuators. In some cases, "point-of-sensing" computation can be used to reduce communication bandwidth. In others, it adds computational bandwidth and resources to the system without requiring a central processor upgrade. Distribution can also increase the modularity of the system by allowing development of subsystems in greater isolation and reducing the complexity of the interaction between subsystems: two microcontrollers connected only by serial line have no resource contention or bus-level interaction between their peripherals. The Three-Beam Laser Rangefinder described in section 4.3.2.2 has its own microcontroller, accruing both of the advantages just mentioned: the data bandwidth of a set of range readings is much lower than that of the video stream from which those range values are generated; in addition, the process of computing those range values effectively exhausts the computational resources of the ranger's processor. In the case of the infrared proximity detector/communicators described in 4.3.2.1, putting the low-level computation associated with proximity detection on separate processors allows the proximity detection procedure to run continually, without significant cost to the rest of the system.

In addition to hardware flexibility and modularity, we wanted our robots to be easy to program. Historically, mobile robots have been programmed in assembly language or a simplified C dialect ([Hor93] is a notable exception). Changing the program running on the robot typically requires recompiling the entire system and either downloading the entire program over a serial line or burning a new EPROM. Little debugging information is usually available, so finding a bug is often difficult. This contrasts with the programming environment available on a workstation running, say, Common LISP, where there is typically a symbolic debugger available that allows for tracing the execution of individual functions, interactive examination of variables, and redefinition of individual functions and constants without need for reloading the entire system. We decided that we wanted a high-level development environment of that sort for programming our robots.

4.1.2 In this Chapter

The purpose of this chapter is to describe in some detail the Cornell Mobile Robots and the design goals that drove their development. We introduce the robots and list their capabilities, then describe the robots' architecture, both hardware and software. Finally, we detail each of the sensors and actuators on the robots. We argue that we met our goals of robustness, flexibility, and ease of use, through a combination of modular design and careful engineering.

Our main goal has been to design and build robots that will be flexible and easy to use. Section 4.2 describes the basic architecture of the Cornell Mobile Robots and why we feel that the architecture makes the mobots flexible and easy to use. We show and describe the topology of the hardware architecture, which is a loosely

Figure 4.1: CAMEL, TOMMY, ROSEMARY, and LILY the Cornell Mobile Robots

coupled set of independent microcontrollers operating sensors and actuators under the control of a central, more powerful processor. We then categorize the processors in the robots by the basic role they play. The microprocessors in our robots perform three main roles: high-level control, interprocessor communication and routing, and low-level, point-of-sensing control. We also talk about how we achieved the physical modularity that is necessary to the degree of flexibility we desire. Finally, section 4.3 discusses each sensor or actuator subsystem that is part of one or more of our robots. We describe the functionality of each subsystem at a high level, and in some cases give a more detailed analysis of the subsystem.

4.1.3 Meet TOMMY, LILY, ROSEMARY, and CAMEL

We have built four mobile robots in our laboratory, TOMMY, LILY, ROSEMARY, and CAMEL. All four of them have 68xxx-based microcomputers that we program in a dialect of SCHEME ([AS85,CR92]) called Mobot SCHEME, which is designed for use on mobile robots. All four are equipped with microcontroller boards that control their sensors and actuators. Each has a modular enclosure equipped with hinged doors and interchangeable mounting plates that allow us to add, remove, and interchange sensors, actuators, and their controlling electronics easily and flexibly. Each of them has an array of 12 Polaroid-style sonar range sensors, 12 infrared proximity-detectors/communicators, a set of pushbutton bumpers, and buttons and lights for user-robot communication. TOMMY has a speech synthesizer that lets him output state verbally, and a mount that allows a video camera to be attached to his head for remote-controlled active vision experiments. LILY has a laser rangefinder mounted on a motorized pan/tilt platform, and a stage equipped with extra batteries, allowing her to perform longer autonomous experiments and bear more electronics. ROSEMARY also has an extra battery stage, plus two individually servoable pan/tilt heads and a DSP/frame-grabber card, enabling her to do on-board computer vision experiments. TOMMY, LILY, and ROSEMARY have Real World Interfaces B12 synchro-drive wheelbases, which have small control and sensing errors. CAMEL has a larger, treaded base, which allows him to move larger objects, but with greater control and sensing errors. In section 4.3 of this chapter, we will discuss each of the subsystems of our robots in more detail. We discuss the Generic Controller and the Three-Beam Laser Rangefinder in sections 4.2.1.2 and 4.3.2.2; in-depth discussions of these subsystems can be found in Appendices A and B.

4.2 The Architecture of the Cornell Mobile Robots

Figure 4.2 is a block diagram of the architecture of a typical one of our robots. This architecture can be described as a loosely coupled network of individual microcontroller systems (our robots contain an average of nine independent CPUs each!) under the direction of a more powerful CPU that runs all user-level programs. That more powerful CPU can communicate with a development/debugging environment that runs on a workstation. The CPU (typically a 68000, but ROSEMARY has a 68340) communicates with a 68HC11-based Polaroid sonar sensor controller over a shared-memory bus connection: this is an off-the-shelf controller that was designed to communicate directly with the CPU. Apart from the sonar, all sensors and actuators communicate with the 68K via an Intel 80C196-based single board computer designed at the Cornell Robotics and Vision Laboratory. We call this computer the Generic Controller (GC). The synchro-drive wheelbase, the laser rangefinder, and the speech synthesizer all have their own microcontrollers, and are all controlled via RS-232 serial lines through the Generic Controller. The bump sensors and the user-programmable buttons and lights are controlled by registers that are wired directly to the Generic Controller via our GC-Bus expansion interface. The infrared communication system consists of 12 receiver/transmitters controlled by four

87C196-based "door controllers", which are interfaced to the GC through a GC-bus expansion board. The laser rangefinder and the infrared door controllers have designs that are similar to that of the Generic Controller: using the same processor architecture and some of the same board-level functionality enabled us to share operating system code and simplified the design of these other controllers. The velcro grabber (a disengageable velcro-covered plate designed for easy manipulation of velcro-covered objects, designed to allow grasping without the requirement for grasp planning) and LILY's pan/tilt head are each controlled by boards that interface to the GC-bus, but that also provide power electronics to drive the motors on these devices. At the present time, each of our robots has a single Generic Controller that oversees all sensors and actuators except for the sonar system. It is, however, possible to split the load among two or more Generic Controllers by hooking the controller cards and the serial lines from the sensors and actuators into multiple GCs and performing a simple software reconfiguration.

4.2.1 The Hardware Architecture

4.2.1.1 The Main Computational Engine

Since we wanted the user-level programs on the robot to be written in a functional language, we needed a main CPU with a fair amount of speed and that could support a large amount of memory; we knew that an 8-bit microprocessor with a 16-bit address space would not be adequate. Because of the availability of gcc and low-level debug monitor software for the 68K family, the availability of small, inexpensive, low-current, expandable, single-board 68000 systems, and the known capacity for upgrading within the 68K family, we elected to use a CMOS 68000 board manufactured by Gespak, Inc. for the main compute for TOMMY, LILY, and CAMEL. This board has built-in timers and serial lines and capacity for 512KB of SRAM and 256KB of EPROM; it also connects to Gespak's G-96 bus, enabling expansion by adding other boards. LILY has an additional 2MB of SRAM on an additional card, to support mapmaking and localization operations. ROSEMARY uses a Gespak 68340 board that supports up to 2MB of RAM and 512KB of EPROM on board, along with the timer and communication electronics built into the 68340 processor. The 68340 is essentially a 68020 with a 16-bit data bus and built-in extras that make it suitable for use as a microcontroller. Both the 68000 and the 68340 boards run a dialect of SCHEME known as Mobot-SCHEME, which is based on the SCHEME-48 virtual machine. Section 4.2.3, below, summarizes how Mobot-SCHEME is used on our robots; [RD92] provides a more detailed description of the system. Low-level

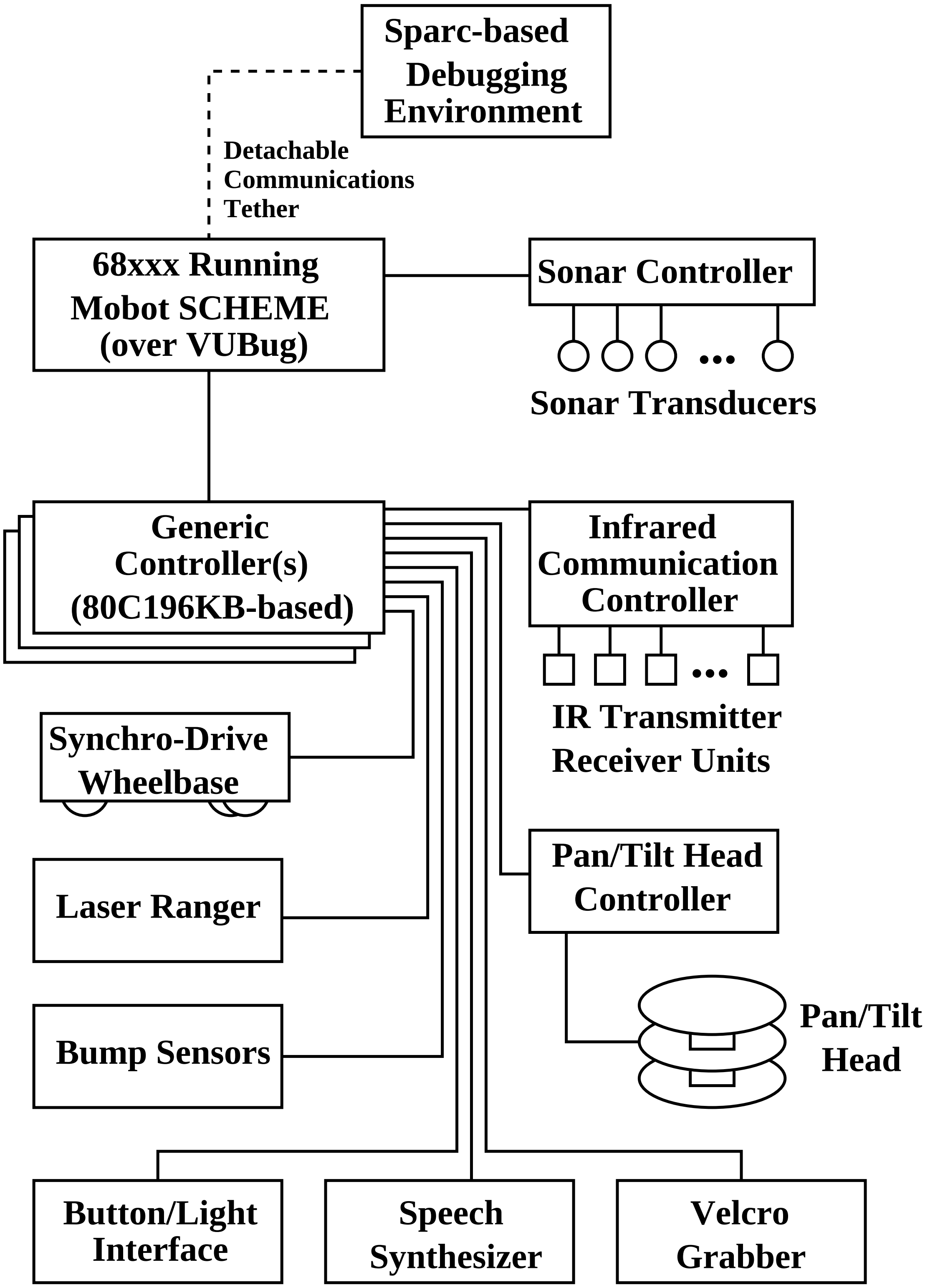


Figure 4.2: A Block Diagram of LILY's Hardware Architecture

I/O and peripheral control is provided by a debug monitor known as VUBug[2]. For the remainder of this chapter, we will refer to this board as the "SCHEME board".

4.2.1.2 Controlling Sensors and Actuators: the Generic Controller

One early design decision that has had great effect on the evolution of our robots was to separate each robot's main computational engine from its low-level control. There were two motivations for this decision. First, we were fairly certain that the SCHEME board would be purchased from an outside vendor. We knew also that we would be interfacing electronics to the bus and microprocessor ports of the controller, leading to the possibility (and eventuality) of components on the controller being destroyed. We decided that it would be best to restrict this destruction to parts of the system that were built in-house, as repairing those would be faster and cheaper than sending purchased boards out for repair. Secondly, by keeping the SCHEME board loosely coupled with the rest of the system, we made it easier to upgrade to a more powerful main computer: if the only connection between the main computer and the rest of the system is a serial connection, then the only work involved in upgrading from, say, a 68000 board to a 68340 board is in porting the software that runs on the 68000 to the 68340. We wanted to eliminate the need for hardware design or construction, to the greatest extent possible, from upgrades to the SCHEME board.

We eventually settled on the design we call the Cornell Generic Controller 3.0 to fill the controller niche on our mobots.[3] Version 3.0 of the Generic Controller (GC) is a single-board microcontroller based on the Intel 80C196KB microcontroller chip. The GC has 32KB of static RAM and 32KB of EPROM. For easily expanded I/O, there is a 256 byte I/O page and provision for up to 8 external interrupts. The microcontroller's data bus and part of its address bus, along with the external interrupt lines, are connected to the GC-Bus, which allows connection of a large number of peripheral devices to the microcontroller by memory-mapping them to the I/O page. Version 3.0 of the GC is a two-layer 160mm by 100mm PC-board that fits a single slot in the mobots' card cages. The GC-Bus is physically a 60-wire ribbon cable that originates at the GC and has multiple connectors, allowing several peripheral cards to be plugged into it. If the peripheral cards needed for a particular robot require more instruction cycles than a single GC can sup-

[2]This monitor was originally developed at Vanderbilt University by Edmund Carter. We extended the monitor and ported it to a new 68000 system while at Vanderbilt, then further extended it and ported it to the Gespak while at Cornell.

[3]Lower-numbered versions of the Generic Controller suffered from insufficient genericity and insufficient reliability engineering. The earliest version was also too large to fit into the robots' enclosures.

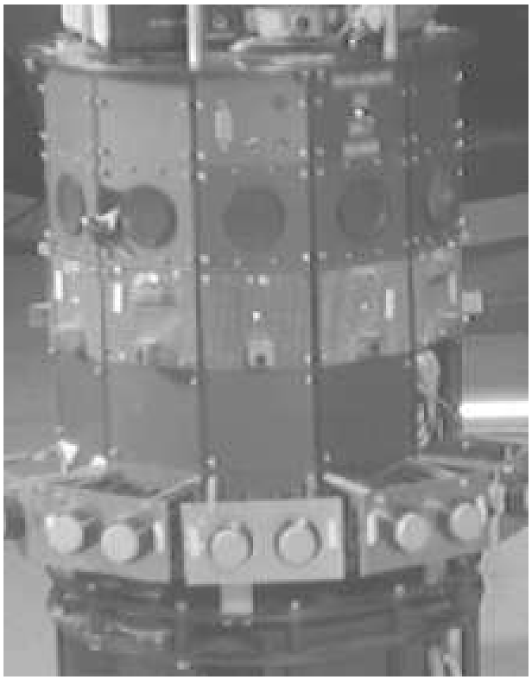


Figure 4.3: The modular enclosure: interchangeable panels and swinging doors make physical reconfiguration of sensors a simple matter.

ply, it is a simple matter to split the peripherals between two or more GCs by adding extra GCs, using multiple GC-Bus cables, and compiling a separate version of the GC's software for each GC, so that each GC runs code for the peripherals it still controls. A detailed description of the program we run on the GCs in our mobots, and of how we make the software modular to match the hardware is given in section 4.2.4. Appendix A gives a detailed description of the GC3, including instructions on how to design new peripheral devices for use with the Generic Controller.

4.2.1.3 Physical Flexibility: The Modular Enclosure

The ability to add sensors and actuators in a flexible, modular fashion requires a physical mount for those sensors and actuators and any electronics associated with them. Our idea was that we should have an outer enclosure that would provide lots of places to mount sensors and actuators on the outside, and an internal cardcage with room for a large number of boards. The enclosure we designed, which was built for us by Real World Interfaces, Inc., is a one-foot high faceted cylinder: The top and bottom plates are one-foot diameter circles, while the sides form a regular 12-gon. There are 4 hinged doors on the enclosure, each of which has three vertical faces facing 30 degrees away from its neighbors. Each face bears five metal plates, each of which is roughly 2 1/2" by 2 1/4". Any plate can be modified or replaced to install a sensor. Any two plates can be interchanged. For example, TOMMY bears a ring of sonar range sensors on 12 of his plates, a ring of infrared communicators on another 12, pushbutton bumpers on four plates, and switches and indicator lights on several others. Inside the enclosure is a cardcage that has room for up to 13 Eurocard-sized boards (100mm by 160mm). The enclosure also contains a DC-DC converter that provides regulated 5-volt power for the contents of the enclosure. The benefits of this sort of enclosure are many: if two robots have different heights, we can mount their infrared communicators in different positions so that the two robots' communicators are at equal height. If a critical sonar sensor on, say, a front panel, malfunctions, we can switch it out with a less critical sensor from a side or back panel, and repair the faulty one at our leisure. In addition, we can add new sensors at any time, without having to perform any machining on the enclosure itself. Finally, the enclosure gives the robots a much tidier appearance by hiding cables and such inside, and protects the robot's electronics by keeping them internal during operation, while making them easily accessible for repair or reconfiguration.

4.2.2 The Software Architecture

The software architecture for the Cornell Mobots is analogous to the hardware architecture; that is, the main SCHEME board, the generic controller, and each of the "smart" sensors has its own software, which is tailored to support our modularity and ease-of-use directives. User code is written in SCHEME and C, and runs exclusively on the SCHEME board. Requests for sensory data and commands to the actuators are primarily routed through the Generic Controller, which is programmed in C and assembly language, in a way that supports easy addition of code to control new sensors and actuators. Each microcontroller-driven sensor or actuator in the system has its own low-level control program, written in C and assembly. Each of these programs supports loosely coupled network communication, allowing the network to be reconfigured with minimal software changes.

4.2.3 Functional Programming for Robots

The development environment we use for writing user-level programs on our robots is one of the main reasons our robots are easy to use. Jonathan Rees has implemented a version of SCHEME called Mobot-SCHEME that allows us to run SCHEME programs on the robot([RD92]). Mobot-SCHEME is based on the SCHEME-48 ([CR92]) multithreaded byte-code interpreter and the byte-code compiler that converts SCHEME text into the byte-codes that the interpreter executes. Our environment consists of three main parts: The byte-code compiler and communication package that runs on a Unix workstation, the byte-code interpreter that runs on the robot, and the built-in library of mobot control functions. The interpreter is the basic SCHEME-48 virtual machine, configured so that its primary read-evaluate-print loop receives and transmits streams of byte-codes through the robot's main serial port. The virtual machine is multithreaded, with a simple fixed-quantum thread-scheduling package that allows multiple threads to execute asynchronously. The workstation-resident part of the system compiles SCHEME text into byte-codes and converts byte-codes coming from the mobot back into human-readable text. The workstation retains all of the symbol table information that was used to compile programs and top-level forms into byte-codes. This allows it to decompile the mobot's output into readable SCHEME text, complete with the variable and function names that were originally given. This combination gives a user the full power of a SCHEME interpreter, but at the same time keeps the size of the programs actually loaded onto the robot small enough to run on a small single-board microcomputer. In addition, it is possible to redefine functions "on the fly"; this differs from the typical robot-programming process, which requires downloading an entire new program whenever a change is made or, in some cases, reprogramming the robot's EPROMs (necessitating turning the robot off and opening it up). The flexibility of this environment makes it possible for users to begin programming the robot with very little startup cost.

Another factor supporting quick startup for new users, as well as for experienced users developing new mobot algorithms, is the expandable mobot function library, `mobot.scm`. `mobot.scm` provides SCHEME function calls to perform all base-level sensory and control actions supported by the mobot. It also provides facilities for adding new actions, and extensive support for user reconfiguration. Communication with the Generic Controller and subsystems controlled by it, for example, are handled by low-level functions `cgc-call` and `handle-asynchronous-message`. `cgc-call` accepts as arguments a header character specifying the destination for a message, a command string, and a list of arguments to be transmitted with the command string. It composes a string from this information and transmits that message to the GC; it optionally waits for a response before returning. `handle-asynchronous-message` is a separate thread that watches for messages from the GC and takes action on them based on the header characters they bear. A look-up table specifies which individual handler function should be invoked when a message bearing a given header character is received. The function `set-message-handler!`

```
(define (wb-signed-command forward backward conversion)
  (lambda (amount)
    (let ((arg (quotient (* (car conversion) amount)
                         (cdr conversion))))
      (with-lock wb-lock
        (lambda ()
          (if (< arg 0)
              (cgc-call-no-reply dest/base backward (- 0 arg))
              (cgc-call-no-reply dest/base forward arg)))))))

(define (wb-query-string s)
    (let* ((result (cgc-call-string dest/base s))
           (l (string-length result)))
      (substring result 3 l)))

(define (wb-command string)
  (lambda ()
    (with-lock wb-lock
      (lambda ()
        (cgc-call-no-reply dest/base string)))))

; Motion
(define translate-power (wb-signed-command "T) " "T( " (cons 1 1)));pw,0-255
(define translate-relative
  (wb-signed-command "T> " "T< " translation-scale-factor))
(define rotate-power (wb-signed-command "R( " "R) " (cons 1 1))) ;pw, 0-255
(define rotate-relative
  (wb-signed-command "R> " "R< " rotation-scale-factor))
(define translate-limp  (wb-command "TL"))
(define translate-halt  (wb-command "TH"))
(define rotate-limp  (wb-command "RL"))
(define rotate-halt  (wb-command "RH"))

; Queries
(define (battery-voltage)   (wb-query-integer "BV")) ;in units of 1/10 volt
(define (battery-current)   (wb-query-integer "BC")) ;in units of 1/10 amp
(define (rotate-current)    (wb-query-integer "RC")) ;in units of 1/10 amp
(define (translate-current) (wb-query-integer "TC")) ;in units of 1/10 amp
(define (translate-where)   (wb-scaled-query "TW" translation-scale-factor))
(define (rotate-where)
  (wb-scaled-query "RW" rotate-where-scale-factor))

; Set parameters
(define set-translate-speed!            ;mm/sec
  (wb-command-with-parameter "TV " translation-scale-factor))
(define set-translate-accel!            ;mm/sec^2
  (wb-command-with-parameter "TA " translation-scale-factor))
```

Figure 4.4: An excerpt from `mobot.scm`

takes a header character and a thunk[4] to be invoked when that header character is seen. This allows the user to specify what should happen when a given subsystem sends a message to SCHEME; this is the main mechanism for specifying the actions associated with a new subsystem (or one like LILY's pan/tilt head, which exists only on LILY, and is therefore not "basic"). Other `mobot.scm` functions allow the user to program actions in intuitive ways: to light a particular LED, for example, the user says, "`(illuminate` $< n >$`)`". Another table-based lookup allows the user to specify actions based on, for example, button presses. A user might specify, for example,

```
(set-button-action!  1 '(lambda () (rotate 'by 360))),
```

which configures the system so that when the user presses button 1, the robot turns around. [RD92] contains more information about how Mobot-SCHEME works. [Jen95] offers an in-depth description of `mobot.scm` and the control system that runs on our mobots.

4.2.4 SCMODS: Modular Sensor Control Software

The Generic Controllers in our robots have to interface with the SCHEME board and with an arbitrary number of sensors and actuators. In many cases, the GC is merely a message router: it takes messages from the SCHEME board, determines where they need to go, and sends them out the appropriate serial port. It receives messages from its serial ports, annotates them with their source, and sends them back to the SCHEME board. In other cases, the GC controls the sensor or actuator directly. In these cases, the GC parses commands from the SCHEME board and sends commands to the appropriate subsystem. It also determines when the requested action is complete, and composes an appropriate response, which it transmits to the SCHEME board. The program that has evolved to perform this duty is called SCMODS, for (S)ensor (C)ommand (MOD)ular (S)oftware. SCMODS initializes all of the modules, then commences a servo loop within which it (a) checks for incoming messages from the SCHEME board and initiates actions called for in those messages, and (b) polls[5] its various subsystems for data and messages headed for the SCHEME board, packaging those messages appropriately and transmitting them to the SCHEME board.

[4] *Thunk* is the term for a SCHEME function taking zero arguments.

[5] At the lower levels, the GC is handling serial I/O and similar functions in an interrupt-driven fashion. When we refer to SCMODS "polling" a subsystem, we mean looking to see if, for example, a complete message packet has been received from that subsystem.

A new SCMODS module consists of three main parts: an initialization routine, a command/outgoing-message handler routine, and an asynchronous return-message handler. When SCMODS is invoked (at cold-boot time or after a reset), it initializes its own state, then invokes the initialization routine for each module; these routines typically initialize variables pertinent to the module, including I/O page registers if the module services a GC-Bus peripheral, and send any necessary messages to the sensor or actuator involved. SCMODS then starts an endless servo loop. Each time through the loop, it looks to see if it has any messages from the SCHEME board. If there are any messages, it looks at the header on the message and determines which module the message is intended for. It invokes the outgoing message handler for that module, which may simply strip off the header and send the remainder of the message to the module, or may invoke a routine to parse that message and issue commands to a GC-Bus peripheral. Finally, the servo loop looks at each peripheral to determine whether it needs to be serviced. If the peripheral is attached to the GC by a serial line, this means checking that line's input buffer; if the peripheral is on the GC-Bus, it means checking its status registers. In the serial case, the incoming message is received and a header is added to the message, specifying where it came from. The message is then routed to the SCHEME board. In the bus case, a string is built up which contains a header and the appropriate information from the module. That string is then routed to the SCHEME board. A new module can be added to the system simply by creating the necessary subroutines and invoking them within in the SCMODS main routines. If a single generic controller becomes unable to handle the load of servicing all of its modules in a timely fashion, its load can be split across two generic controllers (figure 4.5) by moving some of the hardware to a new GC, programming each with a SCMODS for the modules it now controls, and chaining them so that the first GC passes commands aimed at one of the second GC's modules along to that GC over a serial line. Return messages from the second GC are piped straight through the first GC to the SCHEME board. Excerpts from the SCMODS running on LILY can be found in Appendix A.

4.3 The Capabilities of the Cornell Mobile Robots

4.3.1 Mobot Components from Outside Vendors

4.3.1.1 The RWI B12 Wheelbase

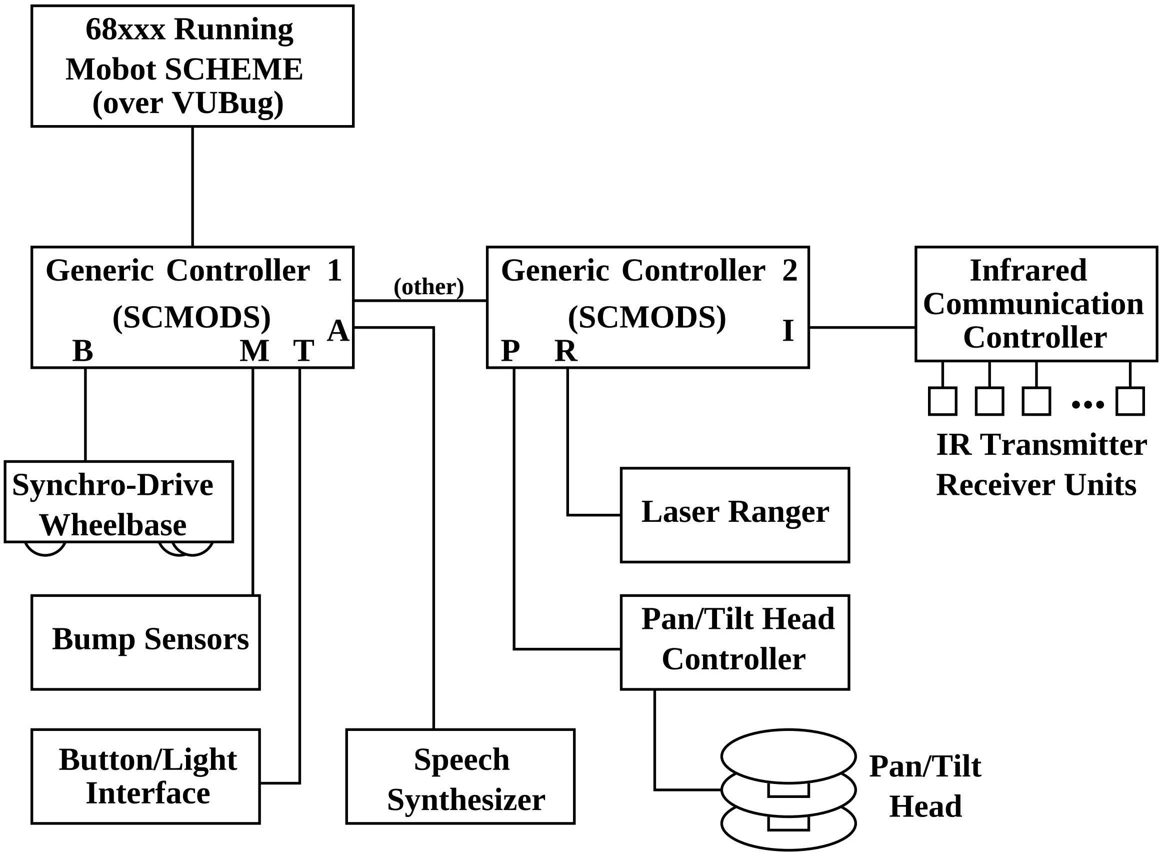


Figure 4.5: Two or more Generic Controllers can be daisy-chained to control large numbers of peripheral devices.

TOMMY, LILY, and ROSEMARY each have a synchro-drive wheelbase manufactured by Real World Interfaces, Inc. The B12 is a one-foot diameter, 7-inch high cylindrical base with a 30 kilogram payload capacity. A synchro-drive wheelbase has wheels that are all driven together. One motor drives a belt that causes all wheels (in our case, three of them) to rotate together for translational motion. A second motor drives a belt that causes all wheels to turn about a vertical axis for rotational motion. The base has its own computer, which controls motion and monitors power consumption. It provides position and velocity control for both axes with a control error of roughly one centimeter and three degrees. Motor shaft encoders provide sensing of translation and rotation with roughly millimeter- and subdegree-accuracy. In addition, at the request of the Cornell CS Robotics and Vision Lab, RWI has incorporated torque control into their software. This allows the robot to make "safe" motions, in which the robot will apply no more than a specified torque to its motors, and therefore will apply no more than a specified force to any object with which

Figure 4.6: The Real World Interfaces B12 Synchro-Drive Wheelbase

it might collide. This is necessary for the safety and integrity of the robot, of any objects in the robots workspace, and of any humans in the workspace.

4.3.1.2 Polaroid Sonar Range Sensors, as Implemented by RWI

We also obtained from RWI a set of twelve wide-angle sonar range sensors, supported by RWI's driver and controller electronics. These are the sonar transducers designed by Polaroid for their Land Cameras; they measure distance by emitting a high frequency auditory chirp, waiting until they receive an echo of that chirp, and measuring the time between emission and reception.[6] The distance to the nearest measurable object is a product of half the travel time and the speed of sound in air. The typical effective range for these sensors is roughly 12cm to a few meters. Each transducer has its own, individually addressed driver card mounted behind it. The drivers are connected to a 68HC11-based controller card via two ribbon-cable chains, enabling one transducer on each chain to be in use at a given time.

[6] These are a very popular range sensor for mobile robots, primarily because they are relatively inexpensive and small enough that several can be mounted on a small mobile robot. Because so many robotics researchers have used them, they have received wide attention in the mobile robotics literature([Fly93,Mat90]). See [Gro87] for detailed technical information on the transducers and drivers. [Kuc90] uses the same transducers with a modified driver that makes use of information from the return signal other than time-of-flight to obtain more accurate and precise data.

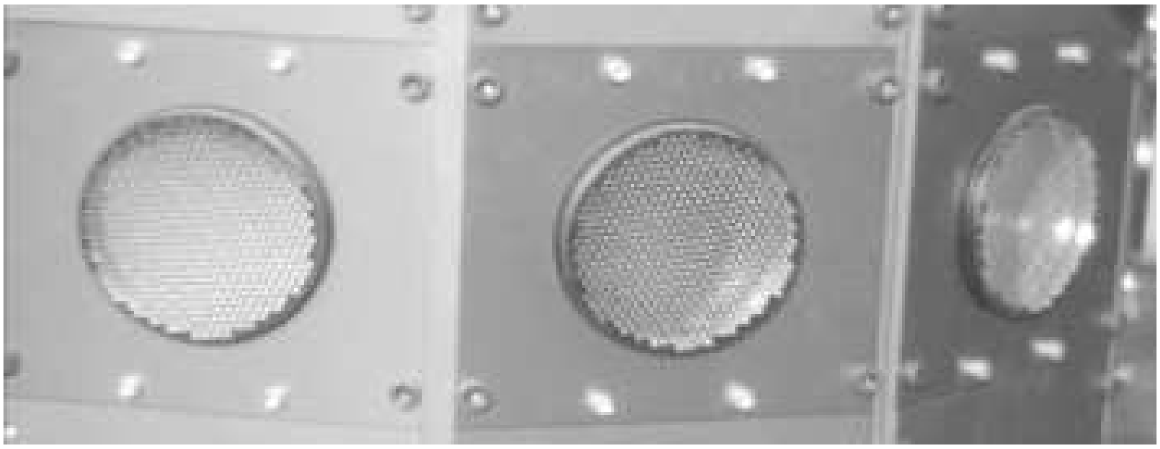


Figure 4.7: RWI Polaroid Sonar Range Sensors

The sonar controller is specifically designed to interface with a G96-bus controller card, so SCHEME interacts with the sonar controller directly over the bus, rather than through SCMODS. For detailed mapbuilding and localization purposes, these sensors are not very accurate. They have several drawbacks: the transducer has a sensitive region that is approximately a $25 - 30°$ cone, making it difficult to detect small objects or open channels, and difficult to determine precisely in which direction an object lies relative to the robot. There is also a tendency for the chirp to reflect specularly off of an oblique surface and never return to the transducer. We use them mostly either in restricted modes in which we have found them to be reliable, or in modes where high accuracy is not required. Used in this fashion (see [JR93] for an example), the sensors are useful enough to justify their inclusion on the robot.

4.3.1.3 The *Accent* Speech Synthesizer

One challenge of performing experiments on a mobile robot is telling why the robot is doing what it is doing, particularly if it is not behaving as desired. To make it easier for the user to provide "output debugging", we installed on TOMMY a speech synthesizer. We used an AICOM *Accent SA* model speech synthesizer. This model accepts text (English or phonic) over its RS232 port and reads it over its built-in speaker.

4.3.1.4 On-Board Computer Vision

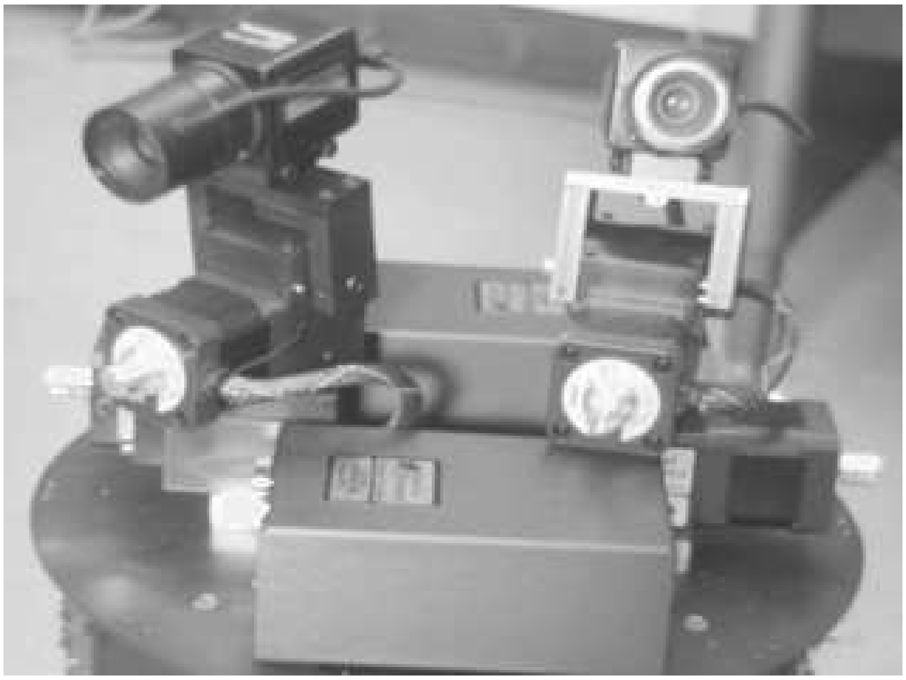


Figure 4.8: ROSEMARY's pan/tilt heads

Our newest robot, ROSEMARY, is equipped with hardware for on-board computer vision. Specifically, she has two cameras that are mounted on individual pan/tilt platforms and connected to an on-board DSP card, which provides sufficient computational power for limited general purpose vision. The two cameras are monochrome CCD cameras equipped with electronic shutters (to eliminate motion blur as the robot and the cameras move). The pan/tilt units were manufactured by Directed Perception, Inc., and are model PTU-C units. They have a pan range of 300 degrees at up to 300 degrees per second and a tilt range of 120 degrees at similar speeds. They have a positioning precision of roughly 3 arc-minutes. We selected these units for their small size (since we wanted to put two of them on top of ROSEMARY) and low power consumption. The DSP card is a Smart-Eye 1, built by Visionex, Inc., around a Texas Instruments TMS320 DSP microprocessor. The card provides for up to four video inputs fed into two frame grabbers; programs running on the DSP can access either frame and write results into the processor's main memory. We selected this card for these features, plus its small size (it is small enough to mount inside the modular enclosure) and low power consumption. As we are only now finishing integrating this part of ROSEMARY, it is not yet clear exactly how we will use and interact with ROSEMARY's vision system, but potential applications will be similar to the visually guided navigation work described in [HLR94a,HLR94b].

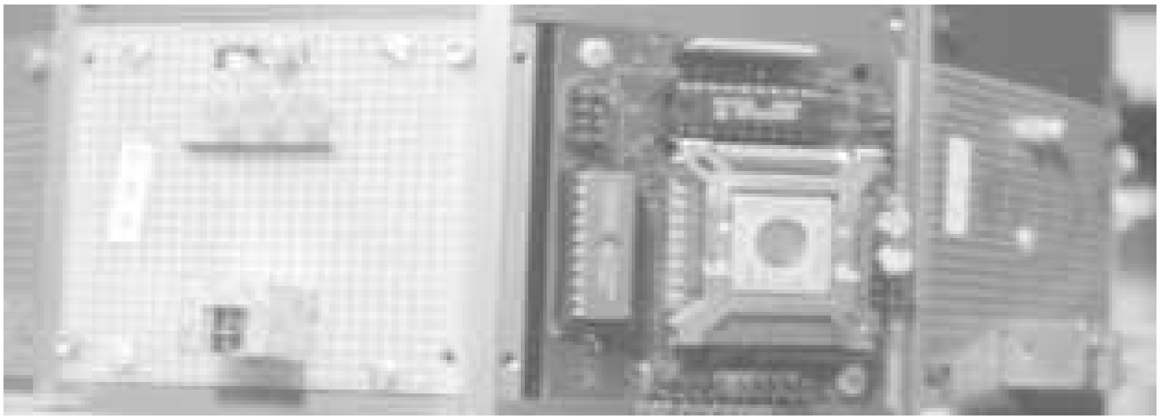


Figure 4.9: The Infrared Door Controller and transmitter/receiver units

4.3.2 Modules designed in the Cornell CS Robotics and Vision Laboratory

4.3.2.1 Infrared Proximity Detection and Serial Communication

Sonar sensors and the laser rangefinder on our robots have a minimum range of several centimeters. Contact sensors have a maximum range of zero centimeters. For this reason, we wanted some sort of proximity sensor, which would indicate to the robot that "there is something close to the robot". Mobot builders before us had used modulated-infrared receiver-transmitter pairs to serve this purpose: If the IR receiver is getting back a signal that looks like the one we're transmitting, then something is reflecting the light back in our direction, and it must be within about half a meter. We initially implemented the "standard" IR transmitter and receiver, using an IR LED and an LM555 timer circuit, and the standard reciever, a Sharp receiver/demodulator hooked to an LM567 tone-decoder. We implemented an array of these circuits for each of our robots. During this process, it occurred to us that these receivers and transmitters could be used for serial communication between robots: If a serial bitstream is transmitted over the transmitting circuit, a receiving circuit on another robot (or workstation) can receive that bitstream and pass it to a serial port on that robot (workstation). As we experimented with that initial implementation and explored the possibility of wireless line-of-sight communication between robots, it became clear that, to enable IR communication, we would have to redesign the IR system completely:

- The timer circuit and the tone decoder circuit both required adjustable RC networks to set the frequencies. These networks tended to drift, requiring frequent recalibration.

- To eliminate false positives in proximity detection, a two-frequency modulation scheme was used: the transmitted signal was a 1200Hz signal modulated at 40KHz. The Sharp module removed the 40KHz carrier and passed the received signal to the tone decoder that looked for a 1200Hz signal. The tone decoder required up to 30 periods of the desired frequency to recognize the presence of that frequency in its input. That meant a maximum transmission rate of about 40 bits per second.

- When the lighting in our lab was converted to high-efficiency fluorescent, the Sharp receiver modules proved extremely sensitive to the high-frequency modulation used to make those lights energy-efficient. The IR component of that lighting was powerful enough to flood out the robot's transmitted signals.

We addressed the first two of these difficulties by replacing the 555 and the 567 with an 87C196KB microcontroller and a small PLD and changing the proximity detection algorithm. We were using "Transmit a signal of frequency ω and look for a signal of frequency ω at the receiver." The new algorithm was "Transmit a binary signal at frequency ω and compare the received signal to the one transmitted." This simplified the proximity detection task to "are the two signals the same $r\%$ of the time, for some empirically determined r. For that task, the 87C196KB has enough internal resources to control three receiver-transmitter pairs. For serial communication, we modulate the bitstream at the carrier frequency (40KHz), and pass the output of the detector module directly to a UART. This allows us to transmit information at the 1200Hz frequency that is the maximum frequency allowed by the detector module. Preliminary experimental results indicate that we can perform directional, line-of-sight serial communication at distances of up to 4 meters at 1200 baud using this system.

We are in the process of addressing the fluorescent-lighting problem by converting our IR systems to a newer generation of IR transmitter and receiver modules, also manufactured by Sharp. The new modules are similar to the old ones, except that they use a carrier frequency of 500KHz, and are rated by the manufacturer as being capable of carrying 19200 baud serial transmissions at distances of up to 3 meters.

4.3.2.2 The Three-Beam Laser Rangefinder

Because we desired a faster, more accurate range sensor than the wide angle sonar, with a smaller area of sensitivity, we designed a simple laser rangefinder. We call our design the Three-Beam Laser Rangefinder. This ranger projects three parallel, coplanar laser diode beams into the world and uses a monochrome video camera to sense where the beams intersect objects in the world. The video signal feeds into a partial frame-grabber, an 87C196KB-based board that digitizes an up to 64-line-wide slice of the video camera's output field. The frame grabber computes the positions of the three brightest spots in the digitized slice. Based on the positions of these spots in the image it outputs a pair of distance values (one based on the center and left beams, and one based on the center and right beams). The relationship between inter-spot distances and range values is determined by perspective. This system differs from typical optical triangulation systems [BHM85,Rio84] in that the camera and the laser diodes are aligned to point in the same direction and are approximately colocated. The center beam is not strictly necessary for range computation, but serves the purpose of significantly reducing the calibration complexity of the system. We compute two range values at once to permit computation of approximate surface normal information. We designed the ranger as a free standing system with serial I/O because we intended to mount it on a pan/tilt platform, and didn't want to transmit video or bus signals through the platform's slip ring. The ranger returns 20 range pairs per second, with an effective range of 30cm to roughly 3 meters. A more detailed description of the three-beam ranger is found in Appendix B.

4.3.2.3 An Infinite-Pan, Finite Tilt Servoing Platform

Lily, the robot with the three-beam rangefinder, also has a servoed pan/tilt head. This head was designed to point the ranger, but can be used for any light (up to two pounds) load with a horizontal diameter of less than a foot (the diameter of the robot). Mechanically, the head is a tilt stage mounted on top of a pan stage. The pan stage is a one-foot diameter, two-inch high cylinder with a rotating top plate. The pan stage includes an electrical slip-ring, so that power and signals can be transmitted through the stage while allowing infinite rotation. The tilt stage has the same form factor, except that the plates are narrowed to six inches to allow for greater tilt range. With the ranger installed, the top plate of the tilt stage has a range of $\pm 26°$; a smaller payload can be tilted to $\pm 45°$. The pan stage can be positioned with sub-degree accuracy, and has a speed range of 0 to 180 degrees per

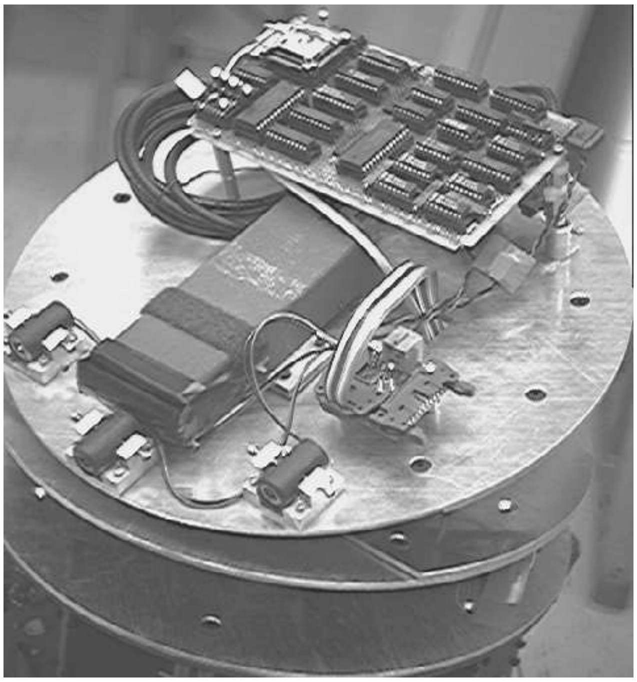


Figure 4.10: The Three-Beam Laser Rangefinder, mounted on LILY's pan/tilt head

second. The tilt stage can be positioned likewise, at speeds of 0 to 8 degrees per second. The pan/tilt head is driven by a controller card that lives on the GC-Bus and incorporates two LM629 motor control ICs, providing position and velocity control with configurable position, velocity, and acceleration limits and control parameters. Also present on the controller card are two LM18293 power H-bridge drivers that interface the pulse-width-modulated output of the motor controllers to the motors. The motor controllers reside within the Generic Controller's I/O space; commands to the pan/tilt head can be issued from SCHEME, and are converted to motor controller commands by a SCMODS subroutine.

4.3.2.4 Enabling User Interaction with a Tetherless Robot: Buttons and Lights

We needed a way for the user to interact with the robot when it is operating autonomously and without its communications tether. The speech synthesizer can be used for output from the robot, but we also wanted the ability to output continuously, for example to indicate what mode the robot is in, and to give commands to the robot. For this reason, we've installed a set of 8 LEDs and 8 pushbuttons on each robot. The lights can be controlled from SCHEME. In addition, our sys-

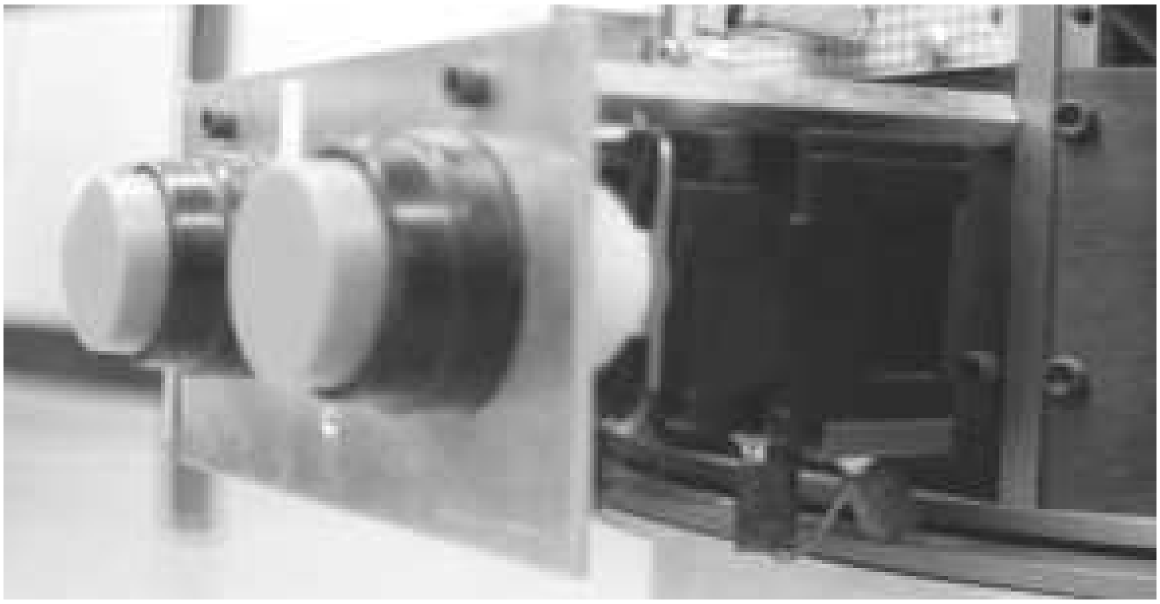


Figure 4.11: Part of TOMMY's bumper array

tem is set up so that a button can be bound to execute an arbitrary zero-argument SCHEME function.

4.3.2.5 Detecting Collisions and Object Boundaries using Bumpers

Bumpers are a necessity for any mobile robot. A contact sensor is the most reliable way to detect the presence of an obstacle in the robot's path. Because there is work going on in our lab involving manipulation of large-scale objects with mobile robots, we have additional requirements on our bumpers: They have to be physically robust enough to withstand a lot of hard pushing, and they have to provide more information than just "contact" and "no contact". We initially experimented with a metal bumper-ring instrumented with strain gages, hoping to satisfy the robustness requirement while providing the robot with relatively precise information about the location and magnitude of forces against the bumper, as well as side-torques against it. Our initial efforts failed to produce a satisfactory design and confirmed a suspicion that the calibration complexity of a strain gage design was unacceptably high for the benefits produced. We settled instead for a partial ring of heavy pushbuttons. We used eight large industrial-quality pushbuttons mounted on aluminum plates, two on each of the four front faces of the modular enclosure. The state of the eight buttons exists as a byte register in the Generic Controller's I/O space, and can be read by SCHEME. Jennings and Rus analyzed this bumper

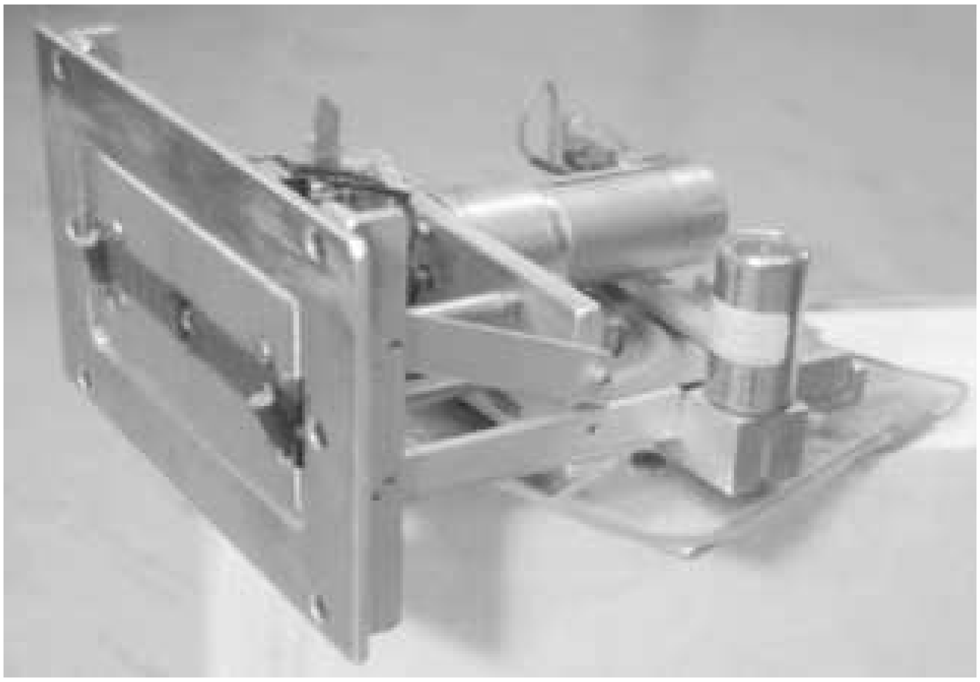


Figure 4.12: The Velcro Grabber

configuration in [JR93], and found that the relative orientation of a planar face in contact with the bumper can be measured with a $\pm 3.5°$ accuracy. They continue to present algorithms for linear pushing and reorientation of polygonal objects using single and multiple robots equipped with this bumper.

4.3.2.6 Grabbing without Planning a Grasp: The Velcro Grabber

When a robotic manipulator arm is used to acquire an object, it can form a grasp by putting fingers on multiple sides of the object and applying sufficient force to prevent the object from slipping away. It can't, however, do this in a stable fashion without some sort of planning that accounts for the shapes and frictional properties of the object and the manipulator. An armless mobile robot, however, can't grasp a large object in the same way; in addition, since it typically has imperfect knowledge of the object's geometry, it cannot usually *plan* a grasp. To enable our robots to acquire objects, we needed some sort of manipulator. We wanted one that would not require grasp planning. The solution we selected was to designate "acquirable objects" as being those that have been prepared so the robot can grasp them. Specifically, acquirable objects are those covered with fuzzy-Velcro. We designed and built a grabbing manipulator equipped with a moveable prickly-Velcro plate. The plate is mounted on a worm-screw shaft so that it can be pulled away from a fixed plate, as

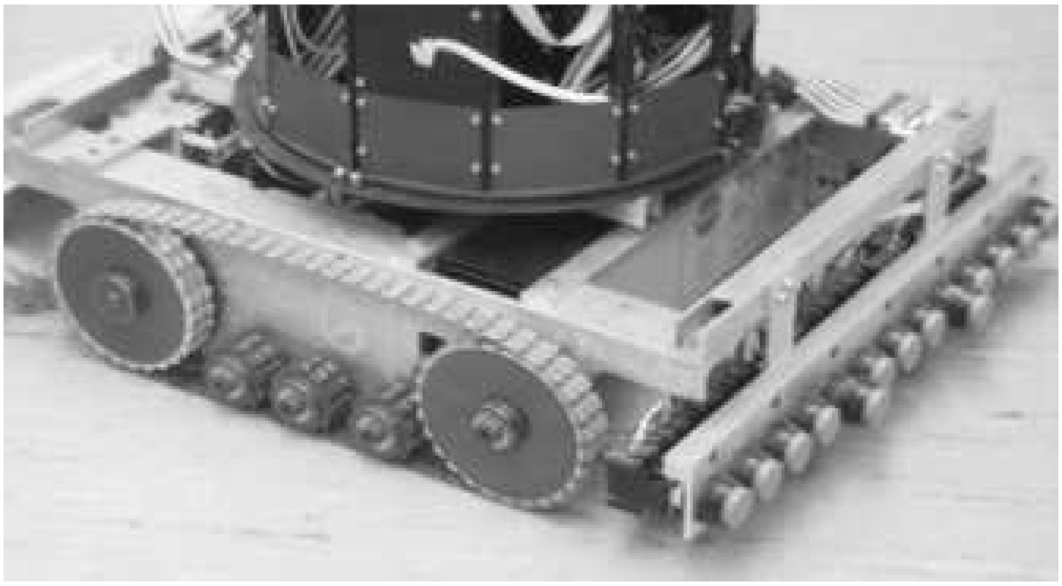


Figure 4.13: CAMEL's treaded base

the difficult operation with Velcro is not acquiring the object, but rather letting go of it. The Velcro plate sits within a fixed frame, so that the object is held in place, while the Velcro plate is pulled away from it. The grabber can rotate relative to the robot, or it can be held rigid relative to the robot. This enables the Velcro plate to move compliantly to obtain a solid grasp on an object, but enables the robot to fix the grabber so that it can freely manipulate the acquired object. We have performed only limited experiments with the Velcro grabber to this point, but anticipate that it will prove useful for operations involving, for example, objects that must be pulled away from a wall.

4.3.2.7 Moving Larger Objects: The Treaded Mobile Robot Base

The robot CAMEL is the only one of our robots with a locally designed and constructed wheelbase. CAMEL was originally designed with large-scale manipulation in mind. For this reason, it is a larger base than the B12 and has stronger motors. For extra stability and traction, CAMEL was designed with treads, akin to those of a large bulldozer or tank. For extra pushing stability, CAMEL has a flat bumper in both the front and the rear. The drawbacks to CAMEL's design are that it is less agile than the synchro-drive bases and that its odometry is much worse. This latter is because a treaded base must have a lot of slip between its treads and the floor. This is necessary, since a base with two perfectly rough treads cannot rotate. Clearly, CAMEL cannot navigate effectively using odometry and self-made maps.

It would be possible to navigate using the localization algorithms from chapter 2, if CAMEL were provided with a map made by another robot, either a priori or by communication from another robot.

4.4 Conclusion

We wanted to design mobile robots that would permit us to perform experiments on a variety of mobile robotic topics. We didn't want to specify what those topics were in advance, though, so we wanted robots that we could change easily. Since we knew that many people would be using these robots, we also wanted robots that would work reliably over an extended period of time. Our design goal, therefore, was to build robots that would be robust, flexible, and easy to use. We realized that goal with our modular robots. Our robots have physical modularity, in that it is straightforward to interchange, add, remove, and move components. They also have software modularity, making it easy to add new sensors and actuators. The fact that each module is a separate entity makes them easier to design, build, and maintain. The open-ended I/O structure of the robots' microcontrollers makes it possible to add many different kinds of sensors and actuators, usually without affecting the performance of other subsystems. In this chapter, we discussed our design goal and the mobots we built in our effort to achieve that goal. A more detailed description of a some parts of the system, specifically the Generic Controller and the Three-Beam Laser Rangefinder, can be found in Appendices A and B.

We believe that the mobots we have built achieve our design goals of robustness, flexibility, and ease of use. The evidence supporting this claim is seen in the number of people who have successfully used our mobots for experiments pertaining to their own work.

- **Robustness**: this claim is supported by the large amount of time that our robots are usable and in use: Over the last two and a half years, TOMMY and LILY have been in an operational state an estimated 85 percent of the time. During that two and a half years, the robots have logged approximately 20 robot-hours per week on average, with peak usage as high as 50 robot-hours per week. Most of the malfunctions the robots have suffered have been minor (the wire to a bumper button broke, or the battery-charger connector became inoperative) and easily repaired.

- **Flexibility**: Our robots have played roles in many different sorts of robotics and computer vision research. Among these have been: mobile robot localization ([BCD93] and this thesis), cooperative mobile robot manipulation ([DJR94a,DJR93] and this thesis), information invariants for mobile robotics ([DJR94b]), visually guided navigation ([HLR94b,HLR94a]), and automatic sensor-configuration ([Bri95,BD94]).

- **Ease of Use**: The Mobot-SCHEME development environment and SCHEME-level interfaces to sensors and actuators make it possible for any student who knows how to program to begin working with one of our robots with less than an hour of additional training. Several Cornell CS undergraduates have participated in robotics and computer vision research (cf. Huttenlocher, Leventon, and Rucklidge, [HLR94b]); these students are typically producing useful results within a few weeks of beginning to use the robots. In addition, Ph.D.-level robotics researchers at Cornell have been able to walk into our lab with fully-developed algorithms and implement a working demonstration of their algorithms in just a few days (cf. Briggs, [Bri95,BD94]).

Including all faculty, graduate and undergraduate researchers, and robotics class students, more than sixty people have programmed tasks on TOMMY, LILY, CAMEL, and ROSEMARY. Some of these tasks have been described in [Jen95,JR93,DJR94a, BCD93,HLR94a,HLR94b,Bri95,BD94].

Chapter 5

The Other Sofa Mover's Problem: Cooperating Mobile Robot Manipulation

5.1 Introduction

We define *large-scale manipulation* as the manipulation of objects whose physical/dynamical properties (primarily size and mass) are of roughly the same order of magnitude as those of the manipulating robot. We also define the concept of *non-prehensile manipulation* as manipulation without grasping or gripping – using the body of the robot to manipulate objects. We are interested in large-scale manipulation for several reasons. Traditionally, the "Sofa Mover's Problem" or the "Piano Mover's Problem" is the findpath problem: given an environment, an object, a start configuration and a goal configuration, plan a path for the object from the start to the goal which does not intersect any obstacles in the environment (see, for example, [Lat92]). We are interested in the "other problem", which is that of enabling the robotic "movers" to manipulate the object along such a path. Manipulation plays a role in most potential mobot applications. A robot with no ability to manipulate objects is limited to information-gathering/data-collection tasks, but a robot that can move objects from one place to another is capable of performing a much wider variety of useful work (transport/delivery and cleanup tasks, for example). In addition, we see the opportunity for great research advances in this area – roboticists are only beginning to determine what the interesting questions are and to try to solve them (there has been much good robotics research done for multiple manipulators, yet little pertaining directly to mobile platforms). Finally, we feel that it is easier

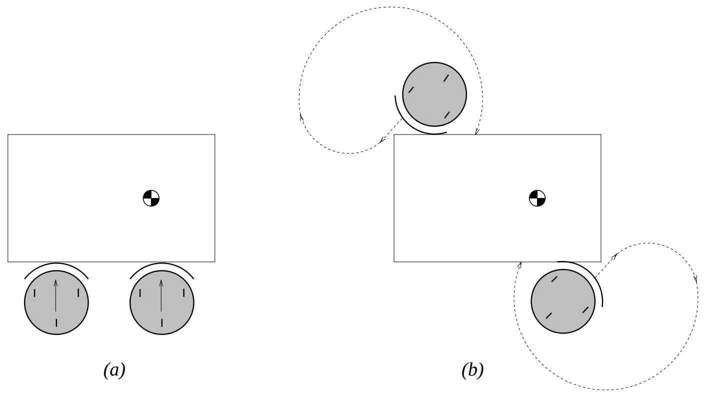


Figure 5.1: [DJR94a,DJR93] present algorithms for multiple robot *(a)* translation and *(b)* reorientation.

to measure success for manipulation tasks than it is for navigation tasks. A random walk algorithm, for example, is as good as any other path-planning system, if we measure only whether or not a mobot can reach its goal position using that algorithm. The probability of a random-walk manipulation strategy achieving a given (reachable) goal is, in general, strictly less than one: it is easy for the object being manipulated to achieve an unrecoverable error state (pushed into a corner, under a desk, etc). Thus, since success is harder to achieve by chance, we can more easily accept that a high rate of success indicates a good algorithm.

Here are some more definitions that will make it easier to compare and discuss multi-robot manipulation protocols: *robot manipulation tasks* use forces to rearrange the physical world in which the robot exists. *Parallel manipulation protocols* are those in which two or more robots simultaneously apply forces to the same coupled dynamical system. *Distributed manipulation protocols* are parallel manipulation protocols that perform tasks in a fashion that is computationally distributed (i. e., the computation is distributed among the robots in a way that qualifies as distributed computation). Finally, we define *Strongly Cooperative* manipulation tasks as tasks that can only be performed by parallel manipulation protocols. We are specifically interested in using distributed manipulation protocols to solve strongly cooperative manipulation tasks. For example, in [DJR94a,DJR93] (figure 5.1), the protocols used are distributed manipulation protocols. The tasks themselves are strongly cooperative: one robot is not strong enough to translate the object in the translation case, and in the rotation case, simultaneous application of forces is required to reorient the box as much in-place as possible. With sufficient communication, these tasks could be performed using non-distributed parallel manipulation protocols;[1] by distributing the protocol both kinematically and computationally, [DJR94a,DJR93] eliminate the need for inter-robot communication.

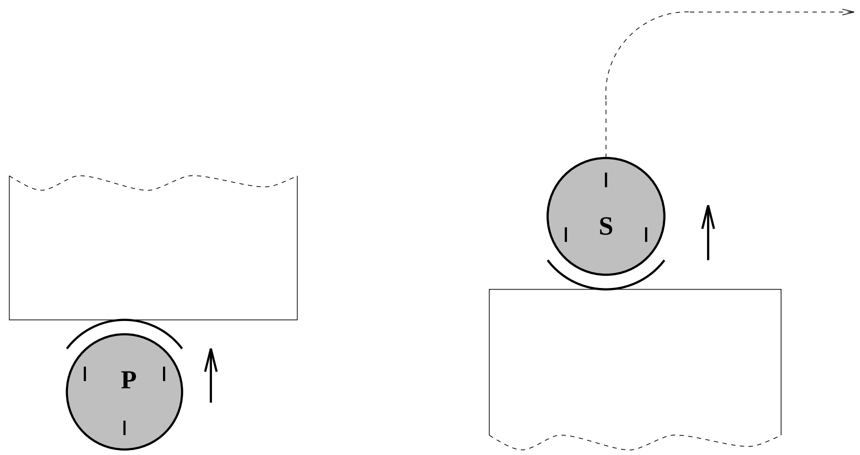
The cooperative mobot manipulation systems we discussed in chapter 1 almost all have strong limitations that we find unsatisfactory. Of the cooperative mobot manipulation works we've seen, all but [DJR94a] fail to analyze their manipulation protocols either for physical characteristics of the system (i. e., mechanics analysis), or for algorithmic robustness. Parker ([Par94]) and Noreils ([Nor93]) discuss cooperative manipulation tasks, but these works discuss protocols that seem to be serializeable – the tasks that they perform with two robots could be performed with one robot with little performance penalty; adding robots appears in their systems to simplify the navigational requirements on the robots, rather than the manipulation requirements (It is always the case in these systems that only one robot is moving at any given time, and that the cooperation is not strong. The additional robot serves to eliminate repositionings of the robots relative to the manipulated objects, which is why we say that the extra robots reduce the navigational requirements in these systems. The manipulation occurring in these systems, though, is equivalent to single-robot manipulation (in [Nor93], the manipulation *is* single-robot manipulation; in [Par94], an example is shown of a single robot executing the two-robot protocol after the "failure" of one of the robots)). Much work has been done in simulation on cooperative manipulation; we do not feel there is evidence that cooperative manipulation research done in simulation reflects the behavior of real-world robots performing similar tasks, due to a combination of assumptions about the mechanics of such systems and about available communication and sensing resources. Lynch and Mason ([LM94]) discuss a simulator and motion planner for single-robot pushing that is based in sound mechanics; this simulator may yet be extended to multiple-robot manipulation, but this has not yet been done.

5.2 The Robot as Nonholonomic Constraint

We are using cooperative manipulation protocols in which two robots move a box (or other object) using what we call a *pusher-steerer* configuration, in which the robots take on the role either of the *pusher* (depicted in figure 5.2a), in which

- Torque-controlled translations push the object in front of the robot,

[1] This is not obvious; see **references**.



(a) (b)

Figure 5.2: The pusher/steerer model: (a),The pusher and (b),the steerer

- the robot follows the object by continually turning[2] to align its front bumpers with the rear face of the object (the rotational and translational motions here are decoupled and occur in parallel), and

- the robot doesn't know the path that the object is supposed to follow.

or the *steerer* (figure 5.2b), in which

- The robot knows a path that it is supposed to follow,

- the robot is translationally compliant (it controls the heading of its wheels, but does not control their rate of rotation), and

- the robot moves forward as a result of being pushed by the object (which is itself being pushed by the pusher).

and the object sits between the robots. We use **no explicit communication** (i. e., no form of transmitted data) between the two robots, but allow only *implicit* communication through the mechanics of the robots-and-box system. The configuration is conceptually similar to a rear-wheel-drive automobile that has been sliced into three sections: the rear wheels push the passenger compartment forward in a direction determined by the front wheels. The challenge to this configuration is that the pieces *are* separate – the robots have to be programmed to allow flexible trajectory following, while keeping the object confined between them. The advantage

[2]Our protocols assume that the robots have (say) wheelbases that can simultaneously and independently move forward and change heading. Our robots have either synchro-drive or treaded bases that meet this condition.

of this configuration is that the robots can trade roles – the pusher can become the steerer and vice-versa. This increases the flexibility of the protocol by allowing such maneuvers as the "back-and-fill" that automobile drivers use for turning cars around on narrow roads and for parallel parking in tight spaces (we describe this in section 5.2.3). It also permits the robots-and-object system to reverse direction without the robots having to move to different faces on the object. Surprisingly, no explicit communication is required for either back-and-fill maneuvers or reversal of direction. A very exciting aspect of these results is the following claim: the pusher-steerer model of manipulation effectively reduces the problem of trajectory-planning for large-scale manipulation to the problem of nonholonomic motion planning. Since Lynch and Mason ([LM94]) and Barraquand and Latombe ([BL93]) have recently published practical nonholonomic path-planners (for single-robot pushers and train-like robots, respectively), it would appear that the technology for planning and executing complex manipulation trajectories is now available!

In this chapter, we present protocols for manipulating large objects with a pusher/steerer system. We first present and analyze simplified protocols that allow for translational manipulation only. We then expand these protocols to allow for circular-arc following. Finally, we modify the protocols to compensate for physical limitations of the robots on which they are implemented (i. e., limited contact information, limited translational compliance, and limitations on maximum rotational and translational torque). For analysis purposes, we have chosen the following model for manipulation tasks: each robot is a point mass with ideal wheel kinematics. We assume the wheels are perfectly rough – the wheels won't slip relative to the floor under any application of force. We have torque control over the wheels, and can turn the robots in place. Our analysis uses Coulomb friction and quasistatic dynamics, and treats the world as planar – the objects we manipulate are 3D right extrusions of planar objects. Each object has an effective center-of-friction (that can move over time). If more than μmg net force is applied against the object (where μ is the coefficient of friction and m is the mass of the box), it will translate; if more than $\mu\rho mg$ net torque is applied around the center-of-friction, the object will rotate. Section 6.3 summarizes the results of several experiments we have conducted using TOMMY and LILY to test these protocols.

5.2.1 Translation

We begin by defining protocols allowing a pusher robot and a steerer robot to translate an object in a straight line. Table 5.1 lists and describes some of the motion

Table 5.1: These are some of the motion control primitives used in LILY and TOMMY's implementations of the protocols described in this chapter.

Command	Selected Options	Brief Description
`rotate-halt`		lock the rotate motor / servo to current heading
`translate-halt`		lock the translate motor / servo to current position
`translate`	`'by,'torque, 'vel,'until`	actuate the translate motor
`rotate`	`'by,'torque, 'vel,'until`	actuate the rotate motor
`rtc-approx`	`contact-mode`	rotate until object contacting robot is in specified contact-mode
`tolerate-impediments`	`form`	exit gracefully (non-fatally) from `form` when torque-limit is exceeded, a non-local exit mechanism using `unwind-protect` ([CR92])
`rotate-limp`		turn off the rotate motor (become rotationally compliant)
`translate-limp`		turn off the translate motor (become translationally compliant)
`rotate-where`		read the rotate motor shaft encoder
`translate-where`		read the translate motor shaft encoder
`translate-power`	`torque`	supply current to the translate motor at the specified power level
`translate*`	`inner-thunk,'by, 'torque,'vel, 'until`	The same as `translate` but executes an arbitrary specified thunk at every iteration of the control loop. When the thunk returns `#t` or distance 'by is completed, the motion is terminated, and `#t` is returned.

control primitives used in the protocols described in this chapter. To translate an object by distance d, the steering robot executes protocol STEERER0:

```
(define (steerer-0 d)
  (rotate-halt)
  (translate ‘by d ‘torque 0)
  (translate-halt)
)
```

The `translate-halt` is how the steerer communicates to the pusher (through the object) that it is time to terminate the motion. For the translation only case, and if we assume that the pusher and the steerer are aligned, it is not necessary for the pusher to align to the rear face of the object (the reason for this becomes clear in the analysis presented later in this section). The pusher, therefore, executes protocol PUSHER0:

```
(define (pusher-0)
  (rotate-halt)
  (translate ‘torque *pushing-torque*)
  (translate-halt)
)
```

Figure 5.3 shows the approximate configuration. For now, assume that the pushing robot and the steering robot have their wheels aligned so that their motions are in parallel directions, and that there is no friction between the object and the robots. Finally, assume that the force required to translate the steering robot is small relative to the force required to translate the box (this can be achieved in practice by applying a small torque bias to the steerer's translation motors, `(translate ‘by d ’torque *small*)`, so that the steerer requires extremely low force to overcome its inertia). If the pushing force is greater than the sum of the static friction of the object and the static friction of the steering robot, then the box will translate, pushing the steering robot along ahead of it. If the line of pushing and the line of steering (the lines parallel to the pushing and steering robots' wheels and passing through the robots' points of contact with the box) are the same, and the center-of-friction (COF) of the object lies on that line (as shown in figure 5.4a), then no object rotation will occur. If the center of friction lies between the lines containing the two robots' force vectors (figure 5.4b), the object will rotate around a point between the COF and the instantaneous point of contact between the box and the steering robot. The box will continue to rotate until the angle between the pushing direction and the side of the box facing the robot is such that either the pusher slides off of the end of the box, or the kinematic jamming condition shown in figure 5.5 occurs. This condition (which we will refer to as *jamming closure*) is caused by the wheel kinematics of the robots – if the box is wider than the distance between the two robots' pushing lines, then the box will jam, instead of rotating until it can slip between the two robots. When the box jams, it ceases rotating, and begins translating in the direction it is pushed. If the line of steering is between the COF and the line of pushing (figure 5.4c), the behavior will be the same as for figure 5.4b. If the line of pushing is between the COF and the line of steering (figure 5.4d), then the box will rotate, while the pusher and steerer slip along the box until the pusher slips off of the end of the box. In this configuration, the jamming closure cannot occur, so the box will never translate.

When we take friction into account, the conditions necessary for the robots to translate the box become less restrictive. This is due to the existence of a closure

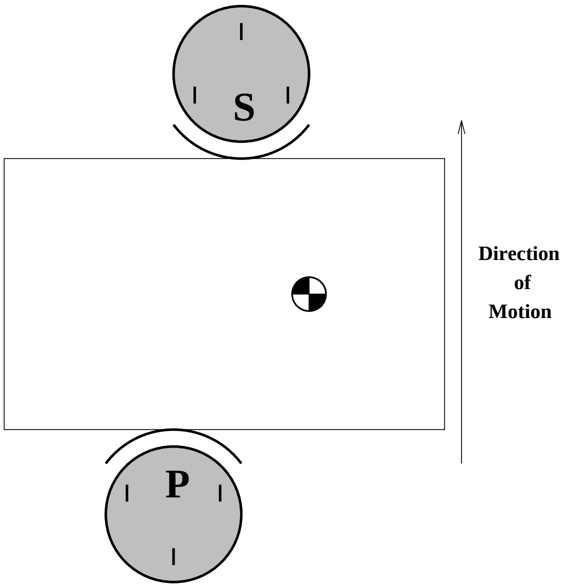


Figure 5.3: Two robots translating a box using the pusher-steerer model

based on frictional forces that is easier to achieve than the jamming closure. Let's first consider the extreme case, where the coefficient of friction is infinite. Because of the infinite friction, the contact points between the robots and the box are fixed (see figure 5.6). In this case, we can ignore the COF of the box. Considered from the reference point of c_s, the pushing robot exerts a torque around c_s. Since the contact points are fixed and the box is rigid, the only way the box can move is to rotate about the point c_s That would require the pusher's contact point, c_p to follow an arc (the dashed arc in figure 5.6). The kinematic restrictions on the pushing robot, however, prohibit this motion. This means that the box cannot rotate at all. The only motion it is permitted is translation parallel to the directions of F_p and F_s.

The tools we use to analyze what happens when a finite amount of friction is present in the system are drawn most directly from Mason's work in [LPMT84] and [MS85], and have also been used by Erdmann in [Erd84] and Donald in [Don87, Don89]. The main tool we use is the concept of the *friction cone*, which is an abstraction used to analyze whether one object pushed against another with a given force vector will stick or slide. If we have a point contact between two objects, for example, we erect a cone whose main axis is normal to the surface of one of the objects at the point of contact, and whose sides have angle $\tan^{-1}(\mu)$, where μ is the coefficient of friction between the two objects. If the force pushing the objects together has an angle that lies within the angular range spanned by the cone, then the two objects stick at that point of contact; otherwise, the point of contact slides.

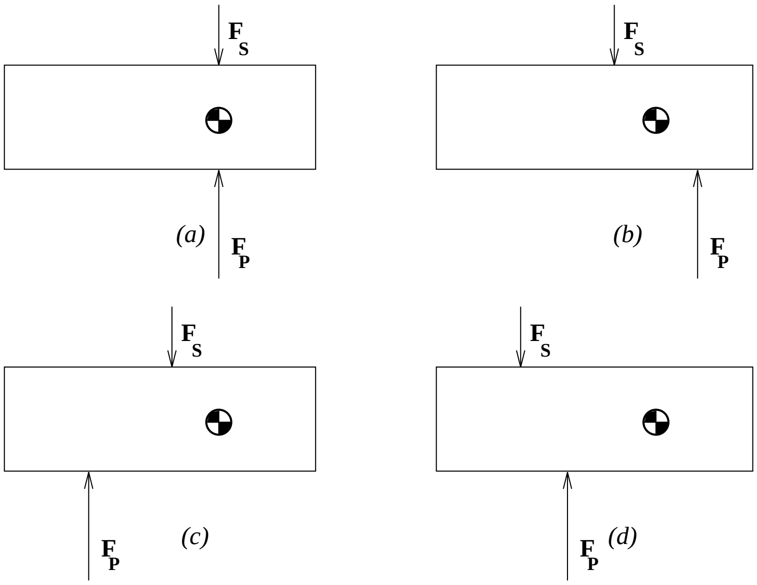


Figure 5.4: Four possible arrangements of a pushing robot, a steering robot, and a pushed object

For our problem, we cite analysis from [MS85] that lets us conclude the following: If the COF of the box lies in the intersection of the inverted (flipped across the contacted surface) friction cones erected at the pusher's point of contact and the steerer's point of contact, then the points of contact will not slide; in this case, the box will translate without rotating. If the COF lies outside the inverted friction cones, then the points of contact will slide, and the box will rotate until (i) the COF enters the intersection of the two friction cones, (ii) the box achieves jamming closure, or (iii) one of the robots slides off of the box. The box begins translating in the direction of the pusher's force vector as soon as (i) or (ii) happens.

5.2.2 Pushing an Object Through a Circular Arc

We next extend our analysis of constrained manipulation from pure translational pushing to pushing along a circular arc. First, we analyze the differential behavior of a pusher-steerer system when the pusher and the steerer are not aligned to face in opposite directions (when their pushing lines are not parallel). Then we perform a rough analysis of the extended behavior of such a system as it follows a circular arc of fixed radius. Finally, we describe the programming of the robots to perform this task, and describe how we program the steering robot to determine the minimum feasible turning radius of the system in an on-line fashion, allowing us to program

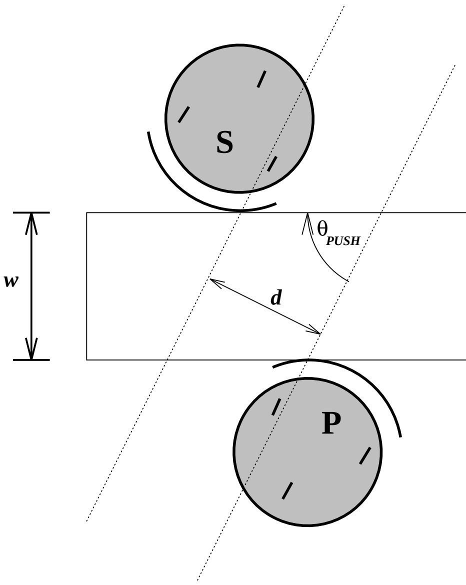


Figure 5.5: The kinematic jamming closure: when $d \leq \frac{w}{\sin\theta_{PUSH}}$, the box stops rotating: in order for the box to rotate further, it has to increase the perpendicular distance between the line containing the pusher's force vector and the line containing the steerer's, which it cannot do because of the wheel kinematics of the robots.

---

arbitrarily tight turns with the knowledge that the robots will approximate them as best as they can, subject to physical constraints.

As depicted in figure 5.8, two robots can steer a box along a circular arc by maintaining a fixed difference in alignment relative to the box. The pusher/object/steerer system moves along the specified arc in the following way: the pusher runs a two-threaded protocol in which it translates at a specified torque while continuously servoing its heading so that its front bumpers are aligned to the back side of the box. The steerer ignores the box completely; it allows itself to be pushed along an arc of specified radius and length. As the point of contact between the steerer and the box moves (say) rightward on the box (in the pusher's frame of reference – i.e., rightward on the page as we look at figure 5.9), the force from the pusher causes the box to rotate so that the left end comes forward (clockwise rotation). Figures 5.9 and 5.10 show the forces and torques exerted on the box by the robots. In general, at a given instant, the box will be in one of three rotation modes: It will rotate in the right direction, in the wrong direction, or not at all (pure translation). As the trajectory continues to evolve, however, wrong-way rotation and pure translation are unstable states, since the steerer's point of contact is moving along the front face of the box. Eventually, the steerer's point of contact with the box will move far

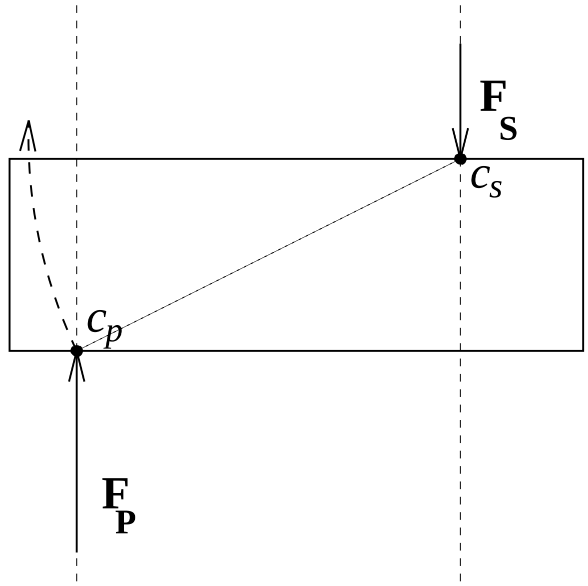


Figure 5.6: With infinite friction, the contact points between the box and the robots are essentially fixed. Because of kinematic restrictions due to the robots' wheels, the box cannot rotate.

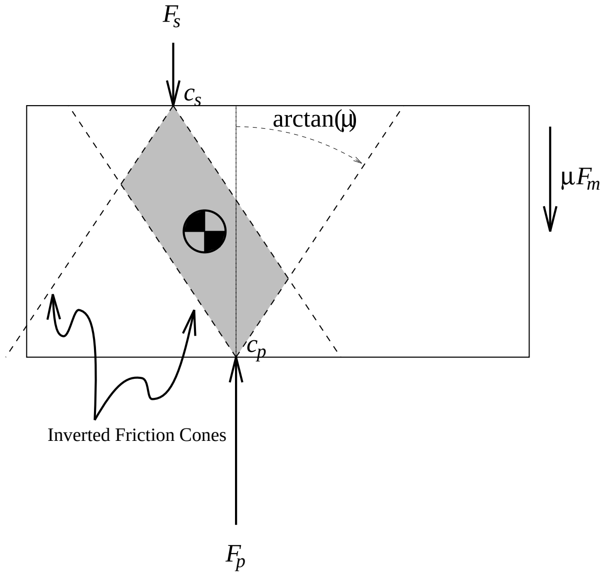


Figure 5.7: If, as shown, the center of friction lies within the inverted friction-cone rays, no slipping or rotation occurs. If $|F_p| > |F_s + \mu F_m|$, then the box will translate parallel to the direction of F_p. (μF_m is the maximum resistive force of the box due to friction with the supporting surface(μ is the coefficient of friction between the box and the supporting surface; F_m is the weight of the box (force due to mass))).

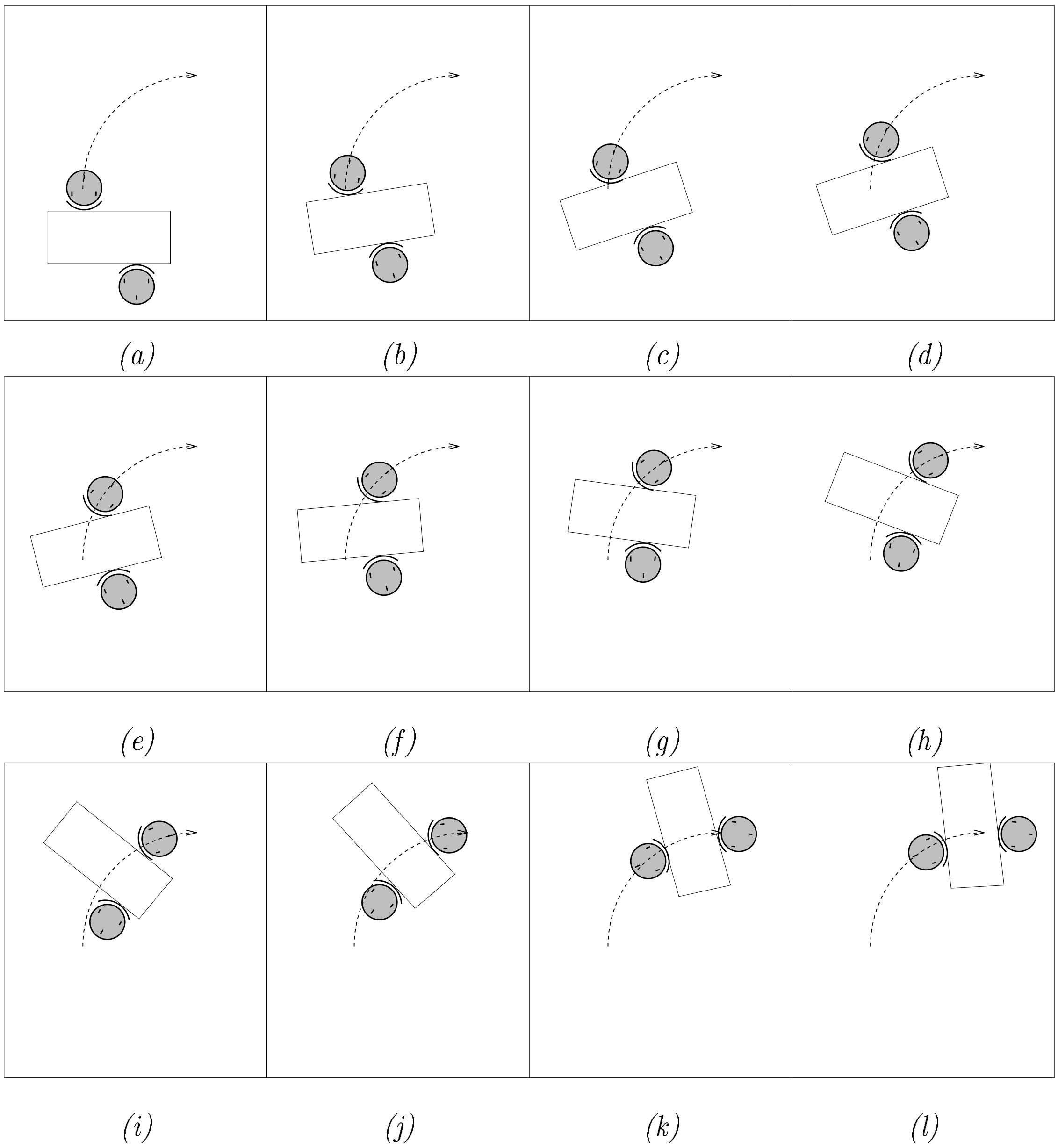


Figure 5.8: This series of figures depict a box being guided through a 90 degree arc by a steering robot (in front, following the arc), and a pushing robot. The box begins with its front and rear faces approximately perpendicular to the path. In parts *(b)* through *(e)*, the box rotates in the wrong direction, due to poor initial placement of the pusher relative to the steerer. By part *(f)*, the pusher, with no model of the box or the path and with no communication, has compensated for the poor initial configuration. By part *(l)*, the box has traversed the arc and rotated until its front and rear faces are approximately perpendicular to the path.

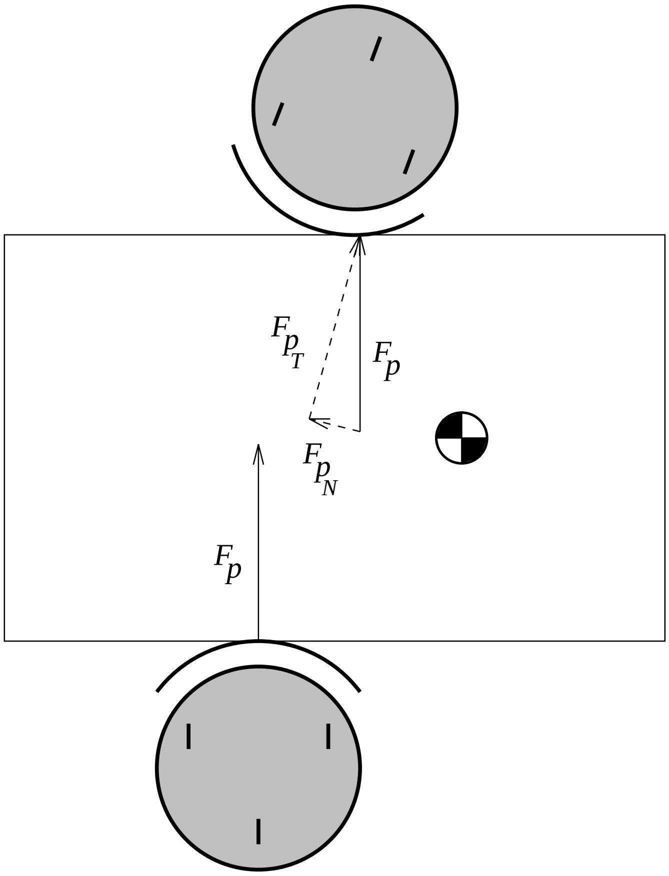


Figure 5.9: The pusher's force, F_p, transmits to the steerer through the box. F_{p_T}, the component of that force that is parallel to the steerer's wheels, pushes the steerer along its current trajectory. F_{p_N}, the component that is perpendicular to the steerer's wheels, is counteracted by the resistive force of the wheels.

enough along the front face that the torque balance will cause the box to rotate in the desired direction. There are two limitations to this set of protocols. One is that the steerer can slide off of the front face of the box, thereby losing control of it. The other is that a combination of box-steerer friction and relative alignment of pusher and steerer wheels can prevent the pusher from forcing the steerer to roll, rendering the system unable to move. In practice, we avoid these situations by restricting the turning radius of the system either to an experimentally predetermined minimum (typically 750mm to 1.5m for normal-sized boxes) or by using the contact angle between the steering robot and the steering face to control the minimum turning radius at runtime. It is possible to compute the minimum turning radius analytically in terms of box dimensions and coefficients of friction, but this has not yet been done.

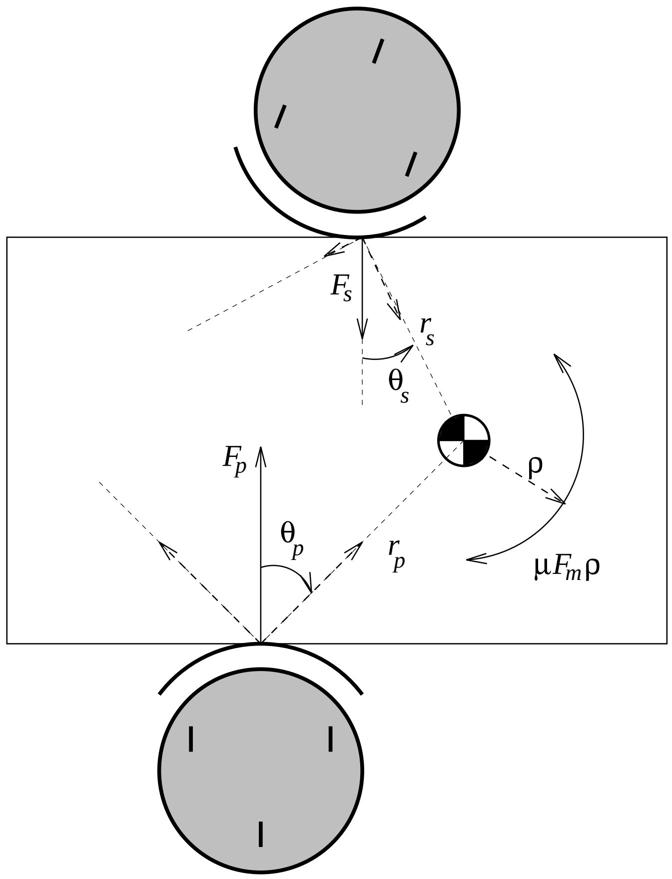


Figure 5.10: The robots exert torques on the box based on the normal component of their forces on the box (F_p and F_s), their distances from the COF (r_p and r_s), and the angle between the surface normals at the points of contact and the lines to the COF (θ_p and θ_s): If $|F_p r_p \cos(\theta_p)| > |F_s r_s \cos(\theta_s) + \mu F_m \rho|$, then the box will rotate as indicated by the sign of the pushing torque, $F_p r_p \cos(\theta_p)$. ($\mu F_m \rho$ is the maximum resistive torque of the box due to friction with the supporting surface, where ρ is the radius of gyration of the box.)

```
;; We implement concurrent push-and-align in the following way:  translate
;; takes an arbitrary thunk for a termination predicate.  The thunk executes
;; for every execution of the translate servo-loop.  We get concurrent push-
;; and-align by specifying a thunk that executes "if we're not in the right
;; contact mode, do rtc-approx to the right contact mode, then return #f."

(define (pusher)
  (if (impediment?
        (push-track *pushing-robot-speed*
                    (lambda ()
                      (if (not (zero?
                                 (- (contact-mode)
                                    *pushing-tracking-contact-mode*)))
                          (rtc-approx *pushing-tracking-contact-mode*)
                      )
                    #f
                    )
        )
      )
  )
)

(define (push-track speed tp)
  (let ((tracking-contact-mode *pushing-tracking-contact-mode*))
    (push-until-contact)
    (if (tp) 'arrived
        (begin
          (let ((result (begin
                          (push-until-contact)
                          (tolerate-impediments
                           (translate*
                            'vel speed
                            'torque *pushing-tracking-torque*
                            'until
                            (lambda (s)
                              (if (tp) 'arrived
                                  (not (contact?)))))))))
            (case result
              ('impediment 'impediment)
              ('arrived 'arrived)
              (else 'lost-contact)))))))
```

Figure 5.11: Protocol PUSHER incorporates the concurrent-align servo loop (in the manner described in the comment block) that keeps the pusher perpendicular to the back face of the object.

```
(define (steerer turn-angle turning-radius bad-contact-modes)
  (let ((distance (- (round/ (* (* (abs turn-angle) turning-radius) 22)
                             (* 7 180))))
        (finish-angle (+ (rotate-where) turn-angle))
        (gain-num 1)
        (gain-den 1)
        (rotate-base (rotate-where))
        (translate-base (translate-where)))
    (set! gain-den (round/ distance turn-angle))
    ; compute numerator and denominator of translate-vs-rotate gain based
    ; on the specified turning radius.
    (translate-limp)
    (let loop ()
            (steerer-aux (+ translate-base bad-contact-distance)
                     rotate-base gain-num gain-den)
            ; steerer-aux actually performs the necessary rotation.
         (if (or (and (< turn-angle 0)
                      (> (rotate-where) (+ rotate-base turn-angle)))
                 (and (> turn-angle 0)
                      (< (rotate-where) (+ rotate-base turn-angle))))
             (loop)   ; if we haven't gone far enough along the arc, loop.
         ))
    ; lock the translate motors to signal 'done' to pusher.
    (translate-halt)
    (sleep (* 2 one-second)) ; stay locked for a couple of seconds.
    (translate-limp)  ; go translate-limp on exit.
  )
)
; steerer-aux is the inner control:  it performs a one-step rotation based
; on the gain values (determined from turning radius) and the initial values
; of translate-where and rotate-where.

 (define (steerer-aux translate-base rotate-base gain-num gain-den)
   (rotate 'by (- (round/ (* gain-num (- (translate-where) translate-base))
                          gain-den) (- (rotate-where) rotate-base))
           'on-exit (lambda (r) (rotate-halt) r)))
```

Figure 5.12: Protocol STEERER follows a circular arc of the specified radius and angular extent. The robot is translationally limp (no current is applied to the translate motors), so it traverses the specified arc as it is pushed.

The actual programs our robots use to perform circular-arc-following are very simple. The pusher runs protocol PUSHER (figure 5.11).[3] For following fixed arcs, the steering robot runs protocol STEERER (figure 5.12), which implements a servo loop that commands a new heading determined as a function of distance travelled: If the steerer is to follow an arc of radius r_t through an arc of ρ_t degrees, then it continuously computes its desired heading, $\rho_s(t)$ as a function of

1. r_t, the specified turning radius,
2. ρ_t, the specified arc length,
3. $\tau_s(t)$, its current translation shaft-encoder reading,
4. $\tau_s(0)$, its initial translation shaft-encoder reading, and
5. $\rho_s(0)$, its initial heading.

$$\rho_s(t) = \rho_s(0) + \frac{\tau_s(t) - \tau_s(0)}{r_t \rho_t} \tag{5.1}$$

The `(translate-halt)` command is issued by the steerer when the appropriate arc length has been traversed:

$$\frac{\tau_s(t) - \tau_s(0)}{r_t \rho_t} = 1$$

In the `steerer` function of figure 5.12, ρ_t and r_t are the specified `turn-angle` and `turning-radius`. $\rho_s(0)$ and $\tau_s(0)$ are `rotate-base` and `translate-base`, which are determined by reading the rotate and translate encoders using `(rotate-where)` and `(translate-where)`. The function `steerer-aux` determines $\tau_s(t)$ from another call to `(translate-where)` and computes $\rho_s(t)$. It then commands a differential rotation computed as the difference between the desired heading, $\rho_s(t)$ and the current heading, `(rotate-where)`.

5.2.3 Following General Trajectories

Given our analysis for pushing an object along a straight line or circular arc, we conclude that our system can push a given object through an arbitrary trajectory, subject to three restrictions:

[3]The SCHEME code implementing this protocol uses a rather subtle mechanism to push and align concurrently: the `translate*` call in the procedure `push-track` incorporates an execution of `tp` into the translate motor control loop. `tp` is assumed to be a thunk (a SCHEME function with zero arguments). The procedure `pusher` calls `push-track` with a `tp` designed to execute the rotate-to-contact-mode function, `rtc-approx`. Thus, the alignment portion of the protocol is effected by the user-specified portion of the inner control loop for `translate*`.

1. The minimum radius of curvature at any point on that trajectory must be no less than the minimum radius for a circular arc through which the robots can guaranteeably push that object (typically 750mm to 1.5m for a box, slightly more for larger objects).

2. The speed at which the robots and the object move along a trajectory must be low enough that the robots' control systems can respond sufficiently quickly to stay on the trajectory and that the object's dynamics remain quasistatic.

3. For the robots to navigate a box with length ℓ along a trajectory, each point on the specified trajectory must be at least distance ℓ' from the nearest obstacle, where we conjecture ℓ' to be equal to ℓ.

These restrictions apply when the pusher and the steerer cannot trade roles. If we run a back-and-fill protocol on both robots, they can use a multi-step pushing strategy to navigate sharper turns and narrower free spaces. A pusher/steerer system capable of executing back-and-fill motions needs three main skills: it must be capable of

1. planning the sequence of straight-line and circular-arc segments needed to traverse the specified path (This includes specifying which robot is the pusher and which is the steerer for a given path segment.),

2. executing the individual path segments, and

3. allowing each robot in the system to determine at each moment whether it is a pusher or a steerer.

The PUSHER and STEERER protocols we've presented provide the second skill. The third skill can be implemented without direct communication using torque and time limits. Figure 5.13 shows the protocol ROLE-REVERSE, which is one way we can do this. Note that, for determining when the robots should switch modes, ROLE-REVERSE is a strictly online protocol. With regard to the first skill, it is beyond the scope of this thesis to analyze the fine-motion aspects of how a path, such as might be generated by a generic path planner, is converted to a sequence of backs and fills executable by the pusher/steerer system. We do observe, however, that there are at least two possible ways to implement that conversion process. An offline planner could be employed to analyze the path and convert it into a series of forward and backward motions, which the robots could then execute. Lynch and Mason's planner to find *stable pushing paths* ([LM94]), or Barraquand and Latombe's planner for train-like (wheeled, multicarted) robots ([BL93]) might be suitable for this purpose. One drawback to this approach is that both robots would need to know at least their own portion of the plan[4]. An online approach capable of solving this problem (albeit, probably for a smaller class of paths) would be for the pusher robot to remember whether it was going clockwise or counterclockwise before it assumed the role of the steerer, and then to "back" by rotating the opposite way while allowing itself to be pushed for a fixed distance. Upon backing for a fixed distance or encountering an obstacle, it would stop moving, causing the other robot to reassume the steering role.

5.2.4 Gains over Single-Robot Arc-Following

It is theoretically possible to navigate an object in a straight line or along a circular arc using a single robot. If the robot knows the location of the object's COF for a given pushing direction, for example, it can push the object in a straight line with no error, assuming that the COF doesn't change with time. If the robot doesn't know the COF, or the COF changes position with time (generally the case for a mobot pushing a real object across a real surface), the robot must continually sense the object's relative orientation and position and compensate for any drift from the planned path that occurs. Particularly, if the robot doesn't know the COF, it must first ascertain the approximate location of the COF, and then position itself behind the COF, and compensate for any drift that might have occurred while the COF was being located. Only then can the robot commence straight pushing to follow the planned line. With a two-robot pusher/steerer system, the location of the COF is almost immaterial. Because of the effect of the steerer's wheel kinematics on the perceived COF of the steerer/object system (as perceived by the pusher), execution of PUSHER causes the object to remain on the line of pushing for almost all locations of the object's COF.

It is similarly possible for a single robot to navigate an object along a circular arc, if it knows the object's COF. In practice, however, when the COF is unknown and can move, there exist instantiations of a single robot pushing an object for which the minimum turning radius is effectively infinite. Consider, for example, the object in figure 5.14. If the object has its COF located right at the corner near the robot, the robot cannot rotate the object counterclockwise while pushing it along any circular arc with finite turning radius, without acquiring a different face of the

[4]If we could devise a system whereby the "front" robot could steer the back robot, even when the front robot is the one pushing, the robots-and-box system would be similar to one of Barraquand and Latombe's train-like robots (where the first car is the only one motorized) moving in reverse. Given this operation, we could use their planner in situations where only one robot knew the path.

object. If the COF is very near the corner, the minimum turning radius will be quite large. As with straight-line two-robot pushing, though, a two-robot pusher/steerer system can push this object around reasonably sized arcs with no need either to know the box's COF or for either robot to acquire another face: if the pusher in a pusher/steerer system is located in the same place as the single robot shown in 5.14, then counter-clockwise rotation of the box is easy. Clockwise rotation, on the other hand, may be difficult from this position. In practice, since the pusher doesn't need to know the position of the COF, it will try to avoid this difficulty by centering itself on the pushing face.

In an attempt to provide a bit more intuition suggesting that the pusher/steerer protocols outperform single robot manipulation algorithms, we provide the following sketch of how one might quantify the increased capability of the two-robot system more formally: Lynch and Mason, in [Lyn92,LM94], analyze controllability with point pushing contact, noting that "open-loop pushing with point contact is inherently unstable." They present a proposition: "The configuration of a slider S with bounded support friction $s(x)$ is controllable by pushing if and only if the pusher can apply two pushing force directions, $\hat{f}_1$ and $\hat{f}_2$ such that they do not both pass through the center of friction, and $\hat{f}_1 \neq -\hat{f}_2$." In the absence of friction between the pusher and the object, this condition will not be true for a single point pusher pushing a convex object: controlled pushing can happen only if the pushing force is aimed directly at the center of friction. As we demonstrated in section 5.2.1, the addition of a compliant body with wheel kinematics on the far side of the object creates a situation in which the pushed object is controlled via a closure that does not depend on friction. While we demonstrated this only for the limited protocols PUSHER0 and STEERER0, we conjecture that a similar result can be obtained for PUSHER and STEERER. In the presence of such a result, we conclude that, at least in the case of zero pusher/object friction, the addition of a steerer enables the pusher to exert controlling forces in more than one direction. This leads us to believe that the two-robot pusher/steerer system has controllability over a manipulated object in ways which a single robot pusher does not. While this argument is not a formal proof, it provides some measure of formal backing to our intuition about the pusher/steerer protocols.

5.3 Contrasting the Theoretical Analysis with the Experimental Setup

In section 5.2, we analyzed a pusher/steerer model for translation in some detail and a pusher/steerer model for circular-arc following in lesser detail. In this section, we describe experiments we ran using pusher/steerer circular-arc following protocols on Tommy and Lily. The details of our experimental setup differ from those of the theoretical setup of our analysis in a number of important ways. The most significant of these are:

1. **Non-point robots**: because the robots have nonzero diameter, they can affect the manipulated object not only by pushing, but also by applying torque: as a robot turns, its contact with the box can be frictionful, causing the contacted surface to slide sideways at the point of contact.

2. **Imperfect translational compliance**: since the steerer doesn't have force sensing, it cannot be perfectly compliant, and therefore applies a nonzero resistive force against the pusher. This reduces the effective capacity (in terms of carrying capability) of the system.

3. **Non-uniform friction**: the friction between the robots and the manipulated object and the friction between the manipulated object and the floor is not a constant. In addition, the effective center of friction varies over time as the box moves across irregularities in the floor. We note that the pusher/steerer system is fairly insensitive to these effects, but we must recognize their existence.

4. **Finite applied and resistive forces**: the pusher cannot apply infinite pushing force to the box-steerer combination. One effect of this is that the pusher may not be able to force the steerer to roll if the difference in heading between the pusher and the steerer is large.

5. **Finite servo rate**: because of the resolution of the robots' relative-heading sensing and the finite response speed of the robots' control systems, the steerer cannot follow a precise, accurate circular arc. Furthermore, the pusher cannot maintain a perpendicular contact with the object, and cannot even guarantee that the angle of contact will be within a few degrees of perpendicular.

Figure 5.15 depicts the contact-mode system that Tommy and Lily use for interpreting data from their bumpers. This system provides approximate relative-heading information: if the robot is in contact with a flat surface, the contact mode defines the approximate surface normal of that surface, relative to the robot's coordinate frame. When the robots execute a pusher/steerer action, the pusher servos to follow the pushed object based on the contact mode between the pusher and the object. To maintain a perpendicular heading, the pusher rotates to keep itself in contact mode 8. The steerer can use contact-mode information to help counteract some of the differences between the theoretical model and the physical system: The STEERER protocol, implemented on the idealized robots described in section 5.2, needs no information about the object's heading. When TOMMY and LILY implement the pusher/steerer model, however, TOMMY needs to take the object's heading into account: Since the pusher can apply only finite force and the steerer is not perfectly translationally compliant, we need to restrict the difference in heading between the pusher and the steerer, to prevent the pusher from terminating prematurely. Figure 5.16 shows the code for STEERER2. The code is similar, but now translational distance covered by the steering robot while in a bad contact mode (one where the angle between the steering robot and the object face) is not counted as part of the control input to the steerer's rotational servo-loop. This allows the steerer to follow a path that has the same angular extent as the specified arc but allows the system to avoid the forbidden contact modes.

5.4 Summary

We've been working on cooperative manipulation, with an eye toward on-line protocols that take direct advantage of the extra robots by allowing tasks that cannot be done by the participating robots individually, and that have low inter-robot communication requirements. We've designed constrained manipulation protocols that maneuver boxes and other large objects along complex paths using the pusher-steerer model. In this chapter, we've analyzed the theoretical behavior of pusher-steerer systems following paths consisting of straight segments and circular arcs. We have performed numerous experiments using the protocols we've designed, including one that maneuvered a two-foot-square box along a 50-foot-long trajectory consisting of four arcs and three straight segments, around couches and tables, through a three-foot-wide door. On the basis of these experiments, along with others, the results of which we describe in more detail in chapter 6, we feel that the pusher-steerer model outperforms other models of non-prehensile mobile robot manipulation.

```
(define (role-reverse segment-list initial-mode)
  (let loop ((current-pf-mode initial-mode)
             (remaining-segments segment-list)
             (old-translate-encoder 0)
             (steerer-thread #f))
    (if (= current-pf-mode 'pusher)
        (begin ; if we're the pusher this time
          (pusher 0)
          (set! current-pf-mode 'steerer)
        )
        (begin ; if we're the steerer this time
          (set! steerer-thread
                (spawn (lambda () (steerer (car remaining-segments)))))
          (let steerer-loop ()
            (set! old-translate-encoder (translate-where))
            (sleep *steerer-timeout*)
            (if (not (= (translate-where) *steerer-timeout*))
                ; if the translate encoder count has changed during the
                ; sleep period
                (steerer-loop))
            (kill-thread steerer-thread)
            (set! current-pf-mode 'pusher)
          )
        )
    )
    (set! remaining-segments (cdr remaining-segments))
    (if (not (null? remaining-segments))
        (loop)
    )
  )
)
```

Figure 5.13: Protocol ROLE-REVERSE takes a list of path segments and traverses them using the pusher/steerer model. It switches roles based on torque limit (going from pusher to steerer) and time (going from steerer to pusher). When the robot is in steerer mode, ROLE-REVERSE spawns a thread that handles the steering, while the ROLE-REVERSE procedure itself monitors time and the robot's motion. The main operations we can perform on threads are creation and destruction; in this case, ROLE-REVERSE creates the steerer thread when the robot enters steerer mode, and kills the thread when it senses (by elapsed time) the end of the steerer motion.

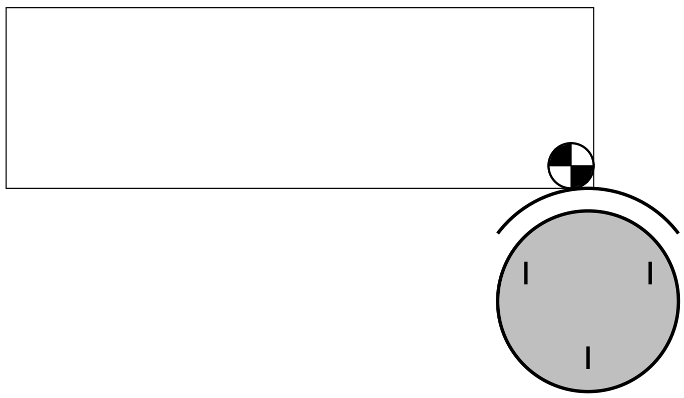
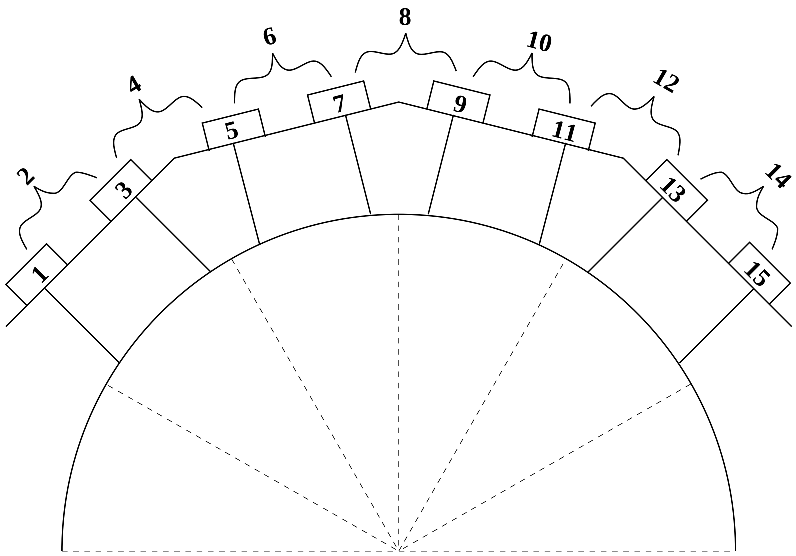


Figure 5.14: The object shown has a COF located right at one of its corners. There is no way for a single robot pushing on the lower face (as shown) to rotate the object counterclockwise. If the COF is very near the corner, the minimum turning radius will be quite large.



Contact Mode	Orientation	
	Max.	Min.
1	90.0	48.5
2	48.5	40.5
3	40.5	33.5
4	33.5	25.5
5	25.5	18.5

Contact Mode	Orientation	
	Max.	Min.
6	18.5	10.5
7	10.5	3.5
8	3.5	-3.5
9	-3.5	-10.5
10	-10.5	-18.5

Contact Mode	Orientation	
	Max.	Min.
11	-18.5	-25.5
12	-25.5	-33.5
13	-33.5	-40.5
14	-40.5	-48.5
15	-48.5	-90.0

Figure 5.15: TOMMY's bumper configuration: Against a flat surface, TOMMY's bumpers enter one of 15 contact modes. Odd-numbered contact modes occur when a single button is depressed; even-numbered contact modes occur when two adjacent buttons are depressed. Contact modes 2 through 14 each indicate the position of a contacted object relative to the robot with a precision of roughly 7.5 degrees.

```
(define (steerer-2 turn-angle turning-radius bad-contact-modes)
  (let ((distance (- (round/ (* (* (abs turn-angle) turning-radius) 22)
                             (* 7 180))))
        (finish-angle (+ (rotate-where) turn-angle))
        (gain-num 1)
        (gain-den 1)
        (rotate-base (rotate-where))
        (translate-base (translate-where))
        (bad-contact-distance 0)
        (previous-distance (translate-where))
        (was-bad #f))
    (set! gain-den (round/ distance turn-angle))
    ; compute numerator and denominator of translate-vs-rotate gain based
    ; on the specified turning radius.
    (translate-power (- *steerer-translate-power*))
    (let loop ()
      (if (not (member (contact-mode) bad-contact-modes))
          (begin
            (if was-bad       ; If we've just emerged from a "bad-contact"
                (begin        ; condition, update the total "bad-contact"
                              ; distance traveled so far.  We do this so
                              ; we can servo the steerer's heading based
                              ; solely on "good-contact" distance traveled.
                  (set! bad-contact-distance
                        (+ bad-contact-distance (- (translate-where)
                                                   previous-distance)))
                  (say bad-contact-distance)))
            (steerer-aux (+ translate-base bad-contact-distance)
                     rotate-base gain-num gain-den)
            ; steerer-aux actually performs the necessary rotation.
            (set! previous-distance (translate-where))
            (set! was-bad #f)
          )
          (set! was-bad #t)
          )
         (if (or (and (< turn-angle 0)
                      (> (rotate-where) (+ rotate-base turn-angle)))
                 (and (> turn-angle 0)
                      (< (rotate-where) (+ rotate-base turn-angle))))
             (loop)   ; if we haven't gone far enough along the arc, loop.
         )
    )
    (translate-limp)  ; go translate-limp on exit.
  )
)
```

Figure 5.16: STEERER2 is equivalent to STEERER, except that the steerer is restricted to gentle turns that prevent the steerer-object contact from staying in a "bad" contact modes.

Chapter 6

Experiments and Results

6.1 Research Program

In chapters 2, 3, and 5, we described a number of algorithms and protocols that we designed and implemented for our robots, TOMMY and LILY. In this chapter, we describe and present the results of experiments we performed to test those algorithms.

Section 6.2 shows maps that our robot, LILY has made of two rooms within our building. It also shows the results of five different runs of the localization algorithm, wherein LILY used the maps she made, along with instantaneous range data, to localize herself within those rooms. Section 6.3 describes experiments that we did to test the behavior of the pusher/steerer cooperative manipulation protocols, and summarizes the results of those experiments.

6.2 Mapmaking and Localization Experiments

Figures 6.1 through 6.10 show results of mapmaking and localization experiments performed using LILY. Figures 6.1, 6.2, and 6.3 show the results of experiments run in a small (about 4m by 4m) office. Figures 6.4 through 6.10 show the results of experiments run in our laboratory (with dimensions roughly 7m by 9m). For these experiments, we specified a map size of 256×256. The scale factor between the world and the map was 50mm = 1 cell. This let us build maps of areas with size up to 12.8m by 12.8m.[1] In the maps generated by LILY, the white regions are those which are most certainly vacant, while the darkest regions are those which are most certainly the boundaries of obstacles. Grey regions are those for which either

[1]For the office examples, we've cropped the map and localization grids to 128 by 128 so we can show the pertinent portions of the map more clearly.

the robot was unable to obtain data (for example, the interiors of obstacles) or the region was ambiguous (eg, occupied at some heights and vacant at others).

The rule the robot used to update cells when it made the maps is an exponential-decay update rule: probability is raised by the rule $p_{new}(x) = 1-(\alpha(1-p_{old}(x)))$ and lowered by the rule $p_{new}(x) = \alpha p_{old}(x)$. In practice, we use distinct α_{raise} and α_{lower} values, for raising and lowering probabilities, respectively. The values we used in the examples shown in this chapter were 0.3 for α_{raise} and 0.15 for α_{lower} (the same values were used for all tests – no retuning was necessary). Also, on the mobot, since we implemented the occupancy grid as an array of bytes, it was adequate to implement the update rules as 256-entry lookup-tables (in effect, making each cell of the map a 256-state finite state machine), speeding the update procedure considerably.

In section 3.2.3, we explained that in normal usage, we slant the laser rangefinder toward the floor so that the laser beams hit the floor about two meters from the robot. In the mapmaking and localization experiments described in this chapter, we used the laser rangefinder in this mode. Since the rangefinder was slanted toward the floor, we discarded those readings which were greater than 2.1 meters, since those corresponded to the rangefinder sensing the floor, rather than an obstacle. For these experiments, we also discarded any readings smaller than the rangefinder's minimum range of 0.3 meters.

In figures 6.2 and 6.3, we show the results of two different runs of the localization procedure in the office environment. The (a) parts of these figures show the hand-drawn map of the office with a circle depicting the actual location of LILY; the circle has the same diameter as LILY drawn to scale. (Note that LILY never uses or sees the hand-drawn maps – these are only shown here for the reader's convenience.) The (b) parts show the same circle overlaid on the map generated by LILY. The (c) parts show the output of the localization procedure run at the indicated location. In the localization output figures, we've coded the display in the following way: cells which are *not* in $FP(M, z)$ for any of the chosen range probes are white. Cells which *are* in $FP(M, z)$ for one or more range probes are shown in shades of grey, such that darker cells are consistent with more probes. The darkest cells (the black ones) are those which are consistent with a maximal number of probes. For these tests, we took probes at fixed intervals (one every 45 degrees). In test 1, there were cells consistent with seven of the eight range probes. In test 2, there were cells consistent with six of the eight. Note that the darkest cells are all within or very near to the circle denoting LILY's position.

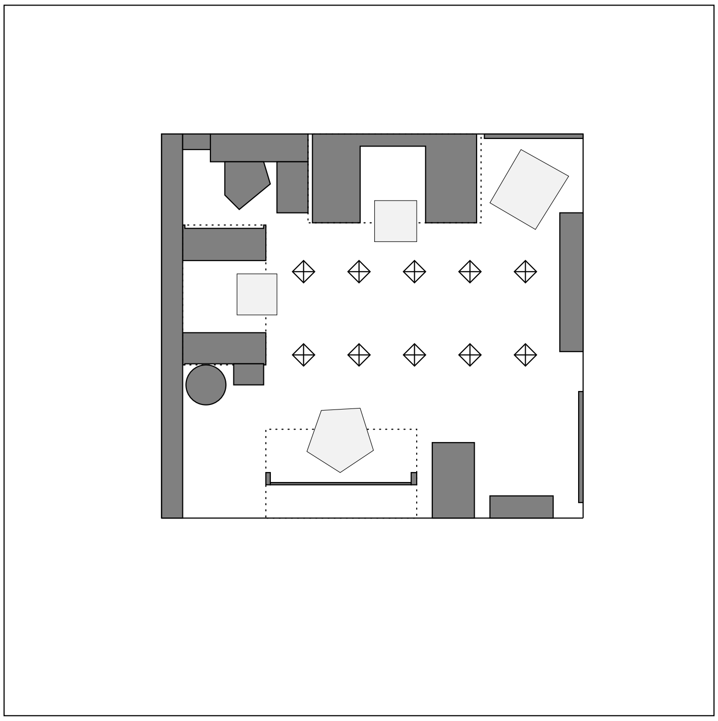
Figures 6.5 through 6.10 show the results of three different runs of the localization procedure in the lab environment. The (a) and (b) parts of figures 6.5, 6.7, and 6.9 are analogous to the (a) and (b) parts of figures 6.2 and 6.3. Figures 6.6, 6.8, and 6.10 are analogous to the (c) parts of figures 6.2 and 6.3. For these three runs of the localization procedure, the darkest pixels are, again, within or bordering on the circle denoting LILY's position for each run. In these cases, we took probes at 15 degree intervals. In test 3, 7 probes were within acceptable range. In tests 4 and 5, 10 and 11 probes were within range, respectively. For test 3, the maximal number of consistent range probes was 6; this value was 9 and 11 for tests 4 and 5.

6.3 Constrained Manipulation Experiments

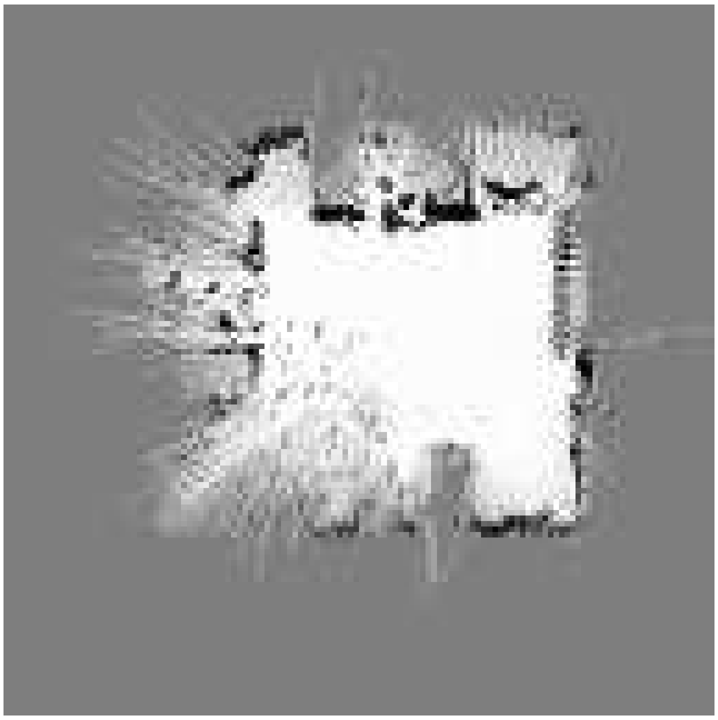
The first set of tests we ran on circular arc following was a simple endurance test: how far around a circle, on average, could TOMMY and LILY carry each of a set of test objects? For these experiments, LILY was the pusher, and TOMMY the steerer. LILY ran the PUSHER protocol (described in section 5.2.2) with zero contact-mode hysteresis (`pusher 0`). TOMMY ran the arc-following portion of the STEERER-2 protocol: `(rotarc-4 -1440 *current-turning-radius* '())` (The -1440 is the arc length in degrees, indicating a maximum arc-traversal of 4 complete circumferences. The `'()` means that all contact-modes are considered valid – for this set of experiments, we did not predicate the steerer's behavior on its contact mode with the box). We ran the protocols at each of a number of turning radii on each box 5 times, and present here (table 6.1) the average arc distance traversed before the steerer loses control. The maximum distance traversed for any test is 1440 degrees (four complete circumferences).

The circle-following test gave us data for fixed turning radii. We also want data on how well the pusher/steerer system behaves when we let the steerer determine on-line what the minimum feasible turning radius is. We ran the following test, for several values of `*bad-contact-modes*`, to obtain comparative performance data: The steerer alternately executes `(steerer-2 180 750 *bad-contact-modes*)` (a counterclockwise semicircle), and `(steerer-2 -180 750 *bad-contact-modes*)` (a clockwise semicircle), for a total of four semicircles, or nine meters, whichever came first[2] (the steerer follows the path shown in figure 6.11). STEERER2 monitors its contact mode with the manipulated object, and follows the specified arc only when the contact mode is good (i. e., not on the `*bad-contact-modes*` list); when the contact mode is bad, the steerer goes in a straight line when pushed

[2]Nine meters was the length of the largest open area available for these tests.

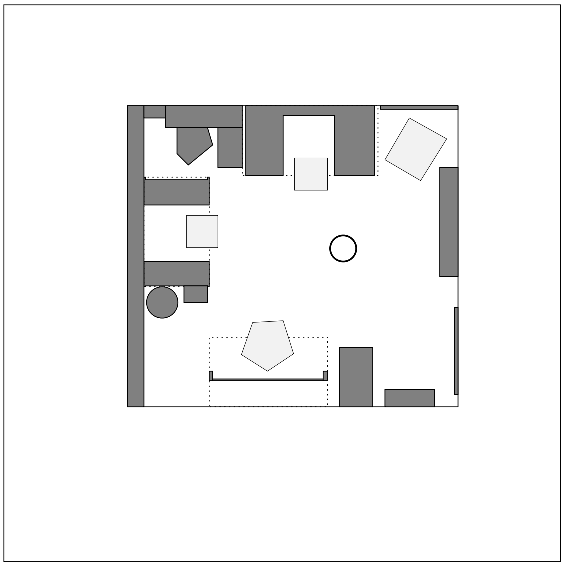


(a)

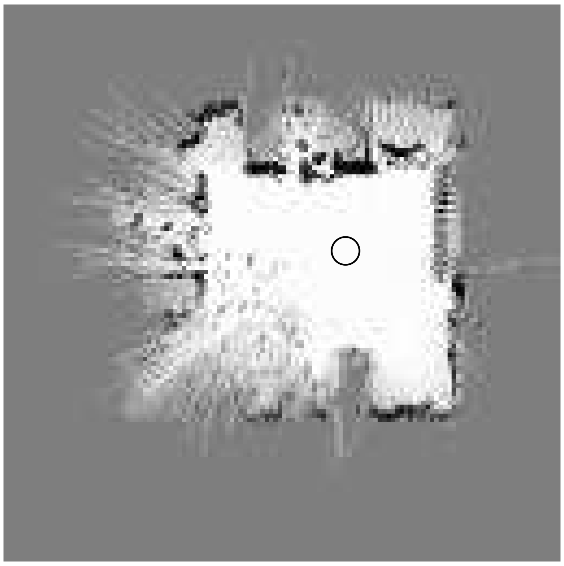


(b)

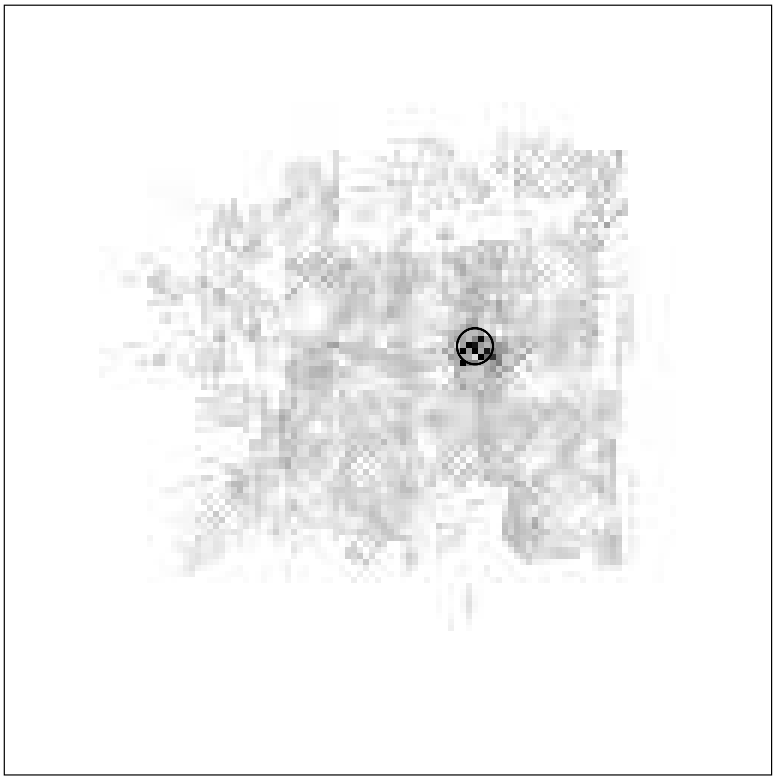
Figure 6.1: (a) A hand-drawn map of my office. The small diamonds are the places where LILY sat while making the map shown in (b). At each location, she performed map updates based on 300 readings.



(a)



(b)



(c)

Figure 6.2: The circles in these figures show the actual location of the robot for localization test 1. The circles are overlaid on (a) the hand-drawn map, (b) LILY's map and (c) the localization procedure output.

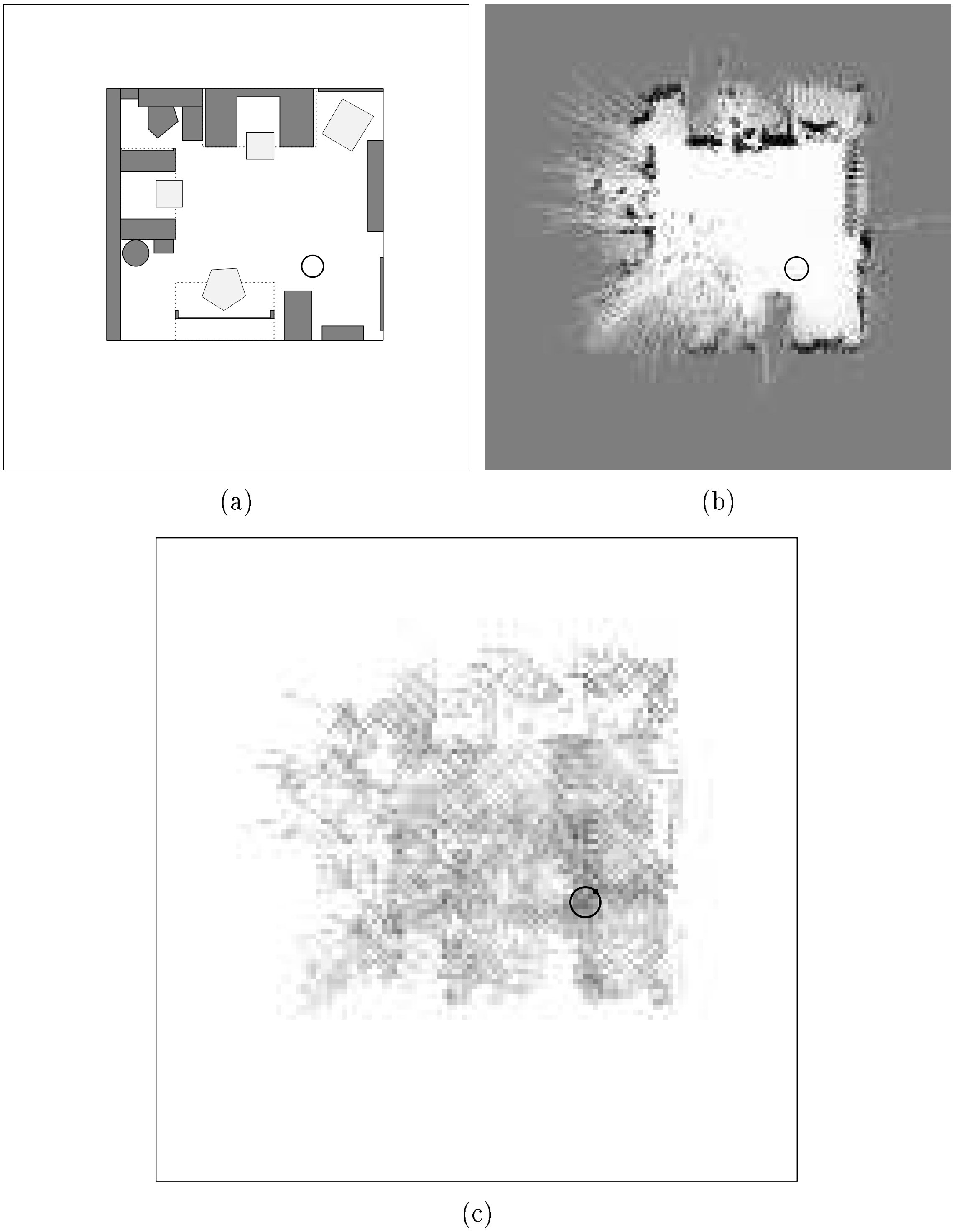
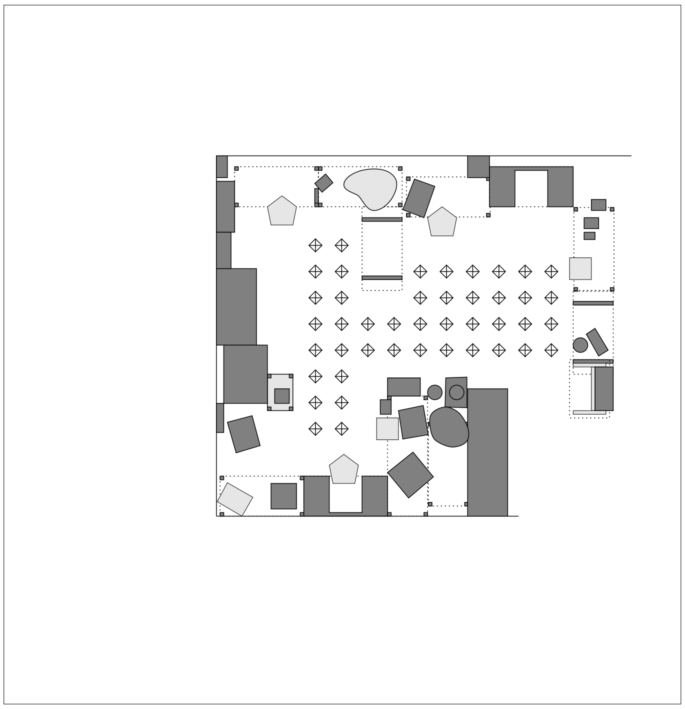
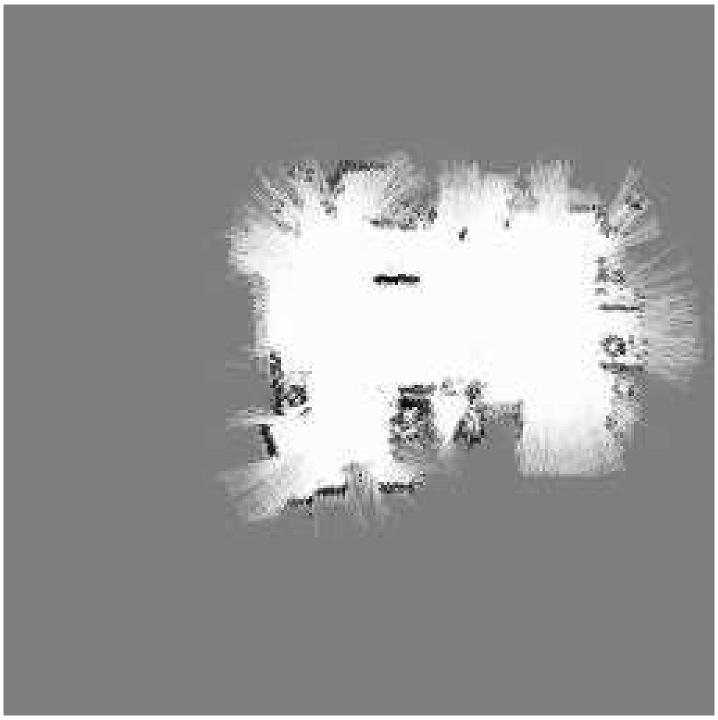


Figure 6.3: The circles in these figures show the actual location of the robot for localization test 2. The circles are overlaid on (a) the hand-drawn map, (b) LILY's map and (c) the localization procedure output.

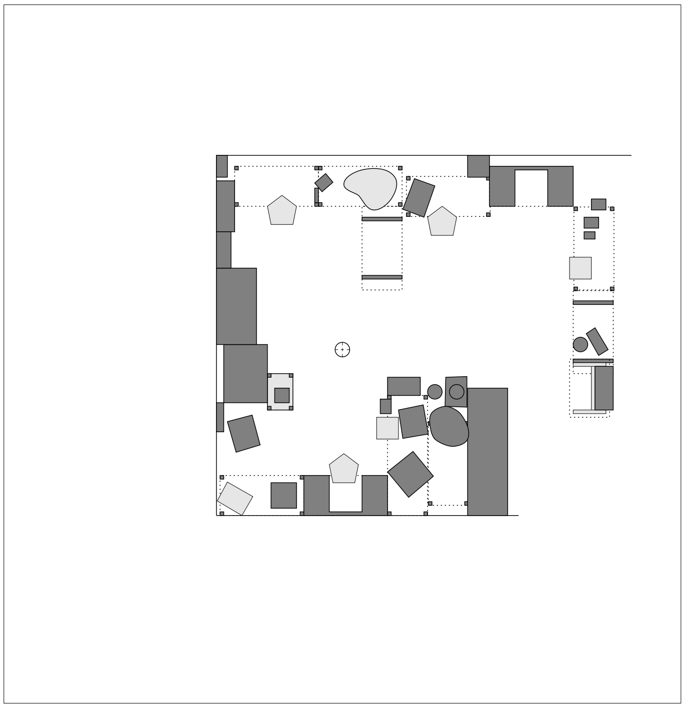


(a)

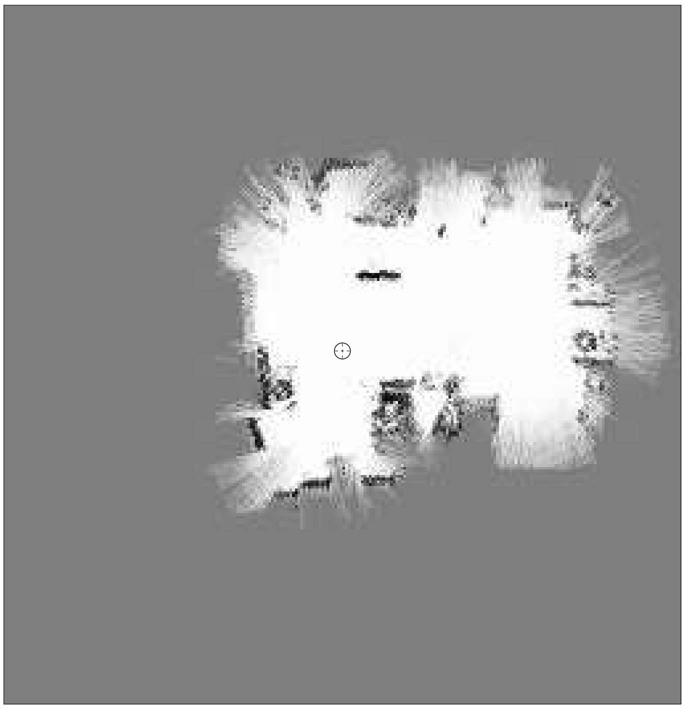


(b)

Figure 6.4: (a) A hand-drawn map of our lab. The small diamonds are the places where LILY sat while making the map shown in (b). At each location, she performed map updates based on 300 readings.



(a)

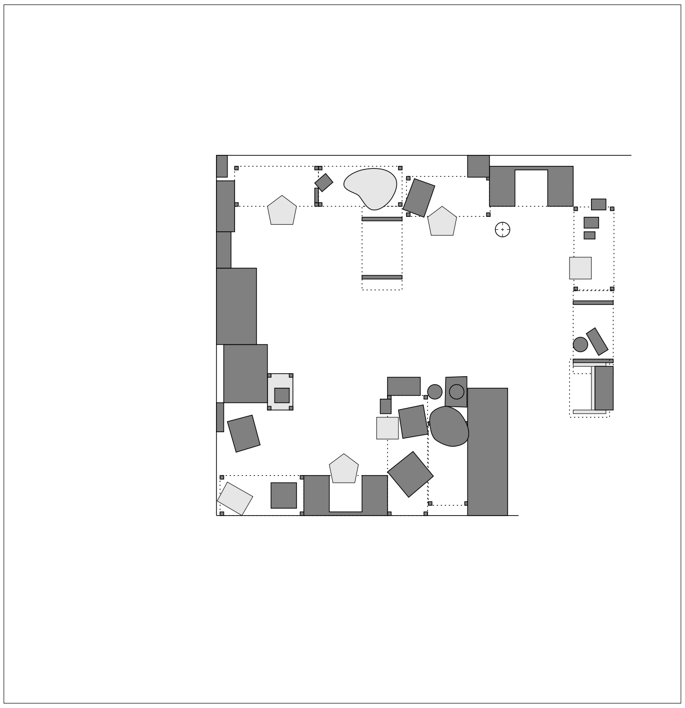


(b)

Figure 6.5: The circles in these figures show the actual location of the robot for localization test 3. The circles are overlaid on (a) the hand-drawn map and (b) LILY's map.

Figure 6.6: The circle shows the actual location of the robot for localization test 3. It is overlaid on the localization procedure output.

(i. e., it temporarily turns off the "servo based on translated distance" behavior, instead keeps its heading constant). The net effect is that the steerer traverses an approximate semicircle whose turning radius is determined by the behavior of the pusher/object/steerer system. We ran this test a number of times, varying the object and the allowed contact-modes, performing 5 repetitions at each combination. Table 6.2 summarizes the result for each combination of object and allowed contact mode: The minimum turning radius for a given execution is measured as the distance between the starting position of the object and the finishing position times 1/2 the number of 180-degree turns completed, for those runs which were completed



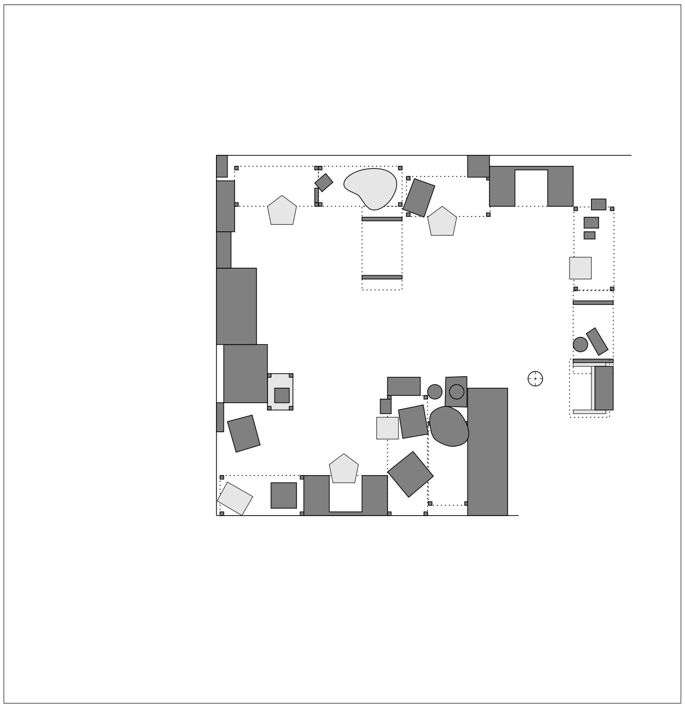
(a)

(b)

Figure 6.7: The circles in these figures show the actual location of the robot for localization test 4. The circles are overlaid on (a) the hand-drawn map and (b) LILY's map.

Figure 6.8: The circle shows the actual location of the robot for localization test 4. It is overlaid on the localization procedure output.

or which terminated when the robots ran out of space. The second entry for each object/contact-mode combination is the fraction of trials which were completed successfully. It should be observed that the number of tests is not really statistically significant; however, each trial takes a few minutes to conduct, and the battery life for the pushing robot is only about 1 1/2 hours, due to the large current drain experienced by the pushing robot. These factors, combined with the fact that we were forced to conduct these experiments in a common area, made it impossible to increase the number and variety of experiments as dramatically as we would have like.



(a)

(b)

Figure 6.9: The circles in these figures show the actual location of the robot for localization test 5. The circles are overlaid on (a) the hand-drawn map and (b) LILY's map.

Figure 6.10: The circle shows the actual location of the robot for localization test 5. It is overlaid on the localization procedure output.

It is somewhat difficult to compare different experimental models of manipulation, especially in the absence of full mechanical analyses: We can compare *our* pusher/steerer protocols to *our* single-robot protocols, but we must bear in mind that these are comparisons between the protocols, *not* between the models – since we have put far more effort into our multi-robot protocols than we have into single-robot manipulation protocols, there is a certain straw-man element to these comparisons. That said, we make the following qualitative observations based on observations we have made in the course of our study of large-scale manipulation: for straight-line translation tasks, the pusher/steerer system translates the specified object in a

Table 6.1: Degrees of arc traversed at given turn radius for several boxes. w is the box dimension between the contact faces; ℓ is the box dimension between the non-contact faces. The values presented are averaged over 5 runs.

	Turning Radius (mm)			
Box ($w \times \ell \times m$)	1000	1500	2000	2500
51cm $\times$ 58cm $\times$ 3Kg	1000	1050	1220	1440
35cm $\times$ 23cm $\times$ 2Kg	225	234	528	618
33cm $\times$ 58cm $\times$ 4Kg	153	342	475	656

straight line with very little variation, independent of the location of the object's center of friction. A single robot protocol we have used for translating objects performs the following steps: (a) determine the relative position of the object's center of friction (normal to the direction of the translation), (b) reposition the robot to push against the detected center of friction, and (c) push until it senses that the object is no longer translating in a straight line (sensed by changes in contact-mode). Since the robots have limited-resolution contact-mode sensing and a slow control loop (the rate at which the SCHEME board can change the command being executed by the wheelbase is on the order of a few hertz, due to the low-bandwidth communication channel we must use to control the wheelbase), this means that the two-robot system tracks the desired translation orders of magnitude better than the single-robot system. For circular arc-following, a similar claim holds: the single-robot system can cause the object to undergo a translation and rotation that put the object in the desired end position, but the trajectory followed by the single-robot system will approximate the desired arc much worse than the two-robot system will.

6.4 Future Experiments

The test results we've obtained with our mapmaking and localization system seem to indicate that our approach to mapmaking and localization (combining a mapmaking system based on the work of Moravec and Elfes with a rasterized implementation of a computational geometry localization algorithm), using a point-and-shoot laser rangefinder, provides a promising avenue for mobile robot navigation. In this thesis, we can show only a few examples of this system in action; understandably, we have selected the ones which best illustrate our claims about the system. We have run

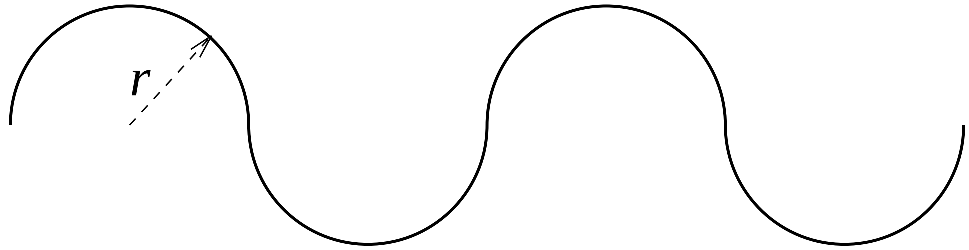


Figure 6.11: The second turning-radius test used a trajectory consisting of four consecutive semicircles: clockwise, counterclockwise, clockwise, counterclockwise.

the mapmaking and localization algorithms many times on LILY, though, and have found them to be consistently reliable.

Our navigation system is, clearly, still in the development stage, and there are many more experiments we would like to perform using it. We would like to extend the implementation of our navigation system to incorporate a number of other features that we have addressed in this thesis. We have not yet been able to test many aspects of our navigation system, including

- **3D**: We feel that the robot's maps would better reflect the environment if it incorporated height information. Approximating a three-dimensional world with a two dimensional map reduces the accuracy of the map and the accuracy that one can expect from the localization algorithm.

- **Time-varying Behavior**: Memory limitations on LILY have prevented us from testing our strategy (described in section 3.2.2) for coping with nonstatic environments.

- **Rotational Localization**: We have not yet tested our rotational localization strategies in concert with our translational localization algorithms.

- **Simultaneous Mapmaking and Localization**: For the examples shown in this chapter, it was the case that we made a map using dead reckoning, then used the localization algorithm to locate the robot after we moved the robot to an unspecified position. We would like to perform experiments in which the robot, for example, makes a map, or a portion thereof, then uses a combination of localization and dead reckoning to build an improved map; for these experiments, the robot would also determine for itself where it needed to explore to improve the quality of its map.

We have used the PUSHER and STEERER2 protocols to perform a large number of manipulation experiments. In addition to the experiments just described, we have

Table 6.2: Turning radius subject to contact-mode restriction for S-curve test. For each combination of box and contact-mode restriction, the test was run 5 times. The first number in each box is the average turning radius; the second is the number of times out of the 5 that the task terminated successfully (either by completing four half circles or by running out of space in our test area).

	Allowed Contact Modes		
Box	3-13	4-12	5-11
51cm $\times$ 58cm $\times$ 3Kg	1.1m/4	1.4m/4	1.5m/5
35cm $\times$ 23cm $\times$ 2Kg	0.8m/2	1.1m/3	1.3m/2

conducted a number of experiments of the form "follow the specified path", where the path is a hand-specified series of up to ten arc segments and line segments. We have observed two main failure modes for this evaluation task: failure on the steerer's part to avoid obstacles, and inability of the pusher to supply sufficient pushing force to push both the object and the steerer. The first failure mode can be overcome by combining the manipulation system with the navigation and localization system, so that the steerer will perform obstacle avoidance (so far these systems have been tested in isolation). While it is clear that to manipulate much larger objects, we would need more powerful robots, the second failure mode can at least be mitigated by using a higher-torque gearing in the robots' wheelbases (all of our robots are capable of higher speed than is necessary for our purposes; by using a lower gearing ratio, we can reduce the top speed but increase the top torque exertable by the robots) and perhaps by improving the control loops we use to simulate translational compliance in the steerer.

In the process of developing the pusher/steerer model, we obtained additional experimental evidence supporting our claims regarding, not only the robustness of the pusher/steerer model, but also the robustness and flexibility of our mobile robots. Two students in our laboratory performed experiments similar to the "follow the specified path" task, but employing Lily in the steering role, rather than the pusher, and Camel (see section 4.3.2.7 and figure 4.13) in the pushing role. They also tested versions of the Role-Reverse protocol. We feel that the success of these experiments, given Camel's differently configured base and contact sensors, speaks well of both the pusher/steerer model and the mobots on which we implemented that model.

6.5 Evaluation

On the basis of the experiments outlined in this chapter, we feel that both our localization algorithms and our manipulation protocols, as well as the mobile robots on which we tested them, performed very well. In addition to the results presented in section 6.2, LILY has built well over a dozen maps, and performed about a hundred localization passes, with varying but generally very good results. We have performed at least 200 experiments testing various aspects of the pusher/steerer system, and have found it to be a flexible set of protocols which, for manipulating objects of appropriate size and mass, are very robust over a wide variety of trajectories.

Chapter 7

Conclusions and Further Research Directions

7.1 Closing Summary

In this thesis we have discussed the major aspects of our research in mobile robotics. Primarily, we discussed algorithms for mobile robot localization, the construction of robust, flexible, easy to use mobot systems, and using cooperating mobile robots to move large-scale objects using a *pusher/steerer model* of cooperative manipulation.

In chapter 2, we defined mobile robot localization as the process of finding *feasible poses*, locations that are consistent with the robot's internal map and with instantaneous range data. We presented algorithms to perform localization in $\mathbb{R}^2$ given a map and a set of range probes; the first of these were exact combinatorial algorithms which produced feasible pose sets from a polygonal map and a set of range probes. We then introduced the concept of *rasterized* algorithms, and presented rasterized algorithms to produce feasible poses when the map is a two dimensional bitmap. Table 7.1 summarizes the time-complexities of our algorithms.

In order to test our localization algorithms and to provide a common framework for localization, mapmaking, and path-planning, we implemented rasterized algorithms to perform each of these functions. We implemented mapmaking functions based on the *statistical occupancy grid* of [Elf87,Mor89,ME85]. Our path-planner uses the C-space floodfill algorithm of [LRDG90]. We discussed these implementations in chapter 3. In addition, we discussed a way to augment the mapmaking algorithm so it keeps track of the time-variability of locations, allowing the robot to detect areas of high change or of high stability. Finally, we described auxiliary functions using our laser rangefinder to detect features such as corners and long

Table 7.1: The time-complexity of our algorithms: For the combinatorial algorithms, n is the number of features in the map, m is the number of range probes, and s is the number of intersections between features, which is $O(m^2n^2)$ worst-case. For the rasterized algorithms, the map has size $n \times n$.

	1 probe	m probes
Exact Combinatorial	$O((n+s)\log n)$	$O((mn^2+s)\log(mn))$
Rasterized	$O(n^2)$	$O(mn^2)$

straight edges in the environment; these functions can serve as parts of a separate mechanism for rotational localization.

We described the design philosophy, the design, and the implementation of the Cornell Mobile Robots in chapter 4. Our design goal was to build mobile robots that would be robust, flexible, and easy to use: we wanted robots that would facilitate our research program by (i) working reliably as much of the time as possible, (ii) allowing easy upgrading and modification (e.g., changing the main processor or adding a new sensor or actuator), and (iii) providing a flexible high-level programming/debugging environment to speed algorithmic development. We achieved our design goal by building our robots with a modular hardware and software architecture that allows us to add new sensors and actuators more quickly and more easily than would be possible with a monolithic architecture. The hardware architecture is essentially a loosely coupled set of independent microcontrollers operating sensors and actuators under the control of a central, more powerful processor. The software architecture allows us to write higher level mobot programs in Mobot-SCHEME, but allows us to write individual functions in C if performance requires. Our robots each contain one or more Cornell Generic Controllers which coordinate control of the robots' sensors and actuators. The GC in each robot runs "SCMODS", a C program which provides a simple interface for adding new sensor and actuator control modules. Evidence that our approach to building mobile robots has been successful is provided by the fact that more than sixty people (including all faculty, graduate and undergraduate researchers, and robotics class students) have programmed tasks on TOMMY, LILY, CAMEL, and ROSEMARY. These tasks have ranged from simple path-planners and obstacle-avoidance routines up to cooperative large-scale manipulation and visually-guided navigation.

We feel that manipulation by mobile robots is an important research area for a number of reasons. Foremost among these is that, for many applications, mobile robots will need the capability to move objects from one location to another. In chapter 5, we described and analyzed a set of protocols to manipulate large-scale objects (boxes, furniture, etc) along complex paths using two mobile robots executing *pusher/steerer* protocols. These on-line protocols that involve no direct communication between the two robots. One robot executes a *pusher* protocol: it doesn't know the path, but it pushes the object forward and concurrently and continually aligns so that it is pushing perpendicular to the object face with which it is in contact. The other robot knows the path. It executes a *steerer* protocol: it remains translationally compliant, but steers as it is pushed along by the pusher. In this way it leads itself, the object, and the pusher along the specified path. We feel that these protocols provide an effective means for transporting an object along a path. Because pinning the object between the two robots greatly reduces the sensitivity of the manipulation protocols to the frictional and mass-distribution characteristics of the object, we conjecture that these protocols effectively allow us to reduce large-scale manipulation to nonholonomic path planning.

Finally, in chapter 6, we presented the results of real experiments performed using LILY and TOMMY. We showed maps made using our laser rangefinder and the results we obtained when we ran our localization algorithms on real sets of range data. In addition, we analyzed the behavior of the pusher/steerer protocols using TOMMY and LILY to manipulate several objects.

7.2 Further Research Directions

The work described in this thesis suggests a number of interesting extensions at both theoretical and implementational levels. One set of extensions to the localization algorithms would be to extend them to $\mathbb{R}^3$ and $\mathbb{R}^3 \times \mathrm{S}^1$. For the rasterized algorithms, this would primarily be an implementational extension, but for the exact algorithms there would be nontrivial complexity analysis required to obtain time bounds for localization in these higher dimensions. From a practical standpoint, it would be more interesting to determine whether $\mathbb{R}^3$ localization using $\mathbb{R}^3$ maps would overcome the difficulties encountered when we use $\mathbb{R}^2$ localization on an $\mathbb{R}^2$ map to find the robot's pose in an $\mathbb{R}^3$ world. Another promising direction would be to further pursue active sensing issues in localization, including (i), which range probes are most useful in localizing the robot and (ii), if we cannot fully disambiguate the robot's current pose using instantaneous range data obtained from the robot's current pose, what is the best motion the robot can make to obtain data that will give it a more unique pose estimate?

Our cooperative manipulation protocols also open up a number of research avenues. One near-term goal would be to integrate the pusher/steerer protocols with a nonholonomic path-planner (such as described in [BL93] and an obstacle avoiding path execution system. We conjecture that such an integrated system, particularly if provided with a robust navigation and localization system, would provide a practical solution for many large-scale manipulation tasks. It would be desirable to further develop the protocols for using "back-and-fill" to manipulate objects through tighter arcs and narrower channels than possible with one-step pusher/steerer operations. We would also like to have more detailed analysis of the behavior of a back-and-fill pusher/steerer system. Huttenlocher, et. al ([HLR94a,HLR94b]) discuss the use of their Minimum Hausdorrf Distance pattern-matching system to enable a mobile robot to home in on a particular landmark in its environment; one could use their system to allow the pusher to detect the position of the steerer, making it possible for the pusher to servo its heading toward the steerer, rather than toward the object. We would also like to examine the effect that providing limited communication between the pusher and the steerer would have on our protocols and on the tasks we could accomplish using them. We are interested in investigating how one might go about using more than two robots in a pusher/steerer system: what are the effects of adding extra pushers? What are the effects of adding extra steerers? Do the pushers need to communicate among themselves? What about the steerers? One method which we might consider using to answer these questions would be a formal analysis tool such as the information invariants suggested by Donald and Jennings in [Jen95,Don95,DJR94b].

7.3 Conclusion

Much work has been done on mobile robot navigation problems. What we have done is to provide exact combinatorial algorithms with complexity analyses for the localization aspect of navigation and to provide rasterized versions of these algorithms which are suitable for execution upon a mobile robot. Our work in mobile robot design has provided instances of relatively inexpensive mobile robots which, nevertheless, have the flexibility and ease of use to permit robotic researchers, both novice and expert, to perform many useful experiments and to develop useful algorithms and protocols quickly and easily. These robots, in addition, display unusually high reliability: they are in a working state almost all of the time, and can be quickly and easily repaired on those occasions when they do break down. Finally, our work in cooperative mobile robotic manipulation provides protocols which can be used to do real, practical large-scale manipulation with multiple robots, even when those robots have fairly primitive sensing and no communication. These protocols are among a very few that have been implemented on real mobile robots, that perform tasks in cooperation that cannot be accomplished by equivalent single robots, and that have received formal analysis.

Appendix A

The Cornell Generic Controller

A.1 Introduction

The Cornell Generic Controller 3.0 (GC3, figure A.1) is a small, versatile, 80C196KB based single-board microcontroller card designed at Cornell in the CS Robotics and Vision Laboratory for use in our mobile robots. We designed the Generic Controller as we designed our robots: to be robust, flexible, and easy to use. Capabilities of the GC3 include:

- **Versatility**: The GC3 has sufficient computational speed to allow it to perform numerous sensor-control tasks at a high servo rate.
- **Serial Communication**: Three serial ports allow connection to a debug/ console port and two other serial devices.
- **Easy Expansion**: The GC-bus carries address, data, interrupt, and clocking signals off-board, so that it is easy to interface devices on other boards to the GC's processor.
- **Simplicity**: low chip count, low external-component count, and a two-layered PC board make construction and testing of a GC3 a simple matter. In addition, the GC3 requires only a single voltage supply (5.0V), simplifying power supply requirements.
- **Small Size**: The GC3 occupies a single Eurocard card slot (100mm wide by about 170mm long).

We have used the GC3 in four robots of various types and have several in use for sensor, actuator, and software prototyping purposes. This document describes the GC3 in detail and provides extensive background on the design and use of the GC.

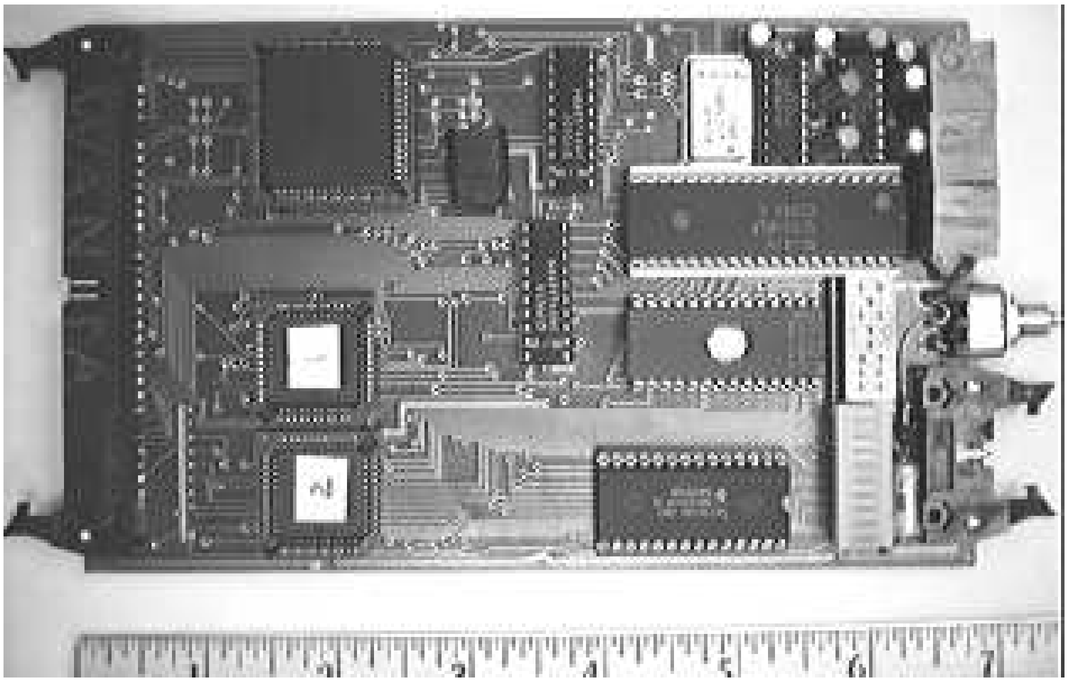


Figure A.1: The Cornell Generic Controller 3.0

We provide instruction on how to interface an external card to the GC using the GC-bus. Finally, we analyze some of the shortcomings of the GC (some of which are due to the age of the design) and suggest how the design of a version 4.0 GC might improve upon version 3.0. A schematic for the version 3.0 GC appears at the end of this appendix.

A.2 Design Goals and Constraints

Our main goal in designing the Generic Controller was to produce a microcontroller-based single-board computer that would support our still-developing mobile robotics program. Since our robots were designed as, and continue to be, development platforms for various sorts of mobile robotics research, we wanted our controllers to be flexible, expandable, and robust. Our original list of design goals is similar to the list of characteristics just listed for the GC 3.0:

- **Computationally Versatile**: We needed a microcontroller with sufficient speed and memory to support a variety of activities.
- **Expandable**: We wanted the ability to add new peripherals and hardware capabilities to the GC without ever having to modify the base GC. We also wanted open-ended expansion, so that we could add as many capabilities to

each GC in a given robot as that GC's processor could service in a timely fashion.

- **Easy Communication**: Knowing that GCs would need to communicate with their robot's main computational engine, certain sensors and actuators, and other GCs by RS-232 serial, we wanted the GC to provide several serial channels on the base unit.

- **Low Power**: Within the limits of readily available and readily employable technology, we wanted to keep the power consumption of the GC as low as possible.

We had several constraints on the GC's design. Some constraints stemmed from our robots' physical requirements and some from the physical and electrical characteristics of the available technology. Other constraints come from our need to have a reasonably easy development cycle: Since the controllers were being designed, built, and programmed by students, we needed good hardware and software debugging tools. Some of the constraints with which we had to deal were the following:

- **Form Factor**: The modular enclosure on our mobots has a Eurocard form-factor (100mm by about 170mm) card-cage.

- **5-volt Only**: The enclosure provides regulated 5-volt power, making it preferable for our designs to use 5-volt only.

- **In-Circuit Emulators**: To facilitate hardware and software debugging, we wanted a processor for which there were readily-available in-circuit emulators with good debugging software.

- **Appropriate Interconnection**: We needed to equip the GC with a large number of connectors so it could interface with the SCHEME-running board and all of its peripherals. We needed these connections to be solid and reliable, but easy to install and remove. We needed connectors small enough that we could put them all on the ends of the board, for easy accessibility, but we needed parts and cables that were readily available and easy to install.

- **Economy**: Since we anticipated using large numbers of GCs, we wanted to use inexpensive technology where possible.

A.3 Controller Overview

This section gives a brief rundown on the capabilities on the Generic Controller, concentrating on larger issues of capacity and user interface. The two sections following this one present more detailed information on how the GC is put together and how peripheral devices can be used with the GC. The organization of the GC is shown in figure A.2 and can be described as follows: The Intel 80C196KB processor is hooked to on-board SRAM and EPROM, as well as to the onboard SCN2681A dual UART, LEDs, DIP switches, and bus connector by two PLDs and two HCMOS buffers which serve the purposes of demultiplexing and buffering signals generated by the processor and controlling the memory-mapped I/O page. Also included on the board are reset and clock-generation circuitry and MAX232 line/logic level converters which convert serial transmissions from logic to RS232 line levels and convert serial receptions from line to logic levels. In order to allow for easy addition of new sensor and actuator controllers, we have provided the GC-Bus, which carries address and data lines and control and interrupt signals to and from the GC's daughter-boards.

Table A.1 shows the basic memory map of the GC3. The 80C196KB register file consists of 24 special purpose registers governing I/O, interrupt, and timing control, and 232 general purpose registers addressible as bytes, words, or longwords. The I/O page is 256 memory locations which are reserved for memory-mapped I/O. Some of those locations are taken up by the on-board DUART, DIP switches, and LEDs, but most are left for use by off-board peripheral devices connected to the GC by the GC-Bus.

Serial communication with the GC is provided by three serial ports; one built-in to the microcontroller, and two contained in the SC2681 DUART. Serial lines are attached to the GC using RJ-11 connectors (the cable should be a 4-conductor modular cable and should have a 6-4 RJ-11 plug on the GC end).

Configuration and diagnostic information are provided by the on-board DIP switches and LEDs. The switches are invaluable for configuration purposes; typical uses for these switches include enable/disable for debugging printout to the serial ports and for various software/hardware modules and select for default serial port baud rates and for power-up mode (for example, whether the GC executes burned-in user level software on reset, or the built-in debug monitor). The LEDs provide extremely useful low-level debugging output (we typically use one for a monitor heartbeat and several for serial activity indicators).

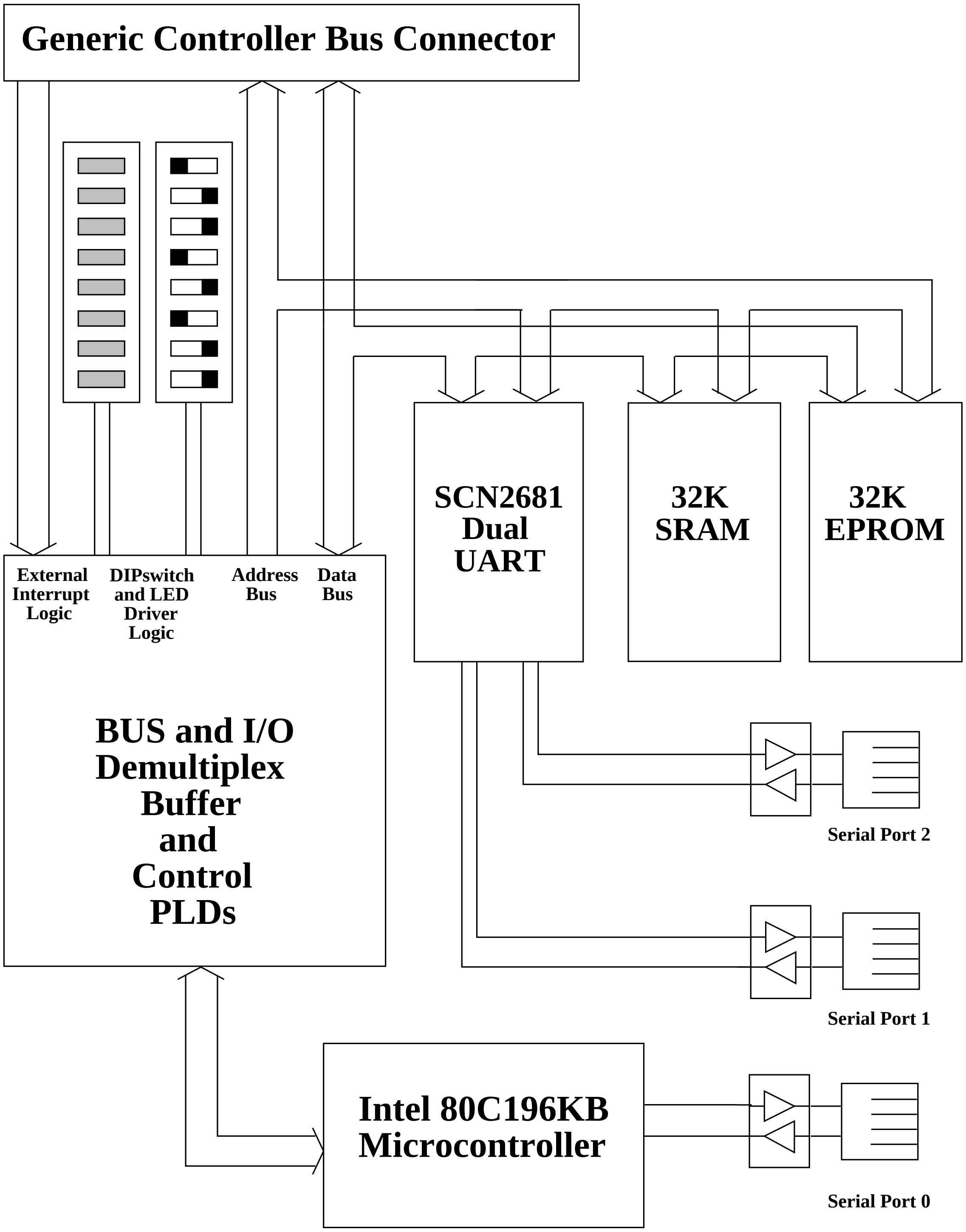


Figure A.2: A Block Diagram of the Generic Controller 3.0

Table A.1: The Generic Controller 3.0 Memory Map

Generic Controller 3.0 Memory Map		
Start	End	Mapping
0x0000	0x00ff	80C196KB register file
0x0100	0x7fff	31.75K EPROM
0x8000	0x80ff	GC I/O Page
0x8100	0xffff	31.75K SRAM

In order to allow asynchronous behavior on the part of peripheral devices, the Generic Controller provides 8 active-low external interrupts. Since the 80C196KB only provides one external interrupt, one of the PLDs provides an 8-way AND of the incoming interrupt signals to the EXINT pin on the processor. That same PLD provides a memory-mapped register which contains the eight incoming interrupt signals, so that an interrupt service routine can determine which of the incoming interrupt signals caused EXINT to be asserted; an external interrupt vector table can be used to provide separate interrupt vectors for each device which is capable of causing an external interrupt. One of the eight external interrupts (number 0) is dedicated to the on-board DUART, but the other seven are routed to the GC-bus, so that they are available to peripheral circuits on other boards.

A.4 Design Details

We are using the 80C196KB microcontroller in the 8-bit data bus, 16-bit address bus mode. The motivation for using 8-bit data mode (instead of the provided 16-bit mode) is that we can get by with a single 8-bit RAM and a single 8-bit EPROM. Given our space and cost requirements, the benefit of the simple memory structure outweighs any performance hit we might take. The processor is running with a 12MHz crystal (which it divides into a 6MHz clock) and also has access to a 2.4565MHz clock (through T2CLK) which is used to clock the built-in UART.

The remainder of the serial I/O subsystem is provided by the SCN2681A dual UART and two MAX232CPE serial driver chips. The SCN2681A has several advantages for a multiple serial port design. Primary among these is that it has two ports whose baud rates are independently software programmable. One minor oddity about our use of this chip on the GC3 involves the input clock to this chip. The SCN2681A is designed to generate its baud clock from a 3.6864MHz clock oscillator; its preprogrammed serial rates, which are selectable by writing a single byte for each channel, are based on that clock rate. In order to simultaneously accomodate the desires to (i) generate the appropriate baud rates on the DUART, (ii) generate the appropriate baud rates within acceptable error tolerances on the microcontroller, and (iii) include only one clock oscillator on the GC, we decided to use a single clock oscillator which would meet (i) and (ii). The 80C196KB requires that its T2CLK clock have a frequency below 3MHz; this means we can't use the 3.6864MHz clock but can use anything below 3MHz. 2.4576MHz is easily divided into the standard serial rates ($2457600 = 300 * 8192$), so we can use it with the microcontroller and also with the DUART, provided we use the DUART's built-in programmable timer to generate the baud clock. This is only a limitation if we wish to run the DUART's two ports at different rates. Each of the three serial ports on the GC is interfaced to the outside world through a MAX232CPE logic to line-level adaptor. The MAX232 was the adaptor of choice due to the fact that it requires only the already available +5 volt supply, and also due to its easy availability. The line-level side of each MAX send/receive channel is connected to an RJ11 socket, allowing for serial connection using a small, convenient cable and a locking connection which is easily installed and removed.

PLD1 (figure A.3) performs three functions. Eight of its inputs are AD0-AD7, which are the data bus multiplexed with the low eight bits of the address bus. Clocking on the microprocessor's ALE (Address Latch Enable) signal, PLD1 demultiplexes out the low eight bits of the address bus and provides these as outputs. In addition, it uses all sixteen bits of the address bus to compute the enable signals for the RAM, the EPROM, the on-board DUART, the LEDs, the DIP switches, and the external interrupt register, as well as the IOPAGE signal which indicates that an address which lies in the I/O page has been placed on the bus, and which is present on the GC bus. Table A.2 shows the exact mapping used. Finally, the reset switch circuit output, $\mathrm{RESET}_{\mathrm{in}}$ is hooked into PLD1, where a digital inversion and an external RC network provide the small amount of hysteresis which the 80C196KB reset circuit requires for reliable operation. The singly inverted $\overline{\mathrm{RESET}}$ signal is connected to the processor's reset input, while the doubly inverted (uncomplemented) RESET signal is provided for those chips (primarily the on-board DUART) which require an active-high reset.

PLD2 implements the microcontroller interface to the DIP switches and the LEDs and enables the microcontroller to accept external interrupts, as well as to determine which external interrupt has fired. As can be seen from figure A.4, the structure of the PLD internal structure consists primarily of an octal latch to drive

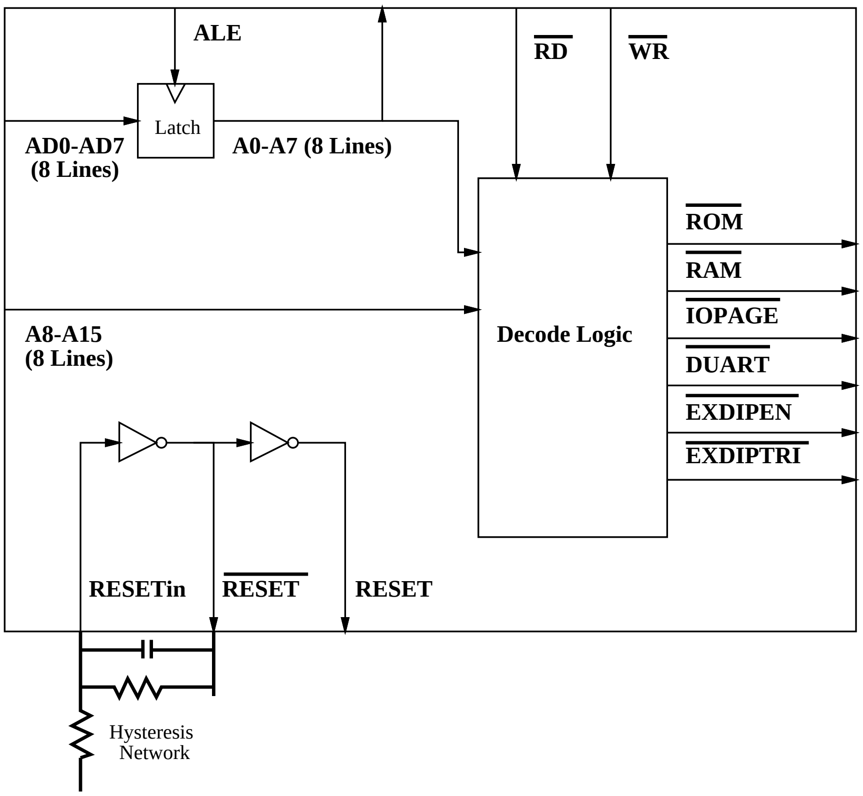


Figure A.3: PLD1 demultiplexes the address/data bus and generates select signals for memory and I/O devices.

Table A.2: The Memory Map for the I/O page of the GC3.0

Generic Controller 3.0 I/O Page Memory Map		
Mapping	Start	End
IOPAGE	`0x8000`	`0x80ff`
DUART	`0x8000`	`0x800f`
LEDs	`0x80d0`	`0x80df`
DIPsw	`0x80e0`	`0x80ef`
EXINTn	`0x80f0`	`0x80ff`

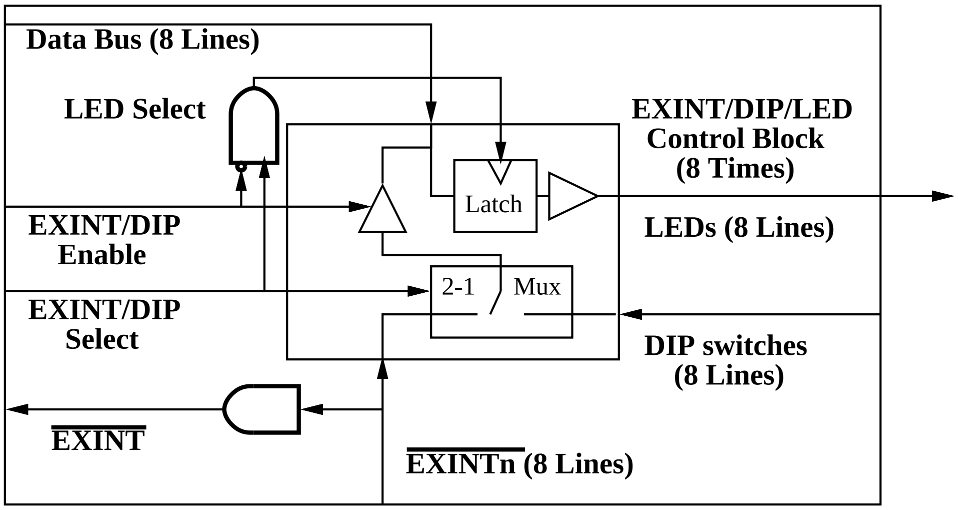


Figure A.4: PLD2 interfaces the DIP switches, LEDs, and external interrupts to the microcontroller.

---

the LEDS (driven by the data bus and clocked by an LED select signal) and an octal multiplexor/tri-state buffer combination which allow the microprocessor to read the DIP switches and the external interrupt register. The EXINT output of PLD2 is the AND of the eight external interrupt pins – if any of these active-low signals is asserted, the EXINT output asserts, causing an interrupt to the microcontroller through its EXINT input. Also, due to a limitation in the number of pins on the PLD, the LED enable is multiplexed with the enable/select lines for the DIP switches and interrupt register.

The GC3 has two other support chips (74HC245s, in this case, which are bidirectional octal tri-state buffers) which are hooked into the microcontroller's bus system. One of these buffers is configured always to be enabled and to input from the microcontroller side and output to the bus side, and is used simply to provide signal buffering for the high eight bits (A8-A15) of the address bus. The other buffer is also always enabled, but switches direction based on the state of $\overline{\text{RD}}$; this buffers the controller data lines from the system data bus. There should also be provisions for buffering control signals from the processor (such as $\overline{\text{RD}}$ and $\overline{\text{WR}}$), but these were not made on the 3.0 version of the Generic Controller.

A.5 Adding a Peripheral to the Generic Controller

Connecting a chip to the 80C196KB from a daughterboard is performed in almost exactly the same way as it would be if the chip were part of the main circuitry of a custom 80C196KB circuitboard. Address lines are connected to the chip, as are data lines, and as are any needed control lines. An address is selected for the new peripheral chip, and decoding logic implements this choice using the address lines. The difference is that the daughterboard can be designed, laid out, and built as a separate unit, in virtual isolation from the rest of the circuit. This facilitates debugging and testing, and allows for easy attachment and removal of various daughterboards; it also eliminates the need ever to cram additional circuitry into an already overcomplex or overcrowded circuitcard.

Of course the daughterboard *doesn't* exist in total isolation from the rest of the circuit in which it resides. Care must be taken that various elements of various daughterboards don't interfere with one another, and the loading factor on the various control signals must be considered. The designer takes care of the former problem at a systemic level by being careful to assign the memory-mapped chips on a particular daughterboard to a portion of the I/O page that is unused in the particular system in which the board will be used; using programmable logic that compares the address on the bus to an address assigned by dipswitch or jumper assignment allows for near-instantaneous change of address. The latter problem must also be dealt with at a systemic level, but a lot can be done to minimize the impact of each peripheral card on the overall system. Primarily, the designer should incorporate buffers on address, data, and control lines used by a given peripheral card, so that at most one "load" is added to each such line by the entire card. Thus, if there are three chips on a given card that reside in the I/O page, any address lines they share should go through a buffer, and the data lines should pass through a single bidirectional buffer so that only one load is added to the data lines on the GC-Bus, rather than one load per chip. The decoding logic on the card should ensure that the data buffer is only allowed to drive the GC-Bus data lines when a chip on that card is selected for reading.

A.5.1 A GC Peripheral Case Study: LILY's Pan/Tilt Head Controller

PANTILT.SCH1 - Wed Jul 28 13:49:28 1993

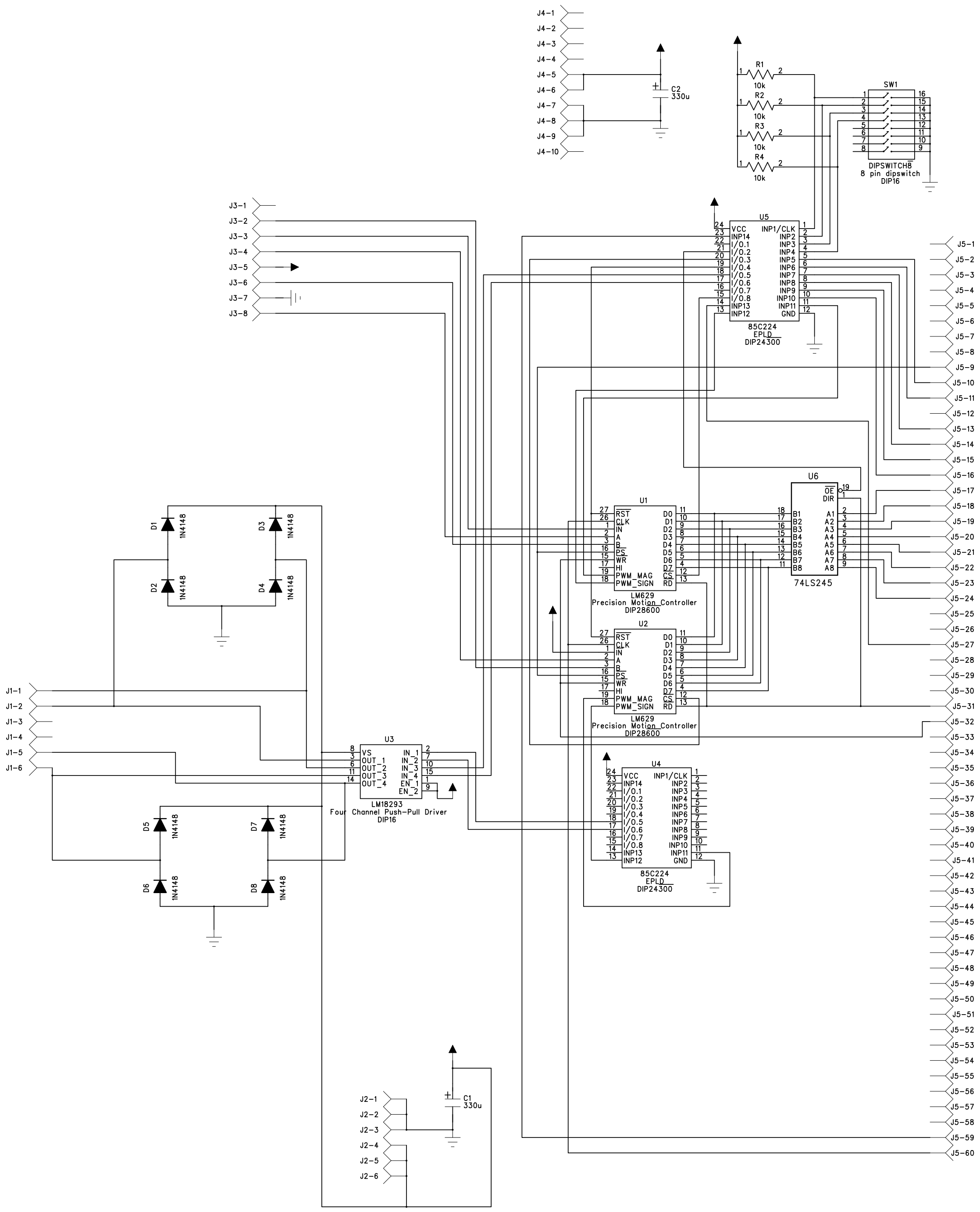


Figure A.5: The schematic for LILY's pan/tilt head controller card

To illustrate the process of designing a new GC-Bus peripheral, we will analyze in detail one of the peripheral cards we use on LILY. Figure A.5 shows the schematic for the controller card which runs LILY's pan/tilt head (the head itself is described in chapter 4, in section 4.3.2.3). The design centers around two National Semiconductor LM629 motor control ICs, each of which provides PID control[1] for one of the pan/tilt head's two motors. The motor controllers provide pulse-width-modulated (PWM)[2] outputs for the motors. The two Intel 85C224 programmable logic devices translate these PWM outputs into a format which is suitable for the National Semiconductor LM18293 motor driver. The LM18293 is a four channel push-pull driver circuit which can be configured to provide bidirectional control for two DC motors. The outputs of the LM18293 go directly to the motors' poles. The motors are equipped with incremental shaft encoders which transmit pulses back to the motor controllers, allowing the motor controllers to keep track of the motor's positions. In addition, the pan stage is equipped with a micro-switch which triggers whenever the pan stage rotates past a particular heading. This output is provided to the pan motor's controller, so that the pan stage can be homed (set to point exactly forward on the robot) automatically. The two sets of flyback diodes on the motor driver's outputs and the two large (330 μF) capacitors across the board's power inputs help prevent current surges at the motors from causing errors at the motor controllers or at the Generic Controller.

The GC-Bus interface portion of the pan/tilt controller consists of the GC-Bus connector, the address-selector DIP-switches, one of the 85C224 PLDs, and a 74LS245 which serves to buffer the GC's data lines. The GC-Bus connector connects the $\overline{\mathrm{RD}}$ and $\overline{\mathrm{WR}}$ inputs of the two motor controllers directly to the $\overline{\mathrm{RD}}$ and $\overline{\mathrm{WR}}$ signals generated by the 80C196KB on the GC. $\overline{\mathrm{RD}}$ is also connected to the direction input on the data-bus buffer: this input determines whether data flows from the GC to the motor controller or vice-versa. The GC-Bus also directly provides the processor's output clock and RESET signals to the motor controllers. The bidirectional buffer exists to reduce the maximum current drain which the data inputs of the motor controllers can place on the GC's data lines: the current drain on the GC's data lines due to ICs on this board is one TTL load per data line, independent of the input current of the motor controllers. The four DIP-switches

[1]Proportional-Integral-Derivative control is a very common modes of automatic plant-control. In-depth analysis of PID control can be found in any controls textbook (for example, see [Row86]).

[2]PWM is a common method for digitally controlling motors. A PWM controller provides two outputs, a sign and a magnitude. The sign signal is 1 if the motor should go forward, 0 otherwise. The magnitude signal is a square wave which takes on the value 1 for a fraction of its duty cycle which is proportional to how hard the motor should be driven.

on the controller card allow the user to specify where in the GC's I/O page the motor controllers' registers reside. The motor controllers get their chip-select signals from the PLD to which the DIP-switches are connected. Each motor controller is selected when the IOPAGE signal from the GC is asserted and A4-A7 of the GC's address bus match the DIP-switch settings and A0-A3 of the GC's address bus match the values programmed into the PLD for that motor controller. The output enable signal on the data-bus buffer is asserted when either of the motor controllers' chip selects is asserted.

The key feature to note about the design of this card is that the connection between the motor control and driver electronics and the Generic Controller is very clean and simple; we have far more isolation than would ordinarily be possible in a single monolithic design and can easily move the pan/tilt controller to the bench, if necessary, without disabling the rest of the robot. In general, we can attach any circuit to the Generic Controller via the GC-Bus that could have been installed directly on the GC; furthermore, by buffering the address, data, and control lines coming onto each peripheral card, we can attach large numbers of ICs (and the sensors and actuators they control) to the GC's processor without sacrificing sound electronic design.

A.6 Programming the GC: CGCM and SCMODS

The Generic Controllers in our robots have to interface with SCHEME and with an arbitrary number of sensors and actuators. In many cases, the GC is only a message router: it takes messages from SCHEME, determines where they need to go, and sends them out the appropriate serial port. It receives messages from its serial ports, annotates them with their source, and sends them back to SCHEME. In other cases, the GC is directly controlling the sensor or actuator. In these cases, the GC parses commands from SCHEME and sends commands to the appropriate subsystem. It also determines when the requested action is complete, and composes an appropriate response, which it transmits to SCHEME. The program which has evolved to perform this duty is called SCMODS, for (S)ensor (C)ommand (MOD)ular (S)oftware. SCMODS initializes all of the modules, then commences a servo loop within which it (a) checks for incoming messages from SCHEME and initiates actions called for in those messages, and (b) polls its various subsystems for data and messages headed for SCHEME, packaging those messages appropriately and transmitting them to SCHEME.

```
int Handle_Module_Commands(buf,port,prefix)
char *buf;
int port;
char prefix;
{   if (prefix == IR_PREFIX)
        handle_ir_command(buf,IR_PFX,IN_PORT);
    else if (prefix == DEBUG_PFX)
        handle_debug_command(buf,DEBUG_PFX,IN_PORT);
    else if ((prefix == RANGER_PFX) && (pantilt_enabled))
        handle_ranger_command(buf,RANGER_PFX,RANGER_PORT,IN_PORT);
    else if ((prefix == PANTILT_PFX) && (pantilt_enabled))
        handle_pantilt_command(buf,PANTILT_PFX,PANTILT_ASYNC_PFX,
                                   IN_PORT);
    else if (prefix == BASE_PFX)
        handle_base_command(buf,BASE_PFX,BASE_ASYNC_PFX,
                                 BASE_PORT,IN_PORT);
    else {  send_message(IN_PORT, DEBUG_PFX, BAD_PFX, CRLF_ASTER);
            return(FALSE); }
    return(TRUE);
}
Init_Modules() /* INIT stuff goes here */
{   serial_set_speed(BASE_PORT,0);  /* set base speed to 19200 */
    ir_init();  /* initialize IR system */
    bumper_state = 0; /* init bumper state to 0 (nothing pressed) */
    init_button_state(); /* initialize buttons and leds. */
    init_led_state();
    modem_init(); /* initialize modem protocol code (if any) */
    if (pantilt_enabled) {   /* initialize pantilt head */
        fpinit(); /* VERY important FP INITIALIZATION CALL */
        pantilt_init(PANTILT_PFX,PANTILT_ASYNC_PFX,IN_PORT); }
}
Handle_Asynchronous_Module_Stuff() /* Asynchronous stuff goes here */
{   check_base(IN_PORT,BASE_PFX,BASE_ASYNC_PFX,BASE_PORT);
    check_bumper(IN_PORT,BUMPER_PFX);
    check_buttons(IN_PORT,BUTTON_PFX,BUTTON_ACTIVE_HIGH);
    /*  if the pantilt head is there, how's it doing? */
    if (pantilt_enabled) {
        check_pantilt(IN_PORT,PANTILT_ASYNC_PFX);
        check_ranger(IN_PORT,RANGER_PFX,RANGER_PORT); }
    /* Fetch incoming Modem stuff (if any) */
    check_modem(MODEM_PORT,RECEIVE_PFX,IN_PORT);
}
```

Figure A.6: Some of the routines at the heart of LILY's SCMODS.

```
#define BUMPER_PFX 'M'
#define BUMPER_ADDRESS 0x80c7

int bumper_state;

/* check_bumper sees if the bumper status has changed since the
   last time we looked. */
void check_bumper(inport,BUMPER_PFX)
int inport;
{       int curbump;
        char hexstring[8];

        curbump = read_bumper();
        if (curbump != bumper_state)
        {       bumper_state = curbump;
                dec_to_hex(curbump,hexstring);
                send_message(inport,BUMPER_PFX,hexstring,CRLF_ASTER);
        }
}
/* read_bumper returns the current contents of port 0, which is
   attached to the 8 bumper switches. */
int read_bumper()
{       unsigned char *cp,data;
        int bumpval;

        cp = (unsigned char *) BUMPER_ADDRESS;
        bumpval = (unsigned) *cp;
        return(bumpval);
}
```

Figure A.7: An example SCMODS module: bumper.c.

A new SCMODS module consists of three main parts: an initialization routine, a command/outgoing-message handler routine, and an asynchronous return-message handler.When SCMODS is invoked (at cold-boot time or after a reset), it initializes its own state, then invokes the initialization routine for each module; these routines typically initialize variables pertinent to the module, including I/O page registers if the module services a GC-Bus peripheral, and send any necessary messages to the sensor or actuator involved. SCMODS then starts an endless servo loop. Each time through the loop, it looks to see if it has any messages from Scheme. If there are any messages, it looks at the header on the message and determines which module the message is intended for. It invokes the outgoing message handler for that module, which may simply strip off the header and send the remainder of the message to the module, or which may invoke a routine to parse that message and issue commands to a GC-Bus peripheral. Finally, the servo loop looks at each peripheral to determine whether it needs to be serviced. If the peripheral is attached to the GC by a serial line, this means checking that line's input buffer; if the peripheral is on the GC-Bus, it means checking its status registers. In the serial case, the incoming message is received and a header is added to the message, specifying where it came from. The message is then routed to SCHEME. In the bus case, a string is built up which contains a header and the appropriate information from the module. That string is then routed to SCHEME. A new module can be added to the system simply by creating the necessary subroutines and invoking within in the SCMODS main routines. If a single generic controller becomes unable to handle the load of servicing all of its modules in a timely fashion, its load can be split across two generic controllers by moving some of the hardware to a new GC, programming each with a SCMODS for the modules it now controls, and chaining them so that the first GC passes commands aimed at one of the second GC's modules along to that GC over a serial line. Return messages from the second GC are piped straight through the first GC to SCHEME. Figure A.6 shows the main subroutines from a simplified version of the SCMODS running on LILY. Figure A.7 shows the module code for one of the subsystems (in this case, LILY's bumper).

A.7 Future Generations of the Generic Controller

Version 3.0 of the Generic Controller serves its purpose very well. There are, however, several things about version 3.0 that could be improved for future generations of the GC. These changes can be divided up into two main categories: some changes are relatively minor and represent lessons learned while using the GC3. These can be thought of as the desired changes for version 3.1. Other changes, those for a version 4.0, are more major, and would be undertaken only as part of a complete redesign/updating of our robots.

The changes we would make in a version 3.1 GC are improvements designed to make the GC easier to use and more efficient without changing its main structure. First among these, of course, would be to correct the handful of incorrect/missing connections on the version 3.0 PC-board. A more important change, as regards the GC's usability and robustness, would be to put buffers on all address, data, and control bus signals going off of the GC on the GC-bus. A desirable arrangement would be for address and outgoing control signals to pass through a buffer so that the address and bus lines going to ICs on the GC would not experience any additional load from GC-bus peripherals. Since the data bus is bidirectional, it is more difficult to buffer in this fashion. One possible scheme, though, would be to have an open-collector enable signal on the buffer for the data bus, which would allow GC-bus peripheral cards to enable transmission from the GC-bus to the on-board data bus without specific intervention from on-board the GC. With open-collector circuitry, it is possible to configure a signal to be "*wired-or*": The signal is a function of several outputs, and is ordinarily unasserted. The signal becomes asserted when one or more of the outputs driving it is asserted. With a data buffer enabled by an open-collector driver configuration, then, an arbitrary number of peripheral cards can be connected to the GC, and the buffer enabled only at the appropriate times, without the GC having to determine when that buffer needs to be enabled. Another version 3.1 improvement is the result of lessons learned from our own usage patterns. We were very careful on the 3.0 GC to provide the built-in I/O ports on the GC-Bus. After designing a dozen or more peripheral cards, we have found that we very rarely use those ports, simply because these ports are not generic enough (there's only one of each set of ports, and only one card on the GC can use them). For this reason, it would be an economical move to eliminate those signals from the GC-Bus, or supply them on a separate connector. That would reduce the cost of the GC and of its peripherals and, more importantly, reduce the amount of cable inside the robots' enclosures. Finally, the available electronic technology has improved considerably since we designed the GC3.0; we could improve the efficiency of the GC by using some of the newer electronics available. Of particular interest are components which would make the GC use less power. One example of this would be the "power supervisor" chips which have been developed for notebook computers. These chips detect and signal low-voltage conditions and provide properly conditioned RESET signals to the processor. In addition, newer equivalents to some of the components on the GC3.0 are available which perform the same functions, but using less power.

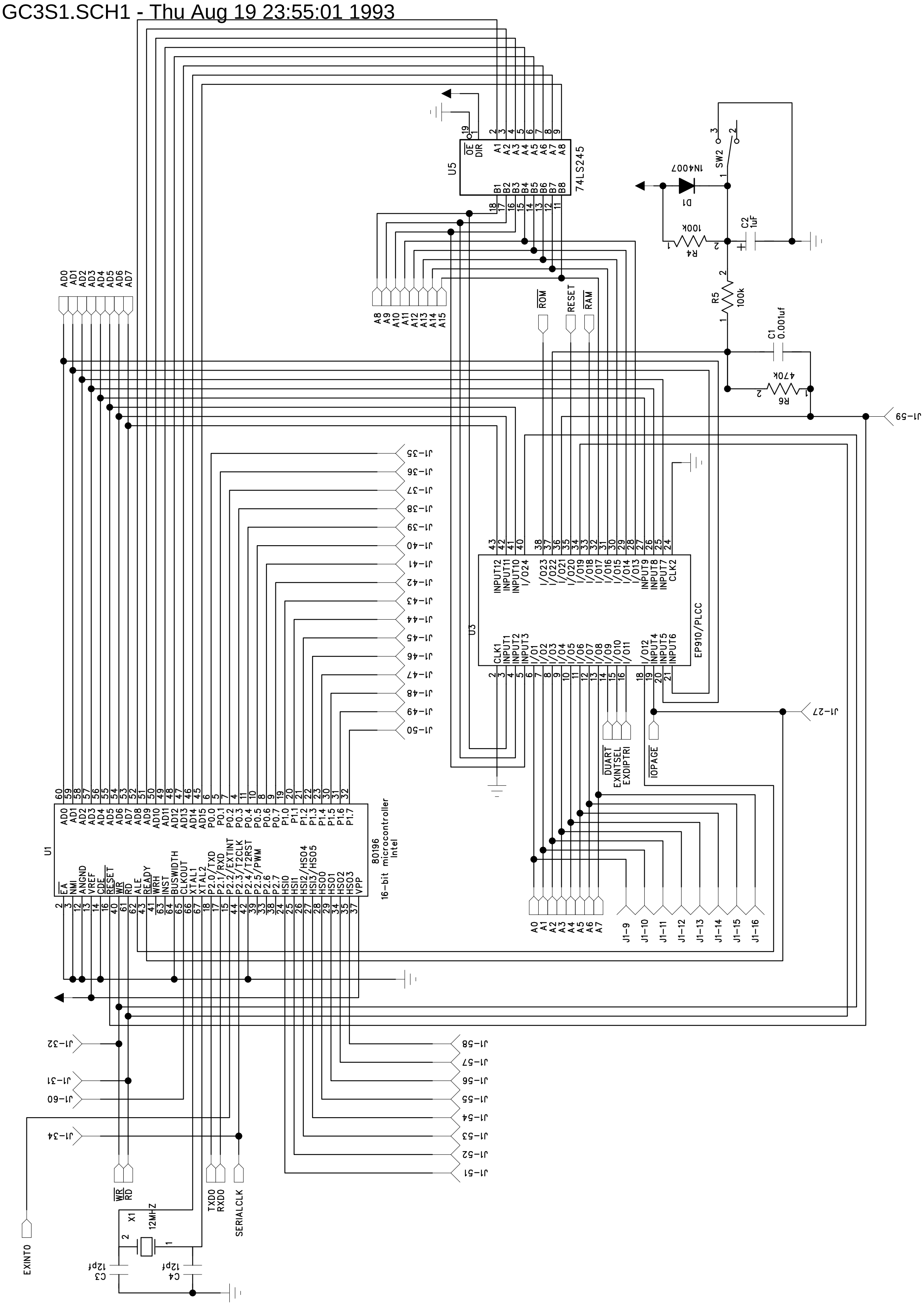
A version 4.0 Generic Controller would be a completely new design which, while it would reflect the same design intent as the GC3.0, would also incorporate technological improvements and lessons learned from our work with the GC3. If we were to start designing the GC4.0 today, we would probably switch to a more powerful processor. The Motorola 68332 and 68340 are 68020-compatible microcontrollers which incorporate many of the features offered by the 80C196KB, but also many features which formerly required additional support chips. One of these processors would be a good choice for a next-generation GC. The other main avenue for improvements to a new GC design is that of communications: In the last few years, several technological developments have been made in the area of high-speed serial communications and protocols. The I^2C high-speed synchronous serial standard allows for serial communication at rates of several hundred kbps, using simple enough circuitry that it can be used for same-circuit interconnection: many components such as serial EEPROMS (electrically eraseable/reprogrammable nonvolatile memory) and peripheral drivers are available which use I^2C to communicate with their host processors. Using this technology, we could eliminate many of the cables found in our current robots and make the system even more robust, with little performance penalty.[3] Another advantage of serial EEPROM is that it allows configuration parameters to be semipermanently modified in software (currently these parameters must be burned into the EPROM, reset by SCHEME whenever the GC is reset, or set by hand at the DIP switches). Other technological improvements may suggest themselves when we undertake the design of the GC4, but our main aim is to identify ways in which we can make a new Generic Controller more powerful, flexible, and robust than the current version, while continuing to simplify hardware interfaces between the GC and its peripherals, sensors, and actuators.

A.8 Summary

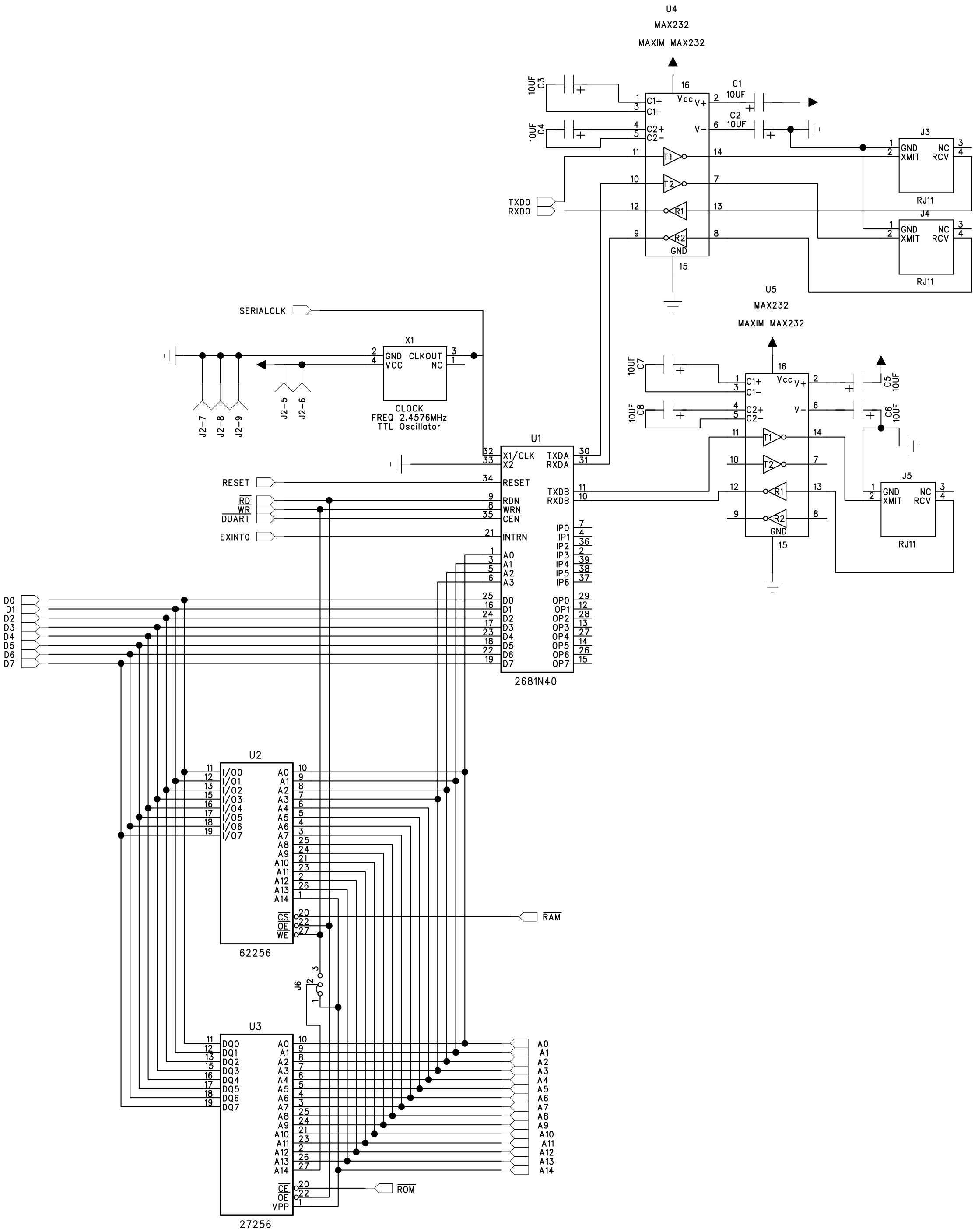
We designed the Cornell Generic Controller to allow easy interface to and control of robotic sensors and actuators. Our main design criteria were versatility and robustness. We met these requirements by designing the GC with a simple, passive-bus connector allowing for easy addition of memory-mapped peripherals, by providing the GC with several serial I/O lines, and by designing it to be small, simple, and power-efficient. We have had great success using the GC: we have GCs in our four mobile robots and in numerous bench-mounted development systems. In addition, we have made use of the basic hardware and software architecture of the GC in several special-purpose microcontroller systems.

[3] We can slow down some of the low-level communication channels without sacrificing overall performance, since the low-level channels (particularly those using the GC-bus) are currently much faster than communication with the SCHEME board.

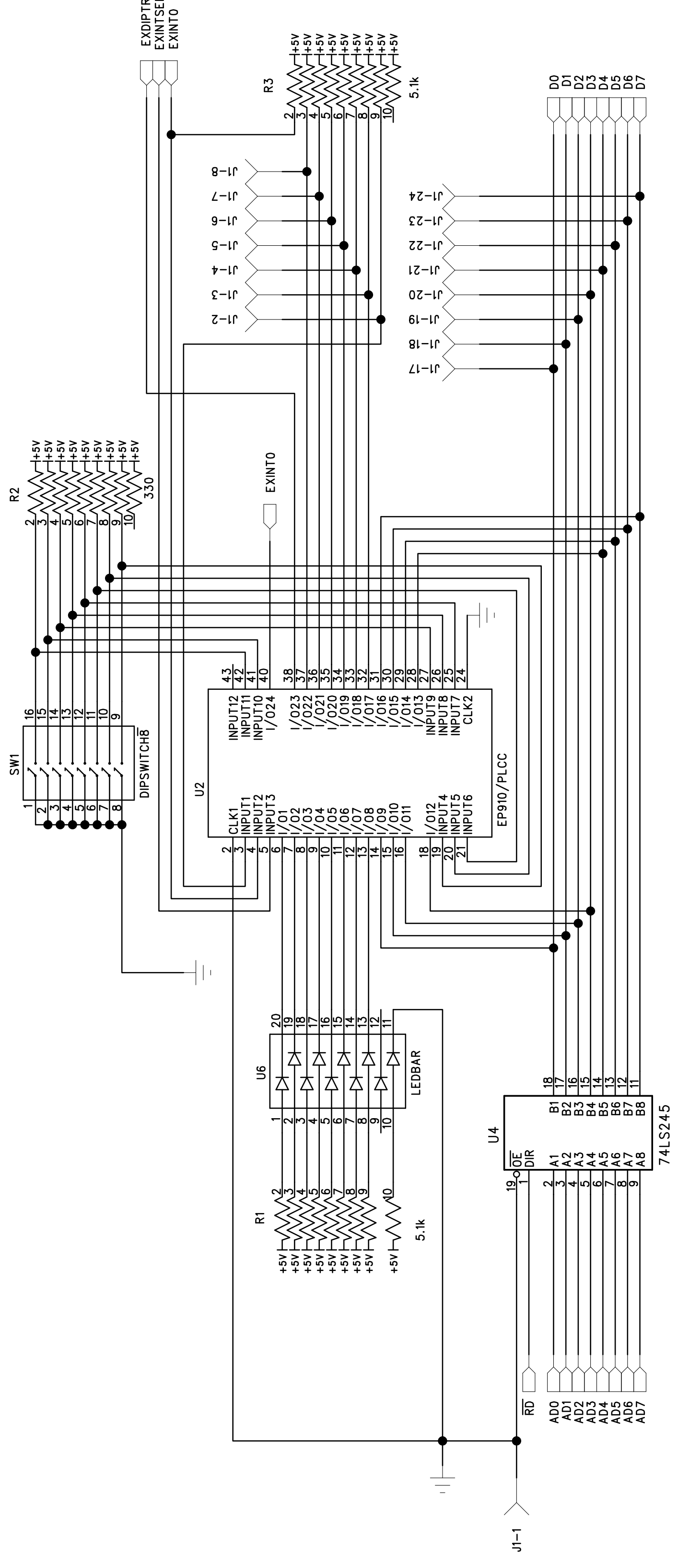
GC3S1.SCH1 - Thu Aug 19 23:55:01 1993



GC3S2.SCH1 - Thu Aug 19 15:21:45 1993



GC3S3.SCH1 - Thu Aug 19 23:11:09 1993



Appendix B

The Three-Beam Laser Rangefinder

B.1 Introduction

We have designed and built a laser rangefinding device for use on our mobile robot, LILY. The rangefinder has the following characteristics:

- It can sense the distance to an object in front of it which is at least 350mm away and within approximately 4 meters.
- It returns two distance values at once, related in a way which allows calculation of approximate surface normal information.
- It can compute up to 20 range reading pairs per second.

This paper provides detailed documentation on the use of the three-beam laser ranger, as well as its design. It also provides a brief background on a few of the many laser rangefinders which have been built for use with mobile robots in the past. Also included is a copy of the schematic for the three-beam ranger.

B.2 Underlying Theory

The main idea behind the three-beam ranger is that of *perspective*. The *perspective projection* from a 3D world onto a 2D imaging surface specifies that each point in the world projects onto the image by projecting a straight line through the point and through a special point called the *focus*, and placing the projection of the point at the imaging surface coordinate where that line pierces the imaging surface. The important thing to note about the perspective projection is that the apparent distance

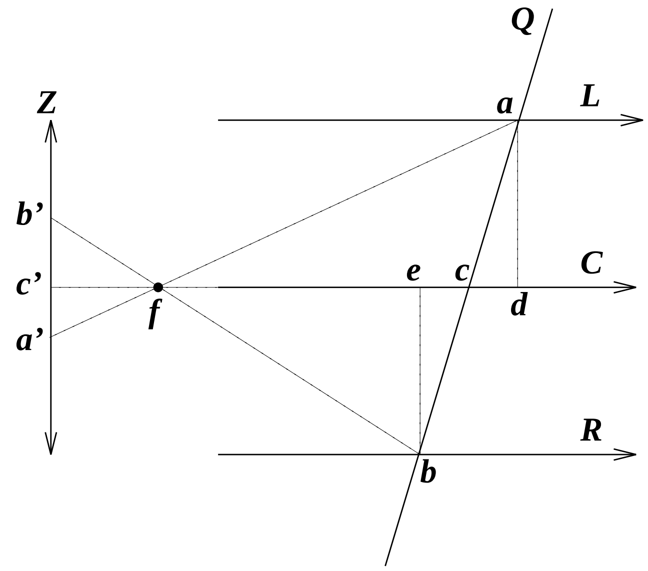


Figure B.1: The Geometry that Makes the Rangefinder Work

(on the imaging surface) between two points in the world is inversely proportional to the distance of those points from the focal point of the imaging surface.

Figure B.1 shows the geometry of the laser ranger. Three diode-projected laser beams, L, C, and R, are projected into the world so that the beams are parallel to each other and perpendicular to the imaging surface, Z, and so that C is as close to colinear with the focal point f as possible. Q is the boundary of an object in the world. a, b, and c mark the intersection of Q with L, R, and C.

Since these rays are parallel, the perpendicular distance between them is constant over all ranges from the focal point of the camera. The projection of a given point on one of the rays, however, is dependent on its distance from the imaging surface. All points on the center ray project onto the center of the imaging surface, directly behind the focal point. Points on the side rays project onto the imaging surface at a distance from that center which is inversely proportional to the distance from the focus to the point in question. This relationship is derived through a straightforward use of similar triangles. We denote the line segment connecting two points, a and b, by $\ell(a,b)$, and the length of that segment by $|\ell(a,b)|$. The dotted segments, $\ell(a,d)$ and $\ell(b,e)$ are perpendicular to the three rays, and mark the lines along which a and b are projected onto the center ray. These segments are parallel to the imaging surface. We can see that, for example, triangle adf and triangle $a'c'f$ are similar. The length of $\ell(c',f)$ is known and fixed, as is the length of $\ell(a,d)$. By similar triangle properties, we know that

$$\frac{|\ell(a',c')|}{|\ell(c',f)|} = \frac{|\ell(a,d)|}{|\ell(d,f)|}$$

By finding the point on the imaging surface, Z onto which point a projects, we determine the third of the four quantities contained in the above identity. We can now determine the distance from the focal point to d from

$$|\ell(d,f)| = \frac{|\ell(c',f)||\ell(a,d)|}{|\ell(a',c')|}$$

We can compute the distance from the focal point to a from $\ell(d,f)$ and $\ell(a,d)$, using Pythagorus. In actuality, determining distance to the intersection point requires a slight bit of scaling. The distance information obtained from looking at the camera image is presented to us as a certain number of pixels. As we want distance to the intersection in a standard linear distance unit, we also have to multiply in a constant which depends on the pixel size. In practice, we simply divide the distance in pixels along the imaging surface into a constant K that incorporates the pixel size, the distance from the image surface to the focal point, and the distance from the center ray to the left or right ray, and that is determined empirically.

We chose to use three laser beams for two reasons. Note that, in theory, we can determine the distance to an object from a single beam. However, if we do this, we cannot determine anything about surface orientation in a single reading. We want to be able to detect surface normal in a single reading because it will be more accurate than normal information obtained by taking a reading, moving the ranger, and taking another reading; it is also faster to be able to get two distance readings at once. This explains the need for two beams, namely the left and right ones. The center beam is not strictly necessary for either the ranging computation or the normal computation; it does, however, drastically reduce the calibration complexity of the system. With only the side beams, we have to determine exactly where the side beams lie relative to the focus of the camera and make sure that this doesn't change. Precise determination of this information requires that numerous measurements be made and large amounts of time be spent on calibration. The center beam, as long as it is close to passing through the center of focus, allows us to eliminate fancy calibration. All that is necessary with the third beam in place, is that we keep the beams parallel, which requires only that we mount the laser diodes a fixed distance apart, and that the distance between the beams at some relatively large distance is the same as the distance between the mounts. Thus, we only have to determine the previously described scaling information once, and only have to take one set of measurements to ensure that the calibration is correct. Thus, the third beam makes the ranger much easier to use, and is well worth the monetary and computational costs of using it.

B.3 The Ranging Algorithm

Given the geometry of the laser rangefinder, the problem of calculating the left and right range values from the horizontal locations of the three bright spots in the image simply involves a couple of divisions. Much more challenging is the problem of finding the horizontal locations of the three bright spots in the image, given the time constraint and the limited computational capacity of the frame grabber board. We meet the time constraint by being careful about how we find the bright spots and also by using a programming technique which, while questionable by software engineering standards, allows us to perform many of the arithmetic operations used by the algorithm with a particularly high efficiency.

We are interested in determining the horizontal positions of the three bright spots in the image caused by the laser diodes, but don't particularly care about the vertical positions of those spots. The obvious way to remove vertical information from the image is to add up the intensity values in each column, and store those sums in a vector, on which all further computation is performed. Unfortunately, we found the cost of summing over all 64 rows and 512 columns of video memory to be prohibitively high. Instead, we determine the vertical region that is most likely to contain the bright spots, and sum over a 16 row section of the image which contains that interesting region. As figure B.2 shows, we begin looking for the bright spots in the image by determining where they lie vertically in the image. Since the center beam has very little horizontal movement over the entire span of possible range values, we can look for it in a small fixed horizontal window (between the vertical dashed lines) and be assured of finding it. This is exactly what we do: we sum up a small number of columns within that window over all 64 rows and look for the row which has the largest sum. This row becomes the center of the 16 row window (indicated by the horizontal dashed lines) over which we compute column sums. After we compute the column sums (figure B.3a), we look for the three brightest regions in the image: We determine the maximum column sum and mask out (set to zero) all entries in the column sum vector which are not at least one fourth of that maximum value (figure B.3b). This separates the column vector's nonzero entries into several zero-separated lumps of nonzero entries. We calculate the left and right boundaries of these lumps, as well as their total weights (the sum of all columns in

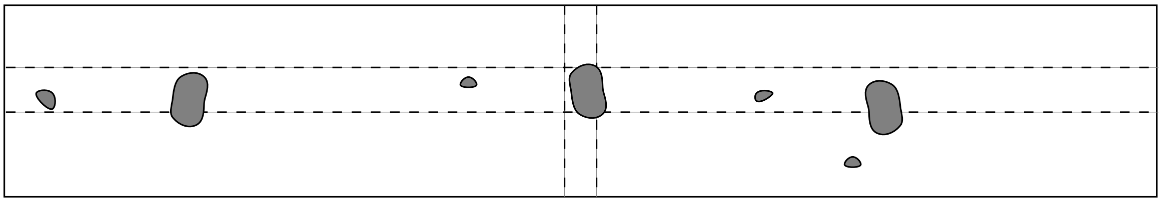


Figure B.2: Finding the 16-row band that contains most of the bright pixels

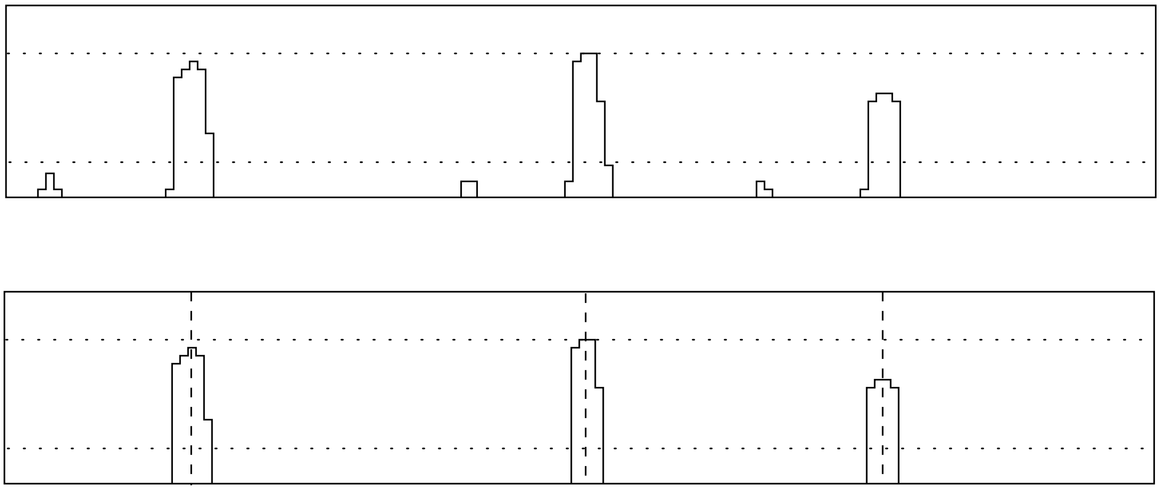


Figure B.3: Separating the column sums into discrete, bright bands, and calculating their centroids.

each lump). We then look for the three largest lumps and calculate their centroids. The centroids of these three lumps are returned as the horizontal positions of the three bright spots. A possible improvement to this algorithm is to retain the original column sums while storing the thresholded sums in a separate vector, so that we could calculate the centroids of the bright spots based on the original column sums. This would make little difference to the values returned on short ranges, but would likely improve accuracy on longer ranges.

Because of the limited number of computational cycles available to us on the microcontroller, we had to take steps to optimize the most common operations as much as we were able. This means that we had to make the column summing operation (adding up the contents of 16 consecutive elements in a given column) as fast as we possibly could. The typical way to code the operation of summing over a range of 16 rows and 512 columns is a doubly nested loop:

```
for i = 0 to 511
   for j = 0 to 15
      colsum[i] += image[j][i];
```

There are, however, inefficiencies in this approach which, while not significant from a theoretical standpoint, allow this algorithm to be sped up by a factor of two or three. The slowness in this code results from the fact that each iteration of this loop, over and above the load and addition operations, performs a two dimensional array index computation (only one dimension of which can be handled in hardware), increments at least one counter, and tests those counters to see if the limits have been reached. By employing a combination of loop-unrolling and self-modifying code, we can eliminate almost all extra operations from the "inner loop" of this computation. The code then looks more like:

```
for i = 0 to 511
   colsum[i] = image[i] + (image +  512)[i] + (image + 1024)[i]
                        + (image + 1536)[i] + (image + 2048)[i]
                        + . . .
                        + (image + 7680)[i]
```

where `(image + 512*j)[i])` refers to the same location as `image[i][j]` (this is just substituting a 1-dimensional array abstraction for a 2-dimensional, column-major array abstraction). Having explicit code for each row allows us to unroll the inner loop completely, eliminating all of the bounds checking for that loop; compiling the row offsets into the code directly eliminates the extra arithmetic needed for 2D array referencing and permits us to make direct use of the indexed addressing mode of the microcontroller for efficient coding. One more step is needed to make this implementation complete: As outlined, we have a routine which computes column sums for a fixed set of 16 rows. But we specified earlier that we wanted to be able to select the 16-row subset of the image dynamically. We achieve this by employing a very old and very questionable coding technique: self-modifying code. During initialization, the microcontroller makes a RAM-resident copy of the column-summing subroutine. During each ranging operation, after determining the bright row which is to be the center of the 16 row window, the microcontroller modifies the opcodes of the RAM-resident routine so that the absolute offsets of the 16 array references in the routine are those of our 16 row window. This allows us to choose the window we want while still allowing us to compute our column sums in the most efficient way.

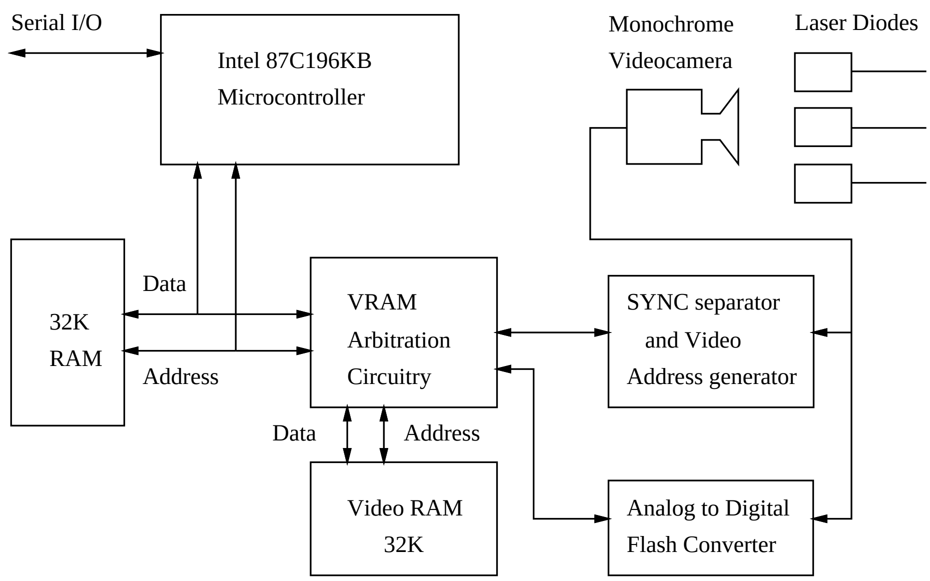


Figure B.4: Block Diagram of the Three Beam Rangefinder Hardware

B.4 The Real-Time Partial Frame Grabber

The only part of the ranger which is not essentially off-the-shelf is the partial frame grabber, which digitizes specified portions of the image provided by the ranger's video camera and provides the system programmer with a limited amount of general purpose computational power with which range values can be computed (or anything else the system programmer might wish to use the partial frame grabber for). The partial frame grabber, a block diagram of which is depicted in figure B.4, is essentially a single-board computer which is built around an Intel 87C196KB microcontroller and a Motorola MC10321 flash A/D converter. Each of these chips is the center of one of the board's two main modules, the microcomputer module and the partial-image digitization module.

The microcomputer module consists of the Intel microcontroller (an 8-/16-bit microcontroller with 256 bytes of built-in RAM and 8K of built-in EPROM), a MAX-232 serial I/O driver chip, 32K of SRAM, and some buffering. The microcomputer module is basically a generic microcontroller/microcomputer circuit, with the ability to communicate via RS-232 with the outside world and the ability to perform general purpose computation. In addition, however, the microcontroller has the ability directly to access the SRAM which is part of the digitization module. This allows the microcontroller to access elements of the digitized image for no higher cost than general-purpose memory access. The microcomputer module can also suppress image digitization. This allows it to control when images are digitized, which is important because (a) range computations (for example) may not always be completed in one field time and (b) if the user wants to examine the raw data that went into the computation of a range reading, we want that data retained as-is until the user is done with it.

The digitization module performs all of the functions necessary to digitizing a video stream and storing partial frames in RAM. The MC10321 flash A/D converter performs the digitization, but a number of support chips are necessary to control timing and storage issues. The video signal is connected to the A/D, but also to an LM1881 sync separator, which extracts HSYNC and VSYNC signals from the video signal. These signals are necessary to synchronize the counters used to keep track of row and column numbers with the actual video signal. There are two counters in the system. The pixel column counter is a 9-bit counter (implemented with three 74LS193s) which is clocked by a voltage-controlled oscillator (74LS624) running at 8.056 MHz and reset by the HSYNC pulse. 8.056MHz is the pixel rate, based on the NTSC standard of 29.97 frames per second and 525 lines per frame, as well as the desire to digitize 512 pixels per scan line ($29.97 \times 525 \times 512 = 8.056$MHz). The row counter is an 8-bit (two 74LS193s) counter clocked by HSYNC and reset by VSYNC; 8 bits is sufficient to count up to 255, which allows digitization of all but the first and last 8 scan lines of a particular video field (the last eight because the counter doesn't go high enough, and the first eight because they are indistinguishable from the last eight with an 8-bit counter). The two counters together form a 17-bit pixel address. Two 8-bit DIP switches allow the user to specify a minimum and maximum scan line number to be digitized within each field; all scan lines between these two values get digitized and stored into RAM (although if the difference is more than 64, only the last 64 scan lines in the specified window are preserved). Two 8-bit comparators (each made up of two 74LS85 4-bit comparator ICs) and some glue logic provide a signal which indicates when the scan line counter is within the specified range. This signal is provided to the microcontroller so that the program can tell at a given time whether or not it has access to the video RAM; it is gated by the microcontroller's override to determine whether or not the digitizer is allowed to access the video RAM.

The bus configuration on the frame grabber is somewhat complicated. This is caused by the requirements that the digitizer and the microcontroller each be able to access the video RAM and that the microcontroller must be able to access its own RAM simultaneously with the digitizer accessing the video RAM. We've satisfied these requirements with a simple processor-controlled arbitration circuit implemented with a handful of tri-state buffers. There are, in fact, two separate data busses and two separate address busses. The video data bus (VDB) connects the A/D converter to the video RAM arbitration circuitry (VAC). The processor data bus (PDB) connects the processor to its own RAM and to the VAC. The two address busses are hooked up similarly: the video address bus (VAB) connects the lower 15 bits of the pixel counter to the VAC, while the processor address bus (PAB) connects the processor to its RAM and to the VAC. The VAC is actually six 8-bit tri-state buffers and some glue logic. Two of the buffers (U6 and U9 on the schematic) allow the video RAM to be connected through to at most one of the VDB (through U9) and the PDB (through U6); the other four allow the RAM to be connected through to at most one of the VAB (through U10 and U11) and the PAB (through U7 and U8). All of these buffers are read-only from the video RAM's perspective, with the exception of the buffer that connects it to the PDB; this buffer is write-only. This configuration allows addresses to be written to the video RAM by either the processor or the digitizer and allows data to be *written* by the digitizer, but only *read* by the processor. The processor controls which busses can access the video RAM via the lowest bit of port 1 (the built-in bidirectional port): writing a 0 to that port disables frame grabbing and enables the processor to access the video RAM, while a 1 enables frame grabbing. When frame grabbing is enabled, the processor can sense (through the lowest bit of port 0 (the built-in input port)) whether the frame grabber is "in-band" (i. e., whether the current scan-line falls within the specified window), so that range computation can always be made between frames. As noted previously, only the lower 15 bits of the pixel counter are connected to the VAC. This makes sense, since the video RAM is 32 Kbytes, which requires a 15 bit address; the upper two bits of the VAB are ignored by the SRAM. 32K divided by 512 is 64, which is why we are able to digitize and retain up to, but no more than 64 scan lines from a given video field.

B.5 Other Laser Rangerfinders

A variety of ranging devices exist which incorporate laser beams. [Bes88] provides a survey of a large number of optical range imaging sensors of various types. He categorizes these sensors as using one or more of the following optical principles: radar, optical triangulation, moiré, holographic interferometry, focusing, and diffraction. The survey provides background on optical ranging, and also gives some basis for comparison of different types of range sensors. We describe only a few rangers here, and refer readers to [Bes88] for a much more comprehensive survey. The most sophisticated of these is the laser radar. These rangers, often employed by surveyors, use essentially a time of flight measurement to obtain the range value. These rangers can be manufactured to be quite accurate over fairly large ranges of distances, often in the tens of kilometers; the two main drawbacks are cost, which can be in the tens to hundreds of thousands of dollars, and the requirement, at least for longer ranges, of a reflector at the other end of the distance to be measured. For more details, see [Deu82].

Another popular laser ranging technique is the one known as *optical triangulation*, or "light-striping". With light striping, the process of acquiring a depth map of a scene consists of taking a number of scans which have the following form: A laser beam is passed through a cylindrical lens to transform it into a plane of light. This light strikes an object in the environment, causing a linear contour. A camera located slightly away from the lens takes an image of the scene with the laser line in place. Image processing is used to extract the location of the line in the image, and triangulation is used to calculate depth. This is typically done several times at closely spaced intervals to obtain a contour of a three dimensional object; it could easily be done a single time with a horizontal plane of light to obtain a contour at a single height. Kanade presents a VLSI based light striping sensor in [KGC91] which relies on analog electronics and a very simple approach to looking for the laser beam to provide a fast and effective light striping range sensor.

A recent result which uses a laser as part of a range finder appears in [BRD91]. This device uses a camera which, instead of a normal diaphragm iris, sports a double pinhole aperture. This causes the picture of the line projected by the laser to appear twice in the image, at separations which are related to distance. This system apparently gives robust performance with respect to performance under high ambient light conditions due to the fact that any line appearing in the image which cannot be matched up with another, similar line coming from the other pinhole, can be treated as spurious data and disregarded.

A very useful-looking laser ranger which does not use a video camera is that designed at Oxford by Pears [PP93]. This system uses a Lateral Effect Photodiode (LEP) and a laser diode configured after the optical triangulation model. The LEP returns signals from which the lateral position of the centroid of the light hitting it can be extracted; by modulating the outgoing laser and filtering the LEP signals, an accurate estimate of the distance to the nearest object can be obtained. Using mirrors, they are able to scan at 10 Hz and obtain 256 range readings per scan over a range from 0.4m to 2.5m.

B.6 Using the Ranger

The three-beam rangefinder is very easy to use. It provides essentially turnkey operation, and has only three commands, two of which are provided for diagnostic purposes. As currently configured, the ranger requires approximately one Ampere of current at 5.5 - 8.0 volts, and communicates over an RS-232 serial line. Its command set is as follows: (all commands are single, lower case characters and require no line termination.)

- 'r': When the laser rangefinder receives the character 'r' over its serial line, it initiates a range reading calculation. Approximately 50msec later, it transmits the results of that calculation over the serial line in the format "`Lllll Rrrrr<CR-LF>*`", where "llll" and "rrrr" are the distances to the points where the leftmost and rightmost beams impinge on the environment.

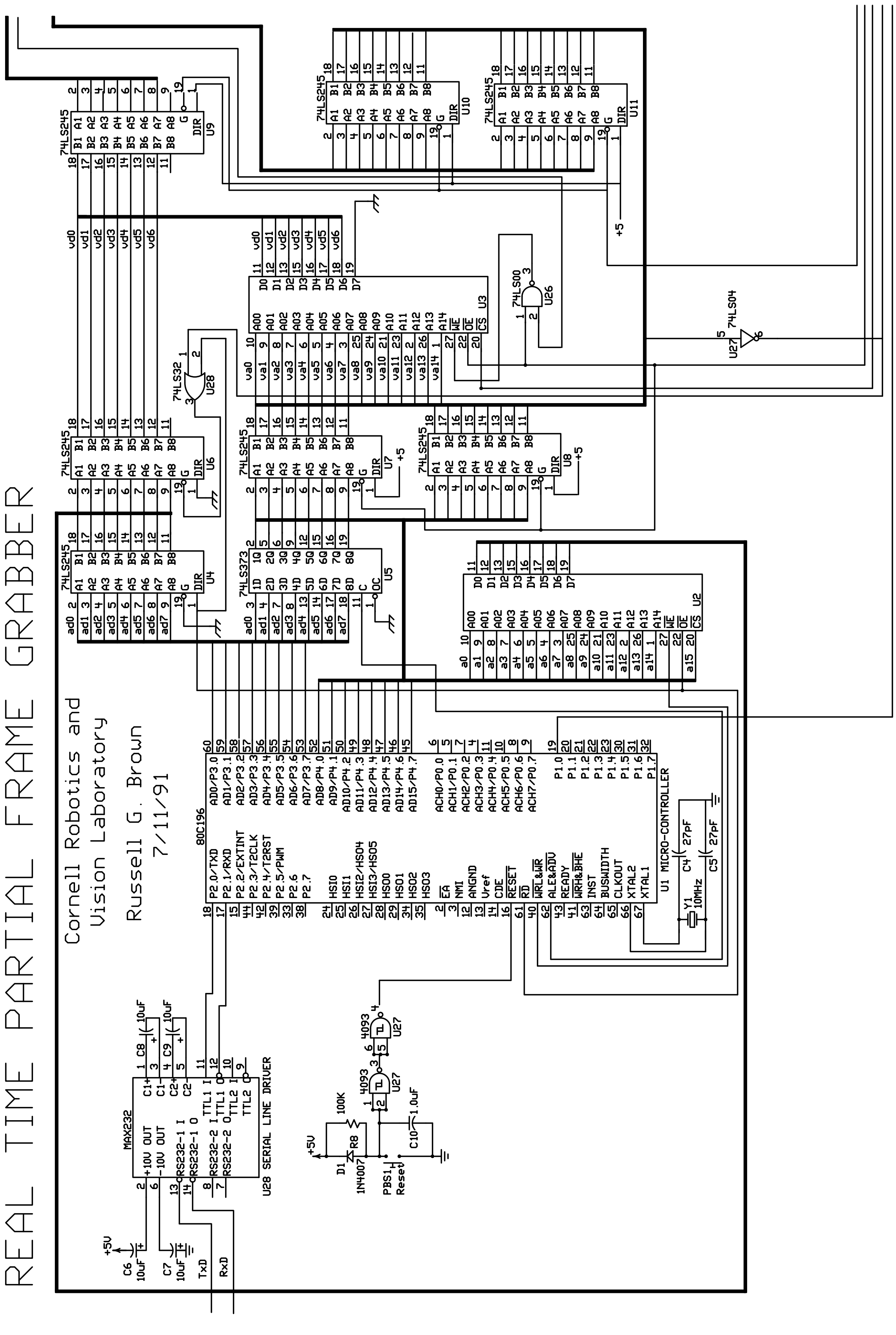
- 'i': The character 'i' causes the laser ranger to send over its serial line a hexidecimal dump of the most recently acquired image of the ranger's frame grabber. This is a diagnostic function, intended to allow the user to determine whether the frame-grabber is functioning properly, and is not ordinarily useful, as the time required to dump an image is on the order of minutes.

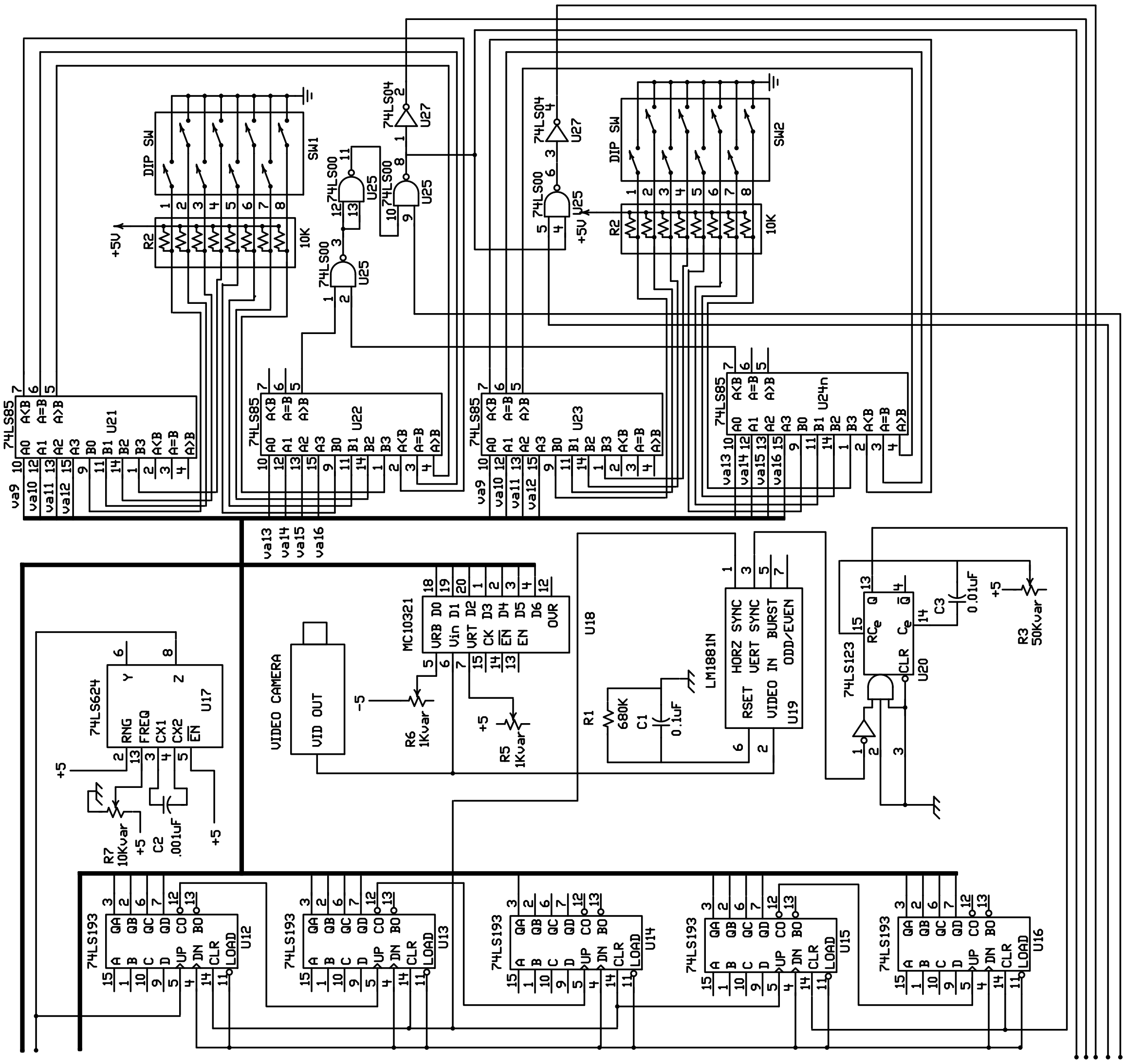
- 'c': this is also a diagnostic function. It causes the ranger to transmit a hexidecimal dump of the column sums calculated in the most recent rangefinding. This allows the user to determine whether or not the bright spots in the image used to calculate distances were in fact those caused by the framegrabber's laser diodes.

B.7 Conclusion

We have designed and built a laser rangefinding sensor which allows us to take point-and-shoot range readings over a range of 350mm to about 4m, at a rate of 20 range readings per second, each of which contains two individual distance measurements, providing limited ability to calculate surface normal information. The sensor is made entirely of off-the-shelf and readily available components, consisting of three laser diodes, a monochrome videocamera, and a real-time partial frame-grabber (which built of an Intel 87C196KB microcontroller, a Motorola flash A/D converter, and a couple of dozen 74LSxx series ICs). The hardware design is somewhat preliminary, in the sense that, were we to make another one, there are numerous changes of hardware which we would make, but it has been functioning nearly flawlessly for well over a year now, and has met our expectations excellently.

We use this rangefinder as the main sensor for our work in mapmaking and localization. While there are numerous other laser rangefinders in existence, ours has a couple of advantages for our application. One is that it is relatively inexpensive and fairly easy to fabricate (the parts for the current design of the ranger would cost less than \$1200, the only machining required was the construction of the mounts for the camera and the diodes, and the frame-grabber was constructed completely with wire-wrap technology). The other, perhaps more important advantage is that the bandwidth of the sensor is very well matched to the rest of our system: It returns data at a rate which is appropriate for our computational hardware's ability to utilize it (it doesn't starve the computer's need for data, and the data it returns is compact). In addition, the frame-grabber is useful in its own right: Given that the cost of a decent line camera is several times that of a 2D camera, and that the frame-grabber can be built for on the order of \$200, using a 2D camera and the partial frame-grabber is an economical alternative to using a line camera for applications where the frame rate of the 2D camera is sufficient. Also, for applications where limited vertical resolution is valuable, this frame-grabber is the correct one for the job.


REAL TIME PARTIAL FRAME GRABBER
Cornell Robotics and Vision Laboratory
Russell G. Brown
7/11/91


74LS85
U21
U22
U23
U24n
DIP SW
SW1
SW2
R2
10K
+5V
74LS00
U25
74LS04
U27
va9
va10
va11
va12
va13
va14
va15
va16
MC10321
U18
VIDEO CAMERA
VID OUT
R6
1Kvar
R5
1Kvar
LM1881N
U19
HORZ SYNC
VERT SYNC
BURST
ODD/EVEN
RSET
VIDEO IN
R1
680K
C1
0.1uF
74LS123
U20
C3
0.01uF
R3
50Kvar
74LS624
U17
R7
10Kvar
C2
.001uF
74LS193
U12
U13
U14
U15
U16

Appendix C

Mobot Code to Perform Localization, Mapmaking, Path Planning, and Constrained Manipulation

The purpose of this Appendix is to document in detail the mapmaking, localization, path-planning, and manipulation algorithms we use on our mobile robots. All of this code is written in either Mobot SCHEME or in C, and runs directly on the robots. The localization algorithm is discussed in detail in Chapter 2. The mapmaking and path-planning algorithms are discussed in Chapter 3. The manipulation algorithms are presented in Chapter 5.

C.1 locate.c

```
#include <stdio.h>

#include "geometry.h"
#include "matrix-accessors.h"
#include "queue.h"
/*#include <math.h>*/
#include "/usr/u/robotlab/rscheme/s48/scheme48.h"

/*********************************************************************
*                                                                   *
*       If you want to use locate.c without map.c, uncomment the line    *
*       that says "#define LOCATE_STANDALONE".  Otherwise, it's not      *
*       needed.                                                     *
*                                                                   *
*********************************************************************/
```

```
/*#define LOCATE_STANDALONE 1 */

#define max(x,y) ((x > y) ? (x) : (y))  /* you always need these */
#define min(x,y) ((x > y) ? (y) : (x))
#define abs(x) ((x>0)?(x):(-x))
#define rint(x) (x)
#define between(x,y,z) ((x >= y) && (x < z))

extern short sin_theta[],cos_theta[];
#define r_sin_theta(r,theta) ((sin_theta[theta % 360] * r + 32767) >> 15)
#define r_cos_theta(r,theta) ((cos_theta[theta % 360] * r + 32767) >> 15)

#define PI2 (6.28319)
#define PI  (3.14159)

#define LEFT 0
#define RIGHT 1

#define get_angle(e) (atan2((double)(e.head.y - e.tail.y),\
        (double)(e.head.x - e.tail.x)))

/***************************************************************************
*                                                                         *
*       static declarations:                                              *
*                                                                         *
*       Since SCHEME eats all of the unallocated memory it can, we can't  *
*       have heap allocation of memory.  Therefore, we must declare       *
*       everything we need ahead of time.                                 *
*                                                                         *
***************************************************************************/

matrix threshmap;
matrix single_vector_locmap;
matrix localization_map;
matrix temp_map;

extern matrix grid; /* that's the map from SCHEME. */

#ifdef LOCATE_STANDALONE

#define MAXPOINTS (256)

qhd cell_list, free_cell_list;

#else

extern qhd cell_list, free_cell_list;

#endif


/* we'll need a queue on which to store the gridpoints which lie on our
   current localization vector, as well as one to store the gridpoints
```

```
   contained in our uncertainty ball. */

qhd point_list;
qhd epsilon_ball;

/* neighbor_map is just an integer array used in the localization process.*/
#define MAXPOINTS (256)
int neighbor_map[MAXPOINTS];
int look_ahead_map[MAXPOINTS];
int look_ahead_x[MAXPOINTS],look_ahead_y[MAXPOINTS];

/*********************************************************************
*                                                                   *
*       get_squared_length returns the square of the length of the      *
*       edge which is the argument.  This is used for length comparison, *
*       which is why we don't really need to bother with taking the     *
*       square root (squaring being monotonic on positive numbers).     *
*                                                                   *
*********************************************************************/

float get_squared_length(e)

edge e;

{

  return((float)((e.head.x - e.tail.x) * (e.head.x - e.tail.x) +
                 (e.head.y - e.tail.y) * (e.head.y - e.tail.y)));

}

/* this lookup table is the angular difference between the heading of the
   pan-tilt head and the angle from the center of the robot to the
   intersection point, indexed by distance in cells. */

static unsigned char BEAM_ANGLE_OFFSET[] = {
  0,  0,  0,  0,  0,  0, 14, 12, 11,  9,  8,  8,  7,  6,  6,  6,  5,  5,  5,
  4,  4,  4,  4,  4,  3,  3,  3,  3,  3,  3,  3,  3,  3,  2,  2,  2,  2,  2,
  2,  2,  2,  2,  2,  2,  2};

#define beam_angle_offset(range) \
  ((range<6)?0:((range > 43)?0:BEAM_ANGLE_OFFSET[range]))

/*********************************************************************
*                                                                   *
*       pixelize_edge converts an edge into a list (queue) of cells     *
*       through which it passes.  This version includes the endpoints   *
*       of the edge.  It stores the cells in the queue argument p_q     *
*       and returns the number of cells.                                *
*                                                                   *
*********************************************************************/

#define MAXLENGTH 256
```

```
int pixelize_edge(e,p_q)

edge e;
qhd p_q;

{

  float slope,islope;
  int i,j;
  float curx,cury;
  point *new_point;

  if (get_squared_length(e) > (MAXLENGTH * MAXLENGTH))
    return(0);

  if (abs(e.head.x - e.tail.x) > (abs(e.head.y - e.tail.y)))
  {
    slope = ((float)(e.head.y - e.tail.y)) / (e.head.x - e.tail.x);
    if (e.head.x < e.tail.x)
    {
      cury = e.head.y;
      for (i = e.head.x; i <= e.tail.x; i++)
      {
        new_point = (point*) dequeue(free_cell_list);
        new_point->x = i;
        new_point->y = (int)rint(cury);
        enqueue(p_q,new_point);
        cury += slope;
      }
    }
    else
    {
      cury = e.head.y;
      for (i = e.head.x; i >= e.tail.x; i--)
      {
        new_point = (point*) dequeue(free_cell_list);
        new_point->x = i;
        new_point->y = (int)rint(cury);
        enqueue(p_q,new_point);
        cury -= slope;
      }
    }
  }
  else
  {
    islope = ((float)(e.head.x - e.tail.x)) / (e.head.y - e.tail.y);
    if (e.head.y < e.tail.y)
    {
      curx = e.head.x;
      for (j = e.head.y; j <= e.tail.y; j++)
      {
        new_point = (point*) dequeue(free_cell_list);
        new_point->y = j;
        new_point->x = (int)rint(curx);
```

```
        enqueue(p_q,new_point);
        curx += islope;
      }
    }
    else
    {
      curx = e.head.x;
      for (j = e.head.y; j >= e.tail.y; j--)
      {
        new_point = (point*) dequeue(free_cell_list);
        new_point->y = j;
        new_point->x = (int)rint(curx);
        enqueue(p_q,new_point);
        curx -= islope;
      }
    }
  }
  return(p_q->qlen);
}

/**********************************************************************
*                                                                     *
*       matrix_offset copies in_matrix into out_matrix translated by  *
*       (xoffset,yoffset).  It fills out the part of out_matrix not   *
*       covered by in_matrix with 0.                                  *
*                                                                     *
**********************************************************************/

void matrix_offset(in_matrix,out_matrix,xoffset,yoffset)

matrix in_matrix,out_matrix;
int xoffset,yoffset;

{

  int i,j,xmin,xmax,ymin,ymax;

  xmin = (xoffset<0)?GRID_MIN:GRID_MIN - xoffset;
  xmax = (xoffset>0)?GRID_MAX:GRID_MAX - xoffset;
  ymin = (yoffset<0)?GRID_MIN:GRID_MIN - yoffset;
  ymax = (yoffset>0)?GRID_MAX:GRID_MAX - yoffset;

/*  printf("xoffset:  %d\n\ryoffset:  %d\n\r",xoffset,yoffset);*/

  for (i = GRID_MIN; i < xmin; i++)
    for (j = GRID_MIN; j < GRID_MAX; j++)
      matrix_set(out_matrix,i,j,0);
  for (i = xmin; i < xmax; i++)
  {
    for (j = GRID_MIN; j < ymin; j++)
      matrix_set(out_matrix,i,j,0);
    for (j = ymin; j < ymax; j++)
      matrix_set(out_matrix,i,j,matrix_ref(in_matrix,i-xoffset,j-yoffset));
    for (j = ymax; j < GRID_MAX; j++)
```

```
      matrix_set(out_matrix,i,j,0);
  }
  for (i = xmax; i < GRID_MAX; i++)
    for (j = GRID_MIN; j < GRID_MAX; j++)
      matrix_set(out_matrix,i,j,0);
}

/**********************************************************************
*                                                                    *
*       matrix_or does a cell-wise, bit-wise OR of im2 into im1 in the   *
*       specified color.  In other words, if color is 1 (true), the two  *
*       matrices are OR'd together, while if color is 0 (false), the     *
*       matrices are AND'd together.  This corresponds to active-high    *
*       and active-low ORing.                                            *
*                                                                    *
**********************************************************************/

void matrix_or(im1,im2,color)

matrix im1,im2;
int color;

{

  int i,j;

  if (color == 1)
  {
    for (i = GRID_MIN; i < GRID_MAX; i++)
      for (j = GRID_MIN; j < GRID_MAX; j++)
        matrix_set(im1,j,i,matrix_ref(im1,j,i) | matrix_ref(im2,j,i));
  }
  else
  {
    for (i = GRID_MIN; i < GRID_MAX; i++)
      for (j = GRID_MIN; j < GRID_MAX; j++)
        matrix_set(im1,j,i,matrix_ref(im1,j,i) & matrix_ref(im2,j,i));
  }
}

/**********************************************************************
*                                                                    *
*       matrix_add does a cell-wise, bit-wise addition of im2 into im1.  *
*                                                                    *
**********************************************************************/

void matrix_add(im1,im2)

matrix im1,im2;

{

  int i,j;
```

```
  for (i = GRID_MIN; i < GRID_MAX; i++)
    for (j = GRID_MIN; j < GRID_MAX; j++)
      matrix_set(im1,j,i,matrix_ref(im1,j,i) + matrix_ref(im2,j,i));

}

/***********************************************************************
*                                                                      *
*       epsilon_disk provides a list of the gridpoints contained in a  *
*       disk of radius r.  It places the gridpoints in the provided    *
*       queue, e_q, and returns the area of the disk (in gridpoints).  *
*                                                                      *
***********************************************************************/

int epsilon_disk(r, e_q)

int r;
qhd e_q;

{

  int x,y;
  point* new_point;

  for (x = -r; x <= r; x++)
    for (y = -r; y <= r; y++)
      if ((x*x+y*y) <= r*r)
      {
        new_point = (point*) dequeue(free_cell_list);
        new_point->x = x;
        new_point->y = y;
        enqueue(e_q,new_point);
      }
  return(e_q->qlen);

}

/***********************************************************************
*                                                                      *
*    threshold_graymap sets binmap[i][j] to (map[i][j] < threshold).   *
*                                                                      *
***********************************************************************/

void threshold_graymap(map,threshold,binmap)

matrix map;
short threshold;
matrix binmap;

{

  short i,j;

  for (i = GRID_MIN; i < GRID_MAX; i++)
```

```
    for (j = GRID_MIN; j < GRID_MAX; j++)
      matrix_set(binmap,i,j,((matrix_ref(map,i,j) < threshold)?1:0));

}

/**********************************************************************
*                                                                     *
*       initialize localization_map to contain all init_val's.        *
*                                                                     *
**********************************************************************/

void init_loc_array(init_val)

char init_val;
{

  int i,j;

  for (i = GRID_MIN; i < GRID_MAX; i++)
    for (j = GRID_MIN; j < GRID_MAX; j++)
      matrix_set(localization_map,i,j,init_val);

}

/**********************************************************************
*                                                                     *
*       one_vector_localize applies the localization algorithm to one *
*       bitmap, one uncertainty, and one range vector.  It puts the   *
*       appropriately flipped and translated output map into locmap.  *
*                                                                     *
**********************************************************************/

void point_queue_to_neighbor_map(point_queue,neighbor_map,look_ahead_map,
                                 look_ahead_x,look_ahead_y)

qhd point_queue;
int *neighbor_map,*look_ahead_map,*look_ahead_x,*look_ahead_y;

{

  int i,j,num_points,dx,dy,this_neighbor_num;
  int dominant_nn,recessive_nn,dominant_neighbor_number;
  int recessive_neighbor_number;
  point *current_point,*next_point;
  int nn_count[8];  /* nn_count keeps track of how many times a given
                       neighbor number shows up in a particular edge. */

  for (i = 0; i < 8; i++)
    nn_count[i] = 0;
  num_points = point_queue->qlen;

#ifdef DEBUG
  printf("number of points to deal with: %d\n\r",num_points);
#endif
```

```
  i = num_points - 1;
  next_point = dequeue(point_queue);
  current_point = dequeue(point_queue);

  while (i-- > 0)
  {
#ifdef DEBUG
    printf("current point:  (%3d,%3d)  next point:  (%3d %3d)",
           current_point->x,current_point->y,next_point->x,next_point->y);
#endif

    dx = next_point->x - current_point->x;
    dy = next_point->y - current_point->y;
    if (dx > 0)
    {
      if (dy == 0)
        this_neighbor_num = 0;
      else if (dy == 1)
        this_neighbor_num = 1;
      else if (dy == -1)
        this_neighbor_num = 7;
    }
    else if (dx == 0)
    {
      if (dy == 1)
        this_neighbor_num = 2;
      else if (dy == -1)
        this_neighbor_num = 6;
    }
    else if (dx < 0)
    {
      if (dy == 0)
        this_neighbor_num = 4;
      else if (dy == 1)
        this_neighbor_num = 3;
      else if (dy == -1)
        this_neighbor_num = 5;
    }
    neighbor_map[i] = this_neighbor_num;
    enqueue(free_cell_list,next_point);
    next_point = current_point;
    current_point = dequeue(point_queue);
    nn_count[this_neighbor_num]++;

#ifdef DEBUG
    printf("  neighbor number:  %3d\n",neighbor_map[i]);
    for (j = 0; j < 30000; j++);
#endif
  }
  dominant_neighbor_number = -1;
  recessive_neighbor_number = -1;
  for (i = 0; i < 8; i++)
  {
```

```
    if ((nn_count[i] != 0) && dominant_neighbor_number == -1)
      dominant_neighbor_number = i;
    else if (nn_count[i] != 0)
      recessive_neighbor_number = i;
  }
  if ((recessive_neighbor_number >= 0) &&
      (nn_count[recessive_neighbor_number] >
               nn_count[dominant_neighbor_number]))
  {
    i = recessive_neighbor_number;
    recessive_neighbor_number = dominant_neighbor_number;
    dominant_neighbor_number = i;
  }
  for (i = 0; i < num_points-1; i++)
  {
    look_ahead_map[i] = -1;
    for (j = i; j < num_points-1; j++)
    {
      if (neighbor_map[j] == recessive_neighbor_number)
      {
        look_ahead_map[i] = j+1;
        break;
      }
    }
    if (look_ahead_map[i] == -1)
      look_ahead_map[i] = num_points - 1;
  }

#ifdef DEBUG
  for (i = 0; i < num_points; i++)
  {
    printf("i:  %2d; n_m[i]:  %2d; l_a_m[i]:  %2d;\n\r",i,
           neighbor_map[i],look_ahead_map[i]);
    for (j = 0; j < 30000; j++);
  }
#endif

  for (i = 0; i < num_points-1; i++)
  {
    look_ahead_x[i] = 0;
    look_ahead_y[i] = 0;
    for (j = i; j < look_ahead_map[i]; j++)
    {
      look_ahead_x[i] += EIGHT_NEIGHBOR_X[neighbor_map[j]];
      look_ahead_y[i] += EIGHT_NEIGHBOR_Y[neighbor_map[j]];
    }
  }
#ifdef DEBUG
  for (i = 0; i < num_points; i++)
  {
    printf("i:%2d;n_m[i]:%2d;l_a_m[i]:%2d;l_a_x[i]:%2d;l_a_y[i]:%2d\n\r",
      i,neighbor_map[i],look_ahead_map[i],look_ahead_x[i],look_ahead_y[i]);
    for (j = 0; j < 30000; j++);
  }
```

```
#endif
}

void one_vector_localize(threshmap,locmap,epsilon,range_vector)

matrix threshmap,locmap;
int epsilon;
point range_vector;

{

  int x_offset,y_offset;
  int point_list_length;
  int epsilon_ball_length;
  edge range_edge;
  int i,j;
  int currval;
/*  int fudge_neighbor;*/
  point *first_point, *current_point, *ball_point;
  int init_x_idx,final_x_idx,x_step_size;
  int init_y_idx,final_y_idx,y_step_size;
  int i_ct,j_ct;

  range_edge.head.x = -range_vector.x;
  range_edge.head.y = -range_vector.y;
  range_edge.tail.x = 0;
  range_edge.tail.y = 0;  /* range_edge is the inverted range vector. */

#ifdef DEBUG
  printf("range vector:  (%d,%d)\n\r",range_vector.x,range_vector.y);
#endif

  point_list_length = pixelize_edge(range_edge,point_list);

/* the new approach to localization is the following: as before, we make
   the localization map by forming an image of the set of pixels which are
   within epsilon of a pixel which is at distance r in direction theta from
   _some_ obstacle pixel, then by subtracting out those pixels which are
   closer than r-epsilon in direction theta to _some_ obstacle pixel,
   leaving only those which are at the right distance in the right
   direction from the _closest_ obstacle pixel (in the appropriate
   direction). We will, however, use a considerably faster tactic for each
   of the two parts.*/

/* the following performs the first step.  We take the end of the inverted
   range vector and offset the thresholded map by that amount.  We then grow
   all of the obstacle pixels in it out by about epsilon using the pseudo-
   circular floodfill in flood.c.  As we start, we have obstacle pixels
   colored 0, freespace colored 1.*/

  first_point = dequeue(point_list);
  matrix_offset(threshmap,locmap,first_point->x,first_point->y);
  pseudo_circle_fill(locmap,locmap,epsilon);
```

```
/* At this point, we've completed the first part of the job, except that we
   have to massage things a little bit.  Note that at this point, the pixels
   which are "feasible pose" are colored 0, with all others colored 255.
   At least, I think that's the case.  We also have to ditch the furthest
   epsilon-1 points of the range vector. */

  enqueue(free_cell_list,first_point);
  for (i = 1; i < epsilon; i++)
      enqueue(free_cell_list,dequeue(point_list));

/*Create a direct access neighbor map for what's left of the range vector.*/

  point_list_length = point_list->qlen; /* how many after epsilon removed?*/
  point_queue_to_neighbor_map(point_list,neighbor_map,look_ahead_map,
                              look_ahead_x,look_ahead_y);

/* Now for the second part.  We've (Paul, mostly) established a way to do a
   single sweep across the grid which will subtract out the pixels we don't
   want, but we have to choose the direction of the sweep based on the
   quadrant of the range vector. */

/* first, we need a single map which contains the original obstacles and the
   "fp" pixels from the first half.  Recolor so that 0 is obstacle, 255 is
   free/undesignated, 254 is fp. */

  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
    {
/*      if ((*(locmap.mdata))[i][j] == 255)
        (*(locmap.mdata))[i][j] = 255;*/
      if ((*(locmap.mdata))[i][j] == 0)
        (*(locmap.mdata))[i][j] = 254;
      if ((*(threshmap.mdata))[i][j] == 0)
        (*(locmap.mdata))[i][j] = 0;
    }

/* Now set up the sweep.  We have four ways to sweep, which we choose based
   on the quadrant of the inverted range vector.

   theta:           major sweep:            minor sweep:

     0-90             decreasing x            decreasing y
    90-180            increasing x            decreasing y
   180-270            increasing x            increasing y
   270-360            decreasing x            increasing y

   There is one bit of ugliness we still have to deal with: If the local
   surface tangent is close enough to the direction of the inverted range
   vector, we may get nothing growing out of an edge pixel (due to it not
   being "recognized" as an edge pixel).  To get around this, we're generous
   about what we consider a first neighbor.  We allow the first neighbor to
   be either the normally correct one or the one which is one tick further
   around the 8-circle.  That means that we might accidentally erase an fp
   pixel in a wierd borderline case, but sobeit.  */
```

```
/* it's actually worse than previously thought.  The matter of surface
   tangent and range vector direction interaction goes beyond the first
   obstacle pixel.  I'm going to try to implement a new solution that goes
   like this: We compute an array in which each value says "for a cell
   colored with color n, what cell other than the one pointed to by
   neighbor_map[n] do we have to color in order to cause us not to lose "too
   close" pixels?  In a case, for example, where the edge looks like:

   0123
       4567
           89AB
               CDEF

   we already have a neighbor map that looks like 000700070007000.  We need
   also to have a look-ahead map that looks like 44448888CCCC----.  What we
   have to do is (i) determine the "dominant" direction of the vector (the
   modal neighbor number, basically), (ii) determine the "recessive"
   direction of the vector (the other neighbor number (we'll only have 2 in
   all cases)), (iii) fill the ith position of the look-ahead map with the
   number j such that (a) the neighbor of the cell colored j-1 is the
   recessive one and (b) all cells colored [i..j-2] have the dominant
   neighbor number.  */

  if (range_vector.x >= 0)
  {
    init_x_idx = GRID_SIZE - 1;
    final_x_idx = 0;
    x_step_size = -1;
  }
  else
  {
    init_x_idx = 0;
    final_x_idx = GRID_SIZE - 1;
    x_step_size = 1;
  }

  if (range_vector.y >= 0)
  {
    init_y_idx = GRID_SIZE - 1;
    final_y_idx = 0;
    y_step_size = -1;
  }
  else
  {
    init_y_idx = 0;
    final_y_idx = GRID_SIZE - 1;
    y_step_size = 1;
  }

#ifdef DEBUG
  printf("x goes from %d to %d by %d.  y goes from %d to %d by %d.\n\r",
     init_x_idx,final_x_idx,x_step_size,init_y_idx,final_y_idx,y_step_size);
#endif
```

```
/*Now we want to actually do the sweep. The behavior of the sweep should be:

   value of cell:       value of neighbor:        action:

         0                       0                nothing
         0                     other              set neighbor to 1
       1 to r-epsilon            0                nothing
       1 to r-epsilon          other              set neighbor to cell_val+1
       more than r-epsilon      any               nothing
*/

#ifdef DEBUG
  printf("point_list_length = %d\n\r",point_list_length);
#endif

  for (i = init_x_idx, i_ct = 0; i_ct < GRID_SIZE; i+= x_step_size,i_ct++)
    for (j = init_y_idx, j_ct = 0; j_ct < GRID_SIZE; j+= y_step_size,j_ct++)
    {
      currval = (*(locmap.mdata))[i][j];
      if (currval < (point_list_length - 1))
         /* if current cell value < r-epsilon-1 */
      {
        if (between(i+EIGHT_NEIGHBOR_X[neighbor_map[currval]],0,GRID_SIZE)&&
            between(j+EIGHT_NEIGHBOR_Y[neighbor_map[currval]],0,GRID_SIZE)&&
            ((*(locmap.mdata))[i + EIGHT_NEIGHBOR_X[neighbor_map[currval]]]
                              [j + EIGHT_NEIGHBOR_Y[neighbor_map[currval]]]
                                 >= 254))
          (*(locmap.mdata))[i + EIGHT_NEIGHBOR_X[neighbor_map[currval]]]
                           [j + EIGHT_NEIGHBOR_Y[neighbor_map[currval]]] =
                                currval+1;
        if (between(i + look_ahead_x[currval],0,GRID_SIZE) &&
            between(j + look_ahead_y[currval],0,GRID_SIZE) &&
            ((*(locmap.mdata))[i + look_ahead_x[currval]]
                              [j + look_ahead_y[currval]] != 0))
          (*(locmap.mdata))[i + look_ahead_x[currval]]
                           [j + look_ahead_y[currval]] =
                             look_ahead_map[(*(locmap.mdata))[i][j]];
      }
    }

/* After doing that sweep, go through one more time and set each cell to
   (cell_val == 254).  That will give us 1 in the cells which are actually
   feasible poses. */

  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
      (*(locmap.mdata))[i][j] = ((*(locmap.mdata))[i][j] == 254)?1:0;

}

/*********************************************************************
*                                                                    *
*       add_vector_localization takes a thresholded map, an existing *
```

```
*       localization map, and a vector and epsilon, and adds             *
*       localization information based on that vector and epsilon to     *
*       the existing localization map.  sch_add_vector_localization      *
*       is the SCHEME interface (which assumes particular maps).         *
*                                                                        *
**************************************************************************/

#define AND (0)
#define ADD (1)

void add_vector_localization(threshmap,locmap,epsilon,
                             range_vector,and_or_add)

matrix threshmap,locmap;
int epsilon;
point range_vector;
int and_or_add;

{

  one_vector_localize(threshmap,single_vector_locmap,epsilon,range_vector);
  if (and_or_add == 0)
    matrix_or(locmap,single_vector_locmap,0);
  else
    matrix_add(locmap,single_vector_locmap);

}

scheme_value sch_add_vector_localization(nargs, args)

long nargs;
scheme_value *args;

{
  int epsilon;
  int range;
  int theta;
  point range_vector;
  int side;
  int and_or_add;

  and_or_add = EXTRACT_FIXNUM(args[4]);
  side = EXTRACT_FIXNUM(args[3]);
  range = EXTRACT_FIXNUM(args[2]);
  theta = EXTRACT_FIXNUM(args[1]);
  epsilon = EXTRACT_FIXNUM(args[0]);

  if (side == RIGHT)
    theta -= beam_angle_offset(range);
  else
    theta += beam_angle_offset(range);
  range_vector.x = r_cos_theta(range,theta);
  range_vector.y = r_sin_theta(range,theta);
  add_vector_localization(threshmap,localization_map,epsilon,
```

```
                        range_vector,and_or_add);
  return(SCHTRUE);

}

/**************************************************************************
*                                                                         *
*       prepare_for_localization initializes the localization map to     *
*       all ones(zeroes) and produces a bitmap from the occupancy grid   *
*       greymap so we can start localizing.  the SCHEME interface is     *
*       sch_prepare_for_localization.  We initialize to all ones if we   *
*       want to do and-ing localization, and all zeroes if we want to    *
*       do add-ing localization.                                         *
*                                                                         *
**************************************************************************/

void prepare_for_localization(graymap,threshmap, threshold, and_or_add)

matrix graymap,threshmap;
int threshold;
int and_or_add;
{

  init_loc_array(1 - and_or_add);
  threshold_graymap(graymap,threshold,threshmap);

}

scheme_value sch_prepare_for_localization(nargs,args)

long nargs;
scheme_value *args;

{
  int threshold;
  int and_or_add;

  and_or_add = EXTRACT_FIXNUM(args[1]);
  threshold = EXTRACT_FIXNUM(args[0]);
  prepare_for_localization(grid,threshmap,threshold,and_or_add);
  return(SCHTRUE);

}

/**************************************************************************
*                                                                         *
*       localization_initialization sets up and initializes all of the   *
*       grids we need for the localization process, including but not    *
*       limited to a matrix for the bitmap, a couple to compute          *
*       localization data into, and some temporary storage of various    *
*       sorts.  Also, if we're running standalone, initialize some       *
*       queue storage.                                                   *
*                                                                         *
**************************************************************************/
```

```
void localization_initialization()

{

  int i,j;

  printf("Initializing localization data structures.\n\r");

  threshmap.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  threshmap.xmin = GRID_MIN;
  threshmap.xmax = GRID_MIN + GRID_SIZE;
  threshmap.ymin = GRID_MIN;
  threshmap.ymax = GRID_MIN + GRID_SIZE;

  single_vector_locmap.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  single_vector_locmap.xmin = GRID_MIN;
  single_vector_locmap.xmax = GRID_MIN + GRID_SIZE;
  single_vector_locmap.ymin = GRID_MIN;
  single_vector_locmap.ymax = GRID_MIN + GRID_SIZE;

  localization_map.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  localization_map.xmin = GRID_MIN;
  localization_map.xmax = GRID_MIN + GRID_SIZE;
  localization_map.ymin = GRID_MIN;
  localization_map.ymax = GRID_MIN + GRID_SIZE;

  temp_map.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  temp_map.xmin = GRID_MIN;
  temp_map.xmax = GRID_MIN + GRID_SIZE;
  temp_map.ymin = GRID_MIN;
  temp_map.ymax = GRID_MIN + GRID_SIZE;

  epsilon_ball = (qhd) himalloc(sizeof(struct queuehead));
  epsilon_ball->qlen = 0;
  epsilon_ball->head = NULL;
  epsilon_ball->tail = NULL;

  point_list = (qhd) himalloc(sizeof(struct queuehead));
  point_list->qlen = 0;
  point_list->head = NULL;
  point_list->tail = NULL;
}

/*************************************************************************
*                                                                        *
*       return_loc_list takes a list of SCHEME arguments:                *
*               (x_vector,y_vector,max_size,startx,starty,endx,endy).    *
*       It assumes that x_vector and y_vector are SCHEME vectors         *
*       max_size long.  It scans the localization map for zeroes (that   *
*       is the sense that indicates a feasible pose), in the rectangle   *
*       ranging from startx,starty to endx,endy.  The first max_size     *
*       zeroes found have their coordinates entered into x_vector and    *
*       y_vector).  The return value is the number of zeroes found.      *
```

```
*                                                                         *
**************************************************************************/

scheme_value return_loc_list(nargs,args)

long nargs;
scheme_value *args;

{

  scheme_value x_vector, y_vector;
  int max_size,startx,starty,endx,endy;
  int i,j,index;

  x_vector = args[6];
  y_vector = args[5];
  max_size = EXTRACT_FIXNUM(args[4]);
  startx = EXTRACT_FIXNUM(args[3]);
  starty = EXTRACT_FIXNUM(args[2]);
  endx = EXTRACT_FIXNUM(args[1]);
  endy = EXTRACT_FIXNUM(args[0]);

  index = 0;
  for (j = starty; j < endy; j++)
    for (i = startx; i < endx; i++)
      if (matrix_ref(localization_map,i,j) != 0)
      {
        vector_ref(x_vector,index) = ENTER_FIXNUM(i);
        vector_ref(y_vector,index++) = ENTER_FIXNUM(j);
        if (index == max_size)
          return(ENTER_FIXNUM(index));
      }
  return(ENTER_FIXNUM(index));
}

#ifdef LOCATE_STANDALONE
int main(argc,argv)

{

  printf("Hello, world.\n\r");

}
#endif
```

C.2 map.c

```
/**********************************************************************
*                                                                    *
*       map.c                                                        *
*                                                                    *
*       SCHEME is too slow on mapping stuff, plus all of the arrays  *
*       count twice because of garbage collection.  I am therefore   *
```

```
*       going to see if I can speed things up by going to C.          *
*                                                                     *
***********************************************************************/

#include <stdio.h>
#include "queue.h"
#include "geometry.h"
#include "matrix-accessors.h"
#include "/usr/u/robotlab/rscheme/s48/scheme48.h"

char **_frontier;      /* this is the location of the heap frontier
                          (I think).  I need to be able to frob it. */

#define max(x,y) ((x > y) ? (x) : (y))  /* you always need these */
#define min(x,y) ((x > y) ? (y) : (x))

/* We statically declare the structures for the three main grids used by
   the mapmaking software:  grid is the occupancy matrix; cgrid is the sum
   of total change that happens in a given cell; tgrid is the number of
   _times_ a cell's value has been changed. */

matrix grid,tgrid;
/*cgrid should be a word_matrix, but I'm running out of storage, so I need
  the extra 64k it's taking up. */
matrix cgrid;

/* need a qhd or two lying around for my use.  Note that free_cell_list is
   so that we never ever ever have to reallocate a cell.  This is necessary
   because we can malloc (at least, before SCHEME starts we can malloc), but
   we cannot free. */

qhd cell_list, free_cell_list;

/* we'll predefine a certain number of points/cells which will go on
   free_cell_list.  MAXPOINTS is how many of these there will be.  We'll
   also predefine a certain number of queue buckets for our use.  There
   will be INITBUCKETS of these.  */

#define MAXPOINTS (4096)
#define INITBUCKETS (6144)

/* IMAGING_SURFACE_OFFSET is the distance from the robot's center to the
   imaging surface of the laser ranger, in mm.  */
#define IMAGING_SURFACE_OFFSET (110)

#ifdef WITH_LOCATE
extern matrix threshmap,single_vector_locmap,localization_map,temp_map;
#endif
#ifdef WITH_FLOOD
extern matrix flood_map,cspace_map;
#endif

/**********************************************************************
*                                                                     *
```

```
*       make-grids in SCHEME makes the grids and initializes them.    *
*       In C, we already have the structures made, so we just need    *
*       to initialize the structures and fill the grids.              *
*                                                                     *
**********************************************************************/

scheme_value make_grids()

{

  int i,j;

  grid.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  grid.xmin = GRID_MIN;
  grid.xmax = GRID_MIN + GRID_SIZE;
  grid.ymin = GRID_MIN;
  grid.ymax = GRID_MIN + GRID_SIZE;

/*  cgrid.mdata = (matrix_data) himalloc(sizeof(matrix_space));*/
  cgrid.xmin = GRID_MIN;
  cgrid.xmax = GRID_MIN + GRID_SIZE;
  cgrid.ymin = GRID_MIN;
  cgrid.ymax = GRID_MIN + GRID_SIZE;

  tgrid.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  tgrid.xmin = GRID_MIN;
  tgrid.xmax = GRID_MIN + GRID_SIZE;
  tgrid.ymin = GRID_MIN;
  tgrid.ymax = GRID_MIN + GRID_SIZE;

  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
    {
      (*(grid.mdata))[i][j] = GRID_INIT_VAL;
/*      (*(cgrid.mdata))[i][j] = 0;*/
      (*(tgrid.mdata))[i][j] = 0;
    }

  return(SCHTRUE);
}

/*********************************************************************
*                                                                     *
*       We want to be able to reset the map grids without rerunning   *
*       everything (resetting the server and reloading all SCHEME     *
*       code, so we have clear_grids to do it for us.                 *
*                                                                     *
**********************************************************************/


scheme_value clear_grids(long nargs, scheme_value *args)

{
```

```
  int i,j;

  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
    {
      (*(grid.mdata))[i][j] = GRID_INIT_VAL;
/*      (*(cgrid.mdata))[i][j] = 0;*/
      (*(tgrid.mdata))[i][j] = 0;
    }
  return(SCHTRUE);
}

/*******************************************************************
*                                                                  *
*       grid_to_vector is my mechanism for getting C matrix infor- *
*       mation back to SCHEME one row at a time (the whole thing at *
*       once is way too big).                                      *
*                                                                  *
*       I'm trying to do a primitive approximation of flexibility, *
*       but it's not really.  It's just a case statement.          *
*                                                                  *
*******************************************************************/

scheme_value grid_to_vector(long nargs, scheme_value *args)

{

  scheme_value sch_vect;
  short i;
  short row_number;
  short matrix_id;
  matrix mat;

  sch_vect = args[0];
  row_number = EXTRACT_FIXNUM(args[1]);
  matrix_id = EXTRACT_FIXNUM(args[2]);
  switch (matrix_id)
  {  /* set up mat to point to the desired grid (HACK!  HACK!) */
  case 0:  mat.mdata = grid.mdata; break;
#ifdef WITH_LOCATE
  case 1:  mat.mdata = localization_map.mdata; break;
  case 2:  mat.mdata = threshmap.mdata; break;
  case 3:  mat.mdata = single_vector_locmap.mdata; break;
  case 4:  mat.mdata = temp_map.mdata; break;
#endif
#ifdef WITH_FLOOD
  case 5:  mat.mdata = flood_map.mdata; break;
  case 6:  mat.mdata = cspace_map.mdata; break;
#endif
  default:  mat.mdata = grid.mdata;
  }
/*  printf("location of matrix:  %x\n\r",mat.mdata);*/
  for (i = 0; i < GRID_SIZE; i+=3)
    /* note that I'm reversing top-to-bottom order here:  This is because of
```

```
      a mirror imaging problem that I'm seeing when I render the greymap.
      I believe this to be merely a function of the difference between the
      mathematical grid (increasing x to right, increasing y upward) and
      the screen grid (increasing x to right, increasing y downward), but
      I'm not positive.  I am, however, sure enough about it that I'm
      fixing the problem (renderwise) here, rather than in the actual
      mapmaking stuff. */
    vector_ref(sch_vect,i/3) =
      ENTER_FIXNUM(((long)((*(mat.mdata))[i][GMAX-row_number]) << 16) +
        (((long)(*(mat.mdata))[min(GMAX-1,i+1)][GMAX-row_number]) << 8) +
        ((long)(*(mat.mdata))[min(GMAX-1,i+2)][GMAX-row_number]));
  return(sch_vect);
}

/*******************************************************************
*                                                                  *
*      grid_to_vector is my mechanism for getting C matrix infor-  *
*      mation back to SCHEME one row at a time (the whole thing at *
*      once is way too big).                                       *
*                                                                  *
*      I'm trying to do a primitive approximation of flexibility,  *
*      but it's not really.  It's just a case statement.           *
*                                                                  *
*******************************************************************/

scheme_value vector_to_grid(long nargs, scheme_value *args)
{

  scheme_value sch_vect;
  short i;
  short row_number;
  short matrix_id;
  matrix mat;

  sch_vect = args[0];
  row_number = EXTRACT_FIXNUM(args[1]);
  matrix_id = EXTRACT_FIXNUM(args[2]);
  switch (matrix_id)
  {  /* set up mat to point to the desired grid (HACK!  HACK!) */
  case 0:  mat.mdata = grid.mdata; break;
  default:  mat.mdata = grid.mdata;
  }
  for (i = 0; i < GRID_SIZE; i++)
    /* note that I'm reversing top-to-bottom order here:  This is because of
      a mirror imaging problem that I'm seeing when I render the greymap.
      I believe this to be merely a function of the difference between the
      mathematical grid (increasing x to right, increasing y upward) and
      the screen grid (increasing x to right, increasing y downward), but
      I'm not positive.  I am, however, sure enough about it that I'm
      fixing the problem (renderwise) here, rather than in the actual
      mapmaking stuff. */
    (*(mat.mdata))[i][GMAX-row_number] =
              EXTRACT_FIXNUM(vector_ref(sch_vect,i));
  return(SCHTRUE);
```

```
}

/* these tables (and other large lookup tables in this file) were
   automatically generated.  For the source (and to regenerate), see
   "~rbrown/robotics/scm/angles/gencalpha.c" and other like files. */

/* Note also that the current tables were generated with UP_ALPHA = 0.3 and
   DOWN_ALPHA = 0.15. */

static unsigned char RAISE_BY_ALPHA[] = {

 76,  77,  78,  79,  79,  80,  81,  81,  82,  83,  84,  84,  85,  86,  86,
 87,  88,  88,  89,  90,  90,  91,  92,  93,  93,  94,  95,  95,  96,  97,
 98,  98,  99, 100, 100, 101, 102, 102, 103, 104, 104, 105, 106, 107, 107,
108, 109, 109, 110, 111, 112, 112, 113, 114, 114, 115, 116, 116, 117, 118,
118, 119, 120, 121, 121, 122, 123, 123, 124, 125, 126, 126, 127, 128, 128,
129, 130, 130, 131, 132, 132, 133, 134, 135, 135, 136, 137, 137, 138, 139,
140, 140, 141, 142, 142, 143, 144, 144, 145, 146, 146, 147, 148, 149, 149,
150, 151, 151, 152, 153, 154, 154, 155, 156, 156, 157, 158, 158, 159, 160,
160, 161, 162, 163, 163, 164, 165, 165, 166, 167, 167, 168, 169, 170, 170,
171, 172, 172, 173, 174, 174, 175, 176, 177, 177, 178, 179, 179, 180, 181,
181, 182, 183, 184, 184, 185, 186, 186, 187, 188, 188, 189, 190, 191, 191,
192, 193, 193, 194, 195, 195, 196, 197, 198, 198, 199, 200, 200, 201, 202,
203, 203, 204, 205, 205, 206, 207, 207, 208, 209, 210, 210, 211, 212, 212,
213, 214, 214, 215, 216, 216, 217, 218, 219, 219, 220, 221, 221, 222, 223,
224, 224, 225, 226, 226, 227, 228, 228, 229, 230, 230, 231, 232, 233, 233,
234, 235, 235, 236, 237, 238, 238, 239, 240, 240, 241, 242, 242, 243, 244,
244, 245, 246, 247, 247, 248, 249, 249, 250, 251, 251, 252, 253, 254, 254,
255

};

static unsigned char LOWER_BY_ALPHA[] = {
  0,   1,   2,   3,   3,   4,   5,   6,   7,   8,   8,   9,  10,  11,  12,
 13,  14,  14,  15,  16,  17,  18,  19,  20,  20,  21,  22,  23,  24,  25,
 25,  26,  27,  28,  29,  30,  31,  31,  32,  33,  34,  35,  36,  37,  37,
 38,  39,  40,  41,  42,  42,  43,  44,  45,  46,  47,  48,  48,  49,  50,
 51,  52,  53,  54,  54,  55,  56,  57,  58,  59,  60,  60,  61,  62,  63,
 64,  65,  65,  66,  67,  68,  69,  70,  71,  71,  72,  73,  74,  75,  76,
 76,  77,  78,  79,  80,  81,  82,  82,  83,  84,  85,  86,  87,  88,  88,
 89,  90,  91,  92,  93,  94,  94,  95,  96,  97,  98,  99,  99, 100, 101,
102, 103, 104, 105, 105, 106, 107, 108, 109, 110, 110, 111, 112, 113, 114,
115, 116, 116, 117, 118, 119, 120, 121, 122, 122, 123, 124, 125, 126, 127,
128, 128, 129, 130, 131, 132, 133, 133, 134, 135, 136, 137, 138, 139, 139,
140, 141, 142, 143, 144, 144, 145, 146, 147, 148, 149, 150, 150, 151, 152,
153, 154, 155, 156, 156, 157, 158, 159, 160, 161, 162, 162, 163, 164, 165,
166, 167, 167, 168, 169, 170, 171, 172, 173, 173, 174, 175, 176, 177, 178,
178, 179, 180, 181, 182, 183, 184, 184, 185, 186, 187, 188, 189, 190, 190,
191, 192, 193, 194, 195, 195, 196, 197, 198, 199, 200, 201, 201, 202, 203,
204, 205, 206, 207, 207, 208, 209, 210, 211, 212, 212, 213, 214, 215, 216,
217 };

#define raise_by_alpha(val) (RAISE_BY_ALPHA[val])
#define lower_by_alpha(val) (LOWER_BY_ALPHA[val])
```

```
static unsigned char SQUAROOT_TABLE[] = {

 0,  1,  1,  2,  2,  2,  2,  3,  3,  3,  3,  3,  3,  4,  4,  4,  4,  4,  4,
 4,  4,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  6,  6,  6,  6,  6,  6,  6,
 6,  6,  6,  6,  6,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,
 8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  9,  9,  9,
 9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9,  9, 10, 10, 10, 10,
10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 11, 11, 11,
11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11, 11,
12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12, 12,
12, 12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13,
13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 13, 14, 14, 14, 14, 14, 14, 14,
14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14,
14, 14, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15,
15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 15, 16, 16, 16, 16, 16, 16,
16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16, 16,
16, 16, 16, 16, 16, 16, 16, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,
17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17, 17,
17, 17, 17, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18,
18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18, 18,
18, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19,
19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19, 19,
19, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20, 20,
20, 20, 20, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21,
21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21, 21,
21, 21, 21, 21, 21, 21, 21, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22,
22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 22, 23, 23, 23, 23, 23, 23,
23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23, 23,
23, 23, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 25, 25, 25, 25, 25, 25, 25,
25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25, 25,
25, 25, 25, 25, 25, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26,
26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26,
26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26, 26,
27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,
27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27,
27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 27, 28, 28, 28,
28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28,
28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28,
28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 28, 29, 29, 29, 29,
29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29,
29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 29, 30, 30, 30,
30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30, 30,
31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31,
31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31,
```

```
31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31, 31,
31, 31, 31, 31, 31, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32,
32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32, 32 };

#define squaroot(val) (SQUAROOT_TABLE[val])

short sin_theta[] = {
     0,    571,   1143,   1714,   2285,   2855,   3425,   3993,   4560,
  5126,   5690,   6252,   6812,   7371,   7927,   8480,   9032,   9580,
 10125,  10668,  11207,  11743,  12275,  12803,  13327,  13848,  14364,
 14876,  15383,  15886,  16384,  16876,  17364,  17846,  18323,  18794,
 19260,  19720,  20173,  20621,  21062,  21497,  21926,  22347,  22762,
 23170,  23571,  23964,  24351,  24730,  25101,  25465,  25821,  26169,
 26509,  26841,  27165,  27481,  27788,  28087,  28377,  28659,  28932,
 29196,  29451,  29697,  29935,  30163,  30381,  30591,  30791,  30982,
 31164,  31336,  31498,  31651,  31794,  31928,  32051,  32165,  32270,
 32364,  32449,  32523,  32588,  32643,  32688,  32723,  32748,  32763,
 32767,  32763,  32748,  32723,  32688,  32643,  32588,  32523,  32449,
 32364,  32270,  32165,  32051,  31928,  31794,  31651,  31498,  31336,
 31164,  30982,  30791,  30591,  30381,  30163,  29935,  29697,  29451,
 29196,  28932,  28659,  28377,  28087,  27788,  27481,  27165,  26841,
 26509,  26169,  25821,  25465,  25101,  24730,  24351,  23964,  23571,
 23170,  22762,  22347,  21926,  21497,  21062,  20621,  20173,  19720,
 19260,  18794,  18323,  17846,  17364,  16876,  16384,  15886,  15383,
 14876,  14364,  13848,  13327,  12803,  12275,  11743,  11207,  10668,
 10125,   9580,   9032,   8480,   7927,   7371,   6812,   6252,   5690,
  5126,   4560,   3993,   3425,   2855,   2285,   1714,   1143,    571,
     0,   -571,  -1143,  -1714,  -2285,  -2855,  -3425,  -3993,  -4560,
 -5126,  -5690,  -6252,  -6812,  -7371,  -7927,  -8480,  -9032,  -9580,
-10125, -10668, -11207, -11742, -12275, -12803, -13327, -13848, -14364,
-14876, -15383, -15886, -16383, -16876, -17364, -17846, -18323, -18794,
-19260, -19720, -20173, -20621, -21062, -21497, -21926, -22347, -22762,
-23170, -23571, -23964, -24351, -24730, -25101, -25465, -25821, -26169,
-26509, -26841, -27165, -27481, -27788, -28087, -28377, -28659, -28932,
-29196, -29451, -29697, -29935, -30163, -30381, -30591, -30791, -30982,
-31164, -31336, -31498, -31651, -31794, -31928, -32051, -32165, -32270,
-32364, -32449, -32523, -32588, -32643, -32688, -32723, -32748, -32763,
-32768, -32763, -32748, -32723, -32688, -32643, -32588, -32523, -32449,
-32364, -32270, -32165, -32051, -31928, -31794, -31651, -31498, -31336,
-31164, -30982, -30791, -30591, -30381, -30163, -29935, -29697, -29451,
-29196, -28932, -28659, -28377, -28087, -27788, -27481, -27165, -26841,
-26509, -26169, -25821, -25465, -25101, -24730, -24351, -23965, -23571,
-23170, -22762, -22347, -21926, -21497, -21062, -20621, -20173, -19720,
-19260, -18794, -18323, -17846, -17364, -16876, -16384, -15886, -15383,
-14876, -14364, -13848, -13327, -12803, -12275, -11743, -11207, -10668,
-10125,  -9580,  -9032,  -8480,  -7927,  -7371,  -6812,  -6252,  -5690,
 -5126,  -4560,  -3993,  -3425,  -2855,  -2285,  -1714,  -1143,   -571};

short cos_theta[] = {
 32767,  32763,  32748,  32723,  32688,  32643,  32588,  32523,  32449,
 32364,  32270,  32165,  32051,  31928,  31794,  31651,  31498,  31336,
 31164,  30982,  30791,  30591,  30381,  30163,  29935,  29697,  29451,
 29196,  28932,  28659,  28377,  28087,  27788,  27481,  27165,  26841,
 26509,  26169,  25821,  25465,  25101,  24730,  24351,  23964,  23571,
```

```
 23170,  22762,  22347,  21926,  21497,  21062,  20621,  20173,  19720,
 19260,  18794,  18323,  17846,  17364,  16876,  16384,  15886,  15383,
 14876,  14364,  13848,  13327,  12803,  12275,  11743,  11207,  10668,
 10125,   9580,   9032,   8480,   7927,   7371,   6812,   6252,   5690,
  5126,   4560,   3993,   3425,   2855,   2285,   1714,   1143,    571,
     0,   -571,  -1143,  -1714,  -2285,  -2855,  -3425,  -3993,  -4560,
 -5126,  -5690,  -6252,  -6812,  -7371,  -7927,  -8480,  -9032,  -9580,
-10125, -10668, -11207, -11743, -12275, -12803, -13327, -13848, -14364,
-14876, -15383, -15886, -16383, -16876, -17364, -17846, -18323, -18794,
-19260, -19720, -20173, -20621, -21062, -21497, -21926, -22347, -22762,
-23170, -23571, -23964, -24351, -24730, -25101, -25465, -25821, -26169,
-26509, -26841, -27165, -27481, -27788, -28087, -28377, -28659, -28932,
-29196, -29451, -29697, -29935, -30163, -30381, -30591, -30791, -30982,
-31164, -31336, -31498, -31651, -31794, -31928, -32051, -32165, -32270,
-32364, -32449, -32523, -32588, -32643, -32688, -32723, -32748, -32763,
-32768, -32763, -32748, -32723, -32688, -32643, -32588, -32523, -32449,
-32364, -32270, -32165, -32051, -31928, -31794, -31651, -31498, -31336,
-31164, -30982, -30791, -30591, -30381, -30163, -29935, -29697, -29451,
-29196, -28932, -28659, -28377, -28087, -27788, -27481, -27165, -26841,
-26509, -26169, -25821, -25465, -25101, -24730, -24351, -23965, -23571,
-23170, -22762, -22347, -21926, -21497, -21062, -20621, -20173, -19720,
-19260, -18794, -18323, -17846, -17364, -16876, -16384, -15886, -15383,
-14876, -14364, -13848, -13327, -12803, -12275, -11743, -11207, -10668,
-10125,  -9580,  -9032,  -8480,  -7927,  -7371,  -6812,  -6252,  -5690,
 -5126,  -4560,  -3993,  -3425,  -2855,  -2285,  -1714,  -1143,   -571,
     0,    571,   1143,   1714,   2285,   2855,   3425,   3993,   4560,
  5126,   5690,   6252,   6812,   7371,   7927,   8480,   9032,   9580,
 10125,  10668,  11207,  11742,  12275,  12803,  13327,  13848,  14364,
 14876,  15383,  15886,  16383,  16876,  17364,  17846,  18323,  18794,
 19260,  19720,  20173,  20621,  21062,  21497,  21926,  22347,  22762,
 23170,  23571,  23964,  24351,  24730,  25101,  25465,  25821,  26169,
 26509,  26841,  27165,  27481,  27788,  28087,  28377,  28659,  28932,
 29196,  29451,  29697,  29935,  30163,  30381,  30591,  30791,  30982,
 31164,  31336,  31498,  31651,  31794,  31928,  32051,  32165,  32270,
 32364,  32449,  32523,  32588,  32643,  32688,  32723,  32748,  32763};

#define r_sin_theta(r,theta) ((sin_theta[theta % 360] * r + 16383) >> 15)
#define r_cos_theta(r,theta) ((cos_theta[theta % 360] * r + 16383) >> 15)

/*the largest range (in cells) we're willing to entertain*/
#define MAXRANGE 44

/*********************************************************************
*                                                                    *
*       range_to_cell_list takes a vector and a tail location and    *
*       converts it to a list of cells through which the vector      *
*       passes.  Note that the technique used does not guarantee     *
*       that each cell in the list only appears once (it doesn't     *
*       really seem necessary for this application).                 *
*                                                                    *
*********************************************************************/

qhd range_to_cell_list(range, direction, root)
```

```
int range;
int direction;
point root;

{

  qbuck bptr;
  point *new_point;
  int dist;

  if (range > MAXRANGE)
    return(cell_list);
  for (dist = 0; dist <= range; dist++)
  {
    new_point = (point*) dequeue(free_cell_list);
    new_point->x = root.x + r_cos_theta(dist,direction);
    new_point->y = root.y + r_sin_theta(dist,direction);
    enqueue(cell_list,new_point);
  }
  return(cell_list);
}

/*****************************************************************
*                                                               *
*       get_length returns the length of the edge passed in.    *
*                                                               *
*****************************************************************/

int get_length(e)

edge e;

{

  return(squaroot((e.head.x - e.tail.x) * (e.head.x - e.tail.x)
              + (e.head.y - e.tail.y) * (e.head.y - e.tail.y)));

}

/*****************************************************************
*                                                               *
*       fixed_alpha_update is a simple occupancy grid update rule *
*       which, in the case that we are _not_ at the end of the range *
*       vector (as specified by the arguments), multiplies the    *
*       probability of occupancy by an alpha less than one, and in *
*       the case where we _are_ at the end of the range vector,   *
*       raises the probability of occupancy by multiplying the    *
*       distance from certainty by a different alpha (also less than *
*       one.                                                    *
*                                                               *
*****************************************************************/

void fixed_alpha_update(grid,cgrid,tgrid,here,theta,update_point,is_range)
```

```
matrix grid,cgrid,tgrid;
point here;
int theta;
point update_point;
int is_range;

{

  if (is_range)
  {
    matrix_set(grid,update_point.x,update_point.y,
            raise_by_alpha(matrix_ref(grid,update_point.x,update_point.y)));
  }
  else
  {
    matrix_set(grid,update_point.x,update_point.y,
            lower_by_alpha(matrix_ref(grid,update_point.x,update_point.y)));
  }
}

#define INFINITY 35

/********************************************************************
*                                                                   *
*       update_grid takes a list of grid cells and an update function *
*       and applies the update function to each cell in the list.     *
*       Here is where the determination of whether a given cell is    *
*       at the end of the vector (approximately) is made.             *
*                                                                   *
********************************************************************/

void update_grid(grid,cgrid,tgrid,here,theta,point_list,update_function)

matrix grid,cgrid,tgrid;
point here;
int theta;
qhd point_list;
void (*update_function)();

{

  qbuck bptr;
  int is_range;

  for (bptr = point_list->head; bptr != NULL; bptr = bptr->next)
  {
    if ((here.x != ((point*)bptr->qdat)->x)
        || (here.y != ((point*)bptr->qdat)->y))
    {
      is_range = (((bptr == point_list->tail) ||
                   (bptr->next == point_list->tail)) &&
                  (point_list->qlen < INFINITY));
      update_function(grid,cgrid,tgrid,here,theta,*((point*)bptr->qdat),
                      is_range);
```

```
    }
  }
}

#define RANGE_SCALE 50
#define LEFT_RANGE_SCALE 53
#define RIGHT_RANGE_SCALE 53

#define HALF_CELL (LEFT_RANGE_SCALE / 2)

#define DIST_SURFACE_TO_CENTER 50

/*********************************************************************
*                                                                    *
*       left_update takes as input a SCHEME list of arguments which   *
*       occur in the order (left to right in the SCHEME call) here.x  *
*       and here.y (the location of the robot), theta (the heading of *
*       the ranger when the readings were taking), range.left, and    *
*       range.right (the two components of a range reading).  Note    *
*       that left_update ignores range.right, while right_update      *
*       (which takes the same SCHEME list as input) ignores           *
*       range.left.  After extracting the pertinent arguments from    *
*       the SCHEME list into C variables, left_update constructs      *
*       the vector which represents the actual path of the laser in   *
*       the world, scaled by constants pertaining to the cell size    *
*       and other random calibration constants (caused by the         *
*       ranger's tendency to get uncalibrated), and performs a map    *
*       update based on that vector.  Note that if we want to change  *
*       the name of the update function being used, here's the place  *
*       to do it (in the call to update_grid, as well as in the same  *
*       call in right_update).                                        *
*                                                                    *
*********************************************************************/

scheme_value left_update(nargs, args)

long nargs;
scheme_value *args;

{

  int theta,range,i,ql;
  qhd left_list;
  point here,root;

  here.x = EXTRACT_FIXNUM(args[4]) / RANGE_SCALE;
  here.y = EXTRACT_FIXNUM(args[3]) / RANGE_SCALE;
  theta =  EXTRACT_FIXNUM(args[2]);
  range =  EXTRACT_FIXNUM(args[1]);

  if ((range < 300) || (range > 3000))
    return(SCHTRUE);

  /* the range we want to use for computation is the distance from the line
```

```
    passing through the center of the robot to the point where the left
    (resp. right) beam hits the world.  We add one more because we raise
    the probability of occupancy of the last _two_ pixels on the edge.  */
  range = (range + IMAGING_SURFACE_OFFSET + HALF_CELL)/LEFT_RANGE_SCALE + 1;
  root.x = here.x + (r_cos_theta(3, (theta + 90) % 360) / 2);
  root.y = here.y + (r_sin_theta(3, (theta + 90) % 360) / 2);

  left_list = range_to_cell_list(range,theta,root);
  update_grid(grid,cgrid,tgrid,here,theta,left_list,fixed_alpha_update);
  ql = left_list->qlen;
  for (i = 0; i < ql; i++)
    enqueue(free_cell_list,dequeue(left_list));
  return(ENTER_FIXNUM(range));
}

/*******************************************************************
*                                                                 *
*       right_update does the same thing as left_update, but for the  *
*       right range reading.  Note that right_update omits arg. 1.    *
*                                                                 *
*******************************************************************/

scheme_value right_update(nargs, args)

long nargs;
scheme_value *args;

{

  int theta,range,i,ql;
  qhd right_list;
  point here,root;

  here.x = EXTRACT_FIXNUM(args[4]) / RANGE_SCALE;
  here.y = EXTRACT_FIXNUM(args[3]) / RANGE_SCALE;
  theta =  EXTRACT_FIXNUM(args[2]);
  range =  EXTRACT_FIXNUM(args[0]);

  if ((range < 300) || (range > 3000))
    return(SCHTRUE);

  range = (range + IMAGING_SURFACE_OFFSET + HALF_CELL)/RIGHT_RANGE_SCALE +1;
  root.x = here.x - (r_cos_theta(3, (theta + 90) % 360) / 2);
  root.y = here.y - (r_sin_theta(3, (theta + 90) % 360) / 2);

  right_list = range_to_cell_list(range,theta,root);
  update_grid(grid,cgrid,tgrid,here,theta,right_list,fixed_alpha_update);
  ql = right_list->qlen;
  for (i = 0; i < ql; i++)
    enqueue(free_cell_list,dequeue(right_list));
  return(ENTER_FIXNUM(range));
}

/*******************************************************************
```

```
*                                                                   *
*       map_update (which is really just a front end for left_update  *
*       and right_update) is the SCHEME interface for updating the    *
*       map given a pair of range readings.                           *
*                                                                   *
*******************************************************************/

scheme_value map_update(nargs,args)

long nargs;
scheme_value *args;

{

  (void)left_update(nargs,args);
  return(right_update(nargs,args));
}

/*******************************************************************
*                                                                   *
*       This main calls the routine that initializes the main matrix  *
*       structures, does some other miscellaneous initialization of   *
*       storage, and invokes the SCHEME virtual machine.              *
*                                                                   *
*******************************************************************/

int main(argc,argv)

int argc;
char **argv;
{

  int i;
  qhd bucket_list;
  point *new_point;

/*********************************************
*                                           *
*  This next piece of code stolen from the    *
*  file ~robotlab/rscheme/rscheme/68k/stub.c  *
*  is the code to invoke the SCHEME VM.  Be   *
*  sure that the call address remains correct *
*  for the VM that's running.                 *
*                                           *
*********************************************/

  /* Without the volatile declaration, gcc will optimize this to a point
     where gas will be unable to cope with the result.  Specifically, gas
     doesn't assemble
        jbsr 557390
     properly for the 68000. */

  volatile long bar = 0x8829e;  /*Should be _main as in file rom-server.nm*/
/*  volatile long bar = 0x0129e; /* for RAM downloaded version */
```

```
  int (*foo)() = (int (*)()) bar;

/*********************************************
*                                             *
*  The rest of the code is mine, except for   *
*  the final return, which is the line that   *
*  actually invokes the VM.                   *
*                                             *
*********************************************/
  _frontier = (char**) 0xffc;
  printf("Initializing map program.\n\r");
  make_grids();                             /*allocate space for the maps */
  bucket_list = (qhd) himalloc(sizeof(struct queuehead));
  bucket_list->qlen = 0;
  bucket_list->head = NULL;
  bucket_list->tail = NULL;
  freelist = NULL;
  for (i = 0; i < INITBUCKETS; i++)         /*and a bunch of queue buckets*/
    enqueue(bucket_list,&i);
  for (i = 0; i < INITBUCKETS; i++)
    (void) dequeue(bucket_list);
  cell_list = (qhd) himalloc(sizeof(struct queuehead));
  cell_list->qlen = 0;
  cell_list->head = NULL;
  cell_list->tail = NULL;
  free_cell_list = (qhd) himalloc(sizeof(struct queuehead));
  free_cell_list->qlen = 0;
  free_cell_list->head = NULL;
  free_cell_list->tail = NULL;
  for (i = 0; i < MAXPOINTS; i++)           /* and a bunch of points. */
  {
    new_point = (point*) himalloc(sizeof(struct pt));
    enqueue(free_cell_list,new_point);
  }

#ifdef WITH_LOCATE
  localization_initialization();
#endif
#ifdef WITH_FLOOD
  flood_initialization();
#endif

/*    *_frontier = (char *) 0x1f000; /* frontier value for RAM version */
  *_frontier = (char *) 0x1000; /*make the new frontier value(the one SCHEME
                          will see) be 0x1000, so I can run C stuff in the
                          extra memory and SCHEME stuff in built-in memory.*/
  return (*foo)(argc, argv);
}
```

C.3 flood.c

```
#include <stdio.h>
```

```
#include "matrix-accessors.h"
#include "geometry.h"
#include "queue.h"
/*#include <math.h>*/
#include "/usr/u/robotlab/rscheme/s48/scheme48.h"

#define max(x,y) ((x > y) ? (x) : (y))  /* you always need these */
#define min(x,y) ((x > y) ? (y) : (x))
#define abs(x) ((x>0)?(x):(-x))
#define rint(x) (x)
#define between(x,y,z) ((x >= y) && (x < z))

#define MAXPOINTS (256)

extern matrix grid;
extern matrix threshmap;
extern matrix single_vector_locmap;
extern matrix localization_map;
extern matrix temp_map;

extern qhd cell_list, free_cell_list;

matrix flood_map;
matrix cspace_map;

/**********************************************************************
*                                                                     *
*       flood_initialization sets up and initializes all of the grids *
*       we need for the flood-fill planner, including but not limited *
*       to a matrix to construct the c-space in and a matrix to perform *
*       the floodfill in.                                             *
*                                                                     *
**********************************************************************/

void flood_initialization()

{

  int i,j;

  printf("Initializing flood-fill data structures.\n\r");

  flood_map.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  flood_map.xmin = GRID_MIN;
  flood_map.xmax = GRID_MIN + GRID_SIZE;
  flood_map.ymin = GRID_MIN;
  flood_map.ymax = GRID_MIN + GRID_SIZE;

  cspace_map.mdata = (matrix_data) himalloc(sizeof(matrix_space));
  cspace_map.xmin = GRID_MIN;
  cspace_map.xmax = GRID_MIN + GRID_SIZE;
  cspace_map.ymin = GRID_MIN;
  cspace_map.ymax = GRID_MIN + GRID_SIZE;
```

```
}

/*********************************************************************
*                                                                    *

*                                                                    *
*********************************************************************/

void flood(goalx,goaly)

int goalx,goaly;

{

  int i,j;
  point *current,*new;
  int currval, newval;
  qhd point_list;

  point_list = (qhd) himalloc(sizeof(struct queuehead));
  point_list->qlen = 0;
  point_list->head = NULL;
  point_list->tail = NULL;
  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
    {
      if ((*(cspace_map.mdata))[i][j] == 0)
        (*(flood_map.mdata))[i][j] = 255;
      else
        (*(flood_map.mdata))[i][j] = 0;
    }
  goalx -= cspace_map.xmin;   /* the user specifies goalx,goaly in world */
  goaly -= cspace_map.ymin;   /* coordinates, but we want them in matrix */
                              /* coordinates, so subtract off xmin,ymin. */
  new = (point*) dequeue(free_cell_list);
  new->x = goalx;
  new->y = goaly;
  enqueue(point_list,new);
  (*(flood_map.mdata))[goalx][goaly] = 1;

  while (point_list->qlen > 0)
  {
    current = (point*)dequeue(point_list);
    currval = (*(flood_map.mdata))[current->x][current->y];
    newval = currval + 1;
    if (newval == 128) /* doing three-valued version of floodfill, so only*/
      newval = 1;      /* values 1, 2, and 3.  but 128-valued is prettier.*/
    if (current->x > 0)
    {
      if ((current->y > 0) &&
          ((*(flood_map.mdata))[current->x - 1][current->y - 1] == 0))
      {
        (*(flood_map.mdata))[current->x - 1][current->y - 1] = newval;
```

```
    new = (point*) dequeue(free_cell_list);
    new->x = current->x - 1;
    new->y = current->y - 1;
    enqueue(point_list,new);
  }
  if ((*(flood_map.mdata))[current->x - 1][current->y] == 0)
  {
    (*(flood_map.mdata))[current->x - 1][current->y] = newval;
    new = (point*) dequeue(free_cell_list);
    new->x = current->x - 1;
    new->y = current->y;
    enqueue(point_list,new);
  }
  if ((current->y < GRID_SIZE) &&
      ((*(flood_map.mdata))[current->x - 1][current->y + 1] == 0))
  {
    (*(flood_map.mdata))[current->x - 1][current->y + 1] = newval;
    new = (point*) dequeue(free_cell_list);
    new->x = current->x - 1;
    new->y = current->y + 1;
    enqueue(point_list,new);
  }
}
if ((current->y > 0) &&
    ((*(flood_map.mdata))[current->x][current->y - 1] == 0))
{
  (*(flood_map.mdata))[current->x][current->y - 1] = newval;
  new = (point*) dequeue(free_cell_list);
  new->x = current->x;
  new->y = current->y - 1;
  enqueue(point_list,new);
}
if ((current->y < GRID_SIZE) &&
    ((*(flood_map.mdata))[current->x][current->y + 1] == 0))
{
  (*(flood_map.mdata))[current->x][current->y + 1] = newval;
  new = (point*) dequeue(free_cell_list);
  new->x = current->x;
  new->y = current->y + 1;
  enqueue(point_list,new);
}
if (current->x < GRID_SIZE)
{
  if ((current->y > 0) &&
      ((*(flood_map.mdata))[current->x + 1][current->y - 1] == 0))
  {
    (*(flood_map.mdata))[current->x + 1][current->y - 1] = newval;
    new = (point*) dequeue(free_cell_list);
    new->x = current->x + 1;
    new->y = current->y - 1;
    enqueue(point_list,new);
  }
  if ((*(flood_map.mdata))[current->x + 1][current->y] == 0)
  {
```

```
        (*(flood_map.mdata))[current->x + 1][current->y] = newval;
        new = (point*) dequeue(free_cell_list);
        new->x = current->x + 1;
        new->y = current->y;
        enqueue(point_list,new);
      }
      if ((current->y < GRID_SIZE) &&
          ((*(flood_map.mdata))[current->x + 1][current->y + 1] == 0))
      {
        (*(flood_map.mdata))[current->x + 1][current->y + 1] = newval;
        new = (point*) dequeue(free_cell_list);
        new->x = current->x + 1;
        new->y = current->y + 1;
        enqueue(point_list,new);
      }
    }
    enqueue(free_cell_list,current);
  }
}

/***********************************************************************
*                                                                      *
*       For modularity's sake (and so I can use it elsewhere), I want   *
*       to be able to use the circular flood in a more general fashion *
*       than originally specified in make_c_space above.  This is an    *
*       attempt in that direction.                                     *
*                                                                      *
***********************************************************************/

static unsigned char CIRCLE_NEIGHBORS[] = { 4, 4, 4, 8, 8, 4, 4, 8, 4, 8,
                                            4, 4, 4, 8, 4, 8, 4, 4, 4, 4};

void pseudo_circle_fill(in_map,grown_map,epsilon)

matrix in_map,grown_map;
int epsilon;

{

  qhd cspace_point_list;
  int i,j,k;
  point *grow_point, *new_point;
  int currval;
  int num_neighbors;

  /* for the grown obstacle space, we want to start with objects at 0,
     untouched space at 255. We assume the input has objects at 0, freespace
     at 1.  That's the result we tend to get from threshold_graymap. */
  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
      (*(grown_map.mdata))[i][j] = - (*(in_map.mdata))[i][j];
  cspace_point_list = (qhd) himalloc(sizeof(struct queuehead));
  cspace_point_list->qlen = 0;
  cspace_point_list->head = NULL;
```

```
  cspace_point_list->tail = NULL;

  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
      if ((*(grown_map.mdata))[i][j] == 0)
      {
        num_neighbors = 0;
        for (k = 0; k < 4; k++)
          if (between(i + FOUR_NEIGHBOR_X[k],0,GRID_SIZE) &&
              between(j + FOUR_NEIGHBOR_Y[k],0,GRID_SIZE) &&
              ((*(grown_map.mdata))[i + FOUR_NEIGHBOR_X[k]]
                                   [j + FOUR_NEIGHBOR_Y[k]] == 255))
            num_neighbors++;
        if (num_neighbors != 0)
        {
          grow_point = dequeue(free_cell_list);
          grow_point->x = i;
          grow_point->y = j;
          enqueue(cspace_point_list,grow_point);
#ifdef DEBUG
          if (!(cspace_point_list->qlen % 100))
            printf("q-len:  %d\n\r",cspace_point_list->qlen);
#endif
        }
      }
  while (cspace_point_list->qlen != 0)
  {
    grow_point = dequeue(cspace_point_list);
    currval = (*(grown_map.mdata))[grow_point->x][grow_point->y];
    if (currval < epsilon)
    {
      if (CIRCLE_NEIGHBORS[currval] == 4)
      {
        for (i = 0; i < 4; i++)
          if (between(grow_point->x + FOUR_NEIGHBOR_X[i],0,GRID_SIZE) &&
              between(grow_point->y + FOUR_NEIGHBOR_Y[i],0,GRID_SIZE) &&
              ((*(grown_map.mdata))[grow_point->x + FOUR_NEIGHBOR_X[i]]
                                 [grow_point->y + FOUR_NEIGHBOR_Y[i]] == 255))
          {
            (*(grown_map.mdata))[grow_point->x + FOUR_NEIGHBOR_X[i]]
                                 [grow_point->y + FOUR_NEIGHBOR_Y[i]]
              = (*(grown_map.mdata))[grow_point->x][grow_point->y]+1;
            new_point = dequeue(free_cell_list);
            new_point->x = grow_point->x + FOUR_NEIGHBOR_X[i];
            new_point->y = grow_point->y + FOUR_NEIGHBOR_Y[i];
            enqueue(cspace_point_list,new_point);
          }
      if (free_cell_list->qlen == 0)
        printf("free queue is empty.\n\r");
      }
      else
      {
        for (i = 0; i < 8; i++)
          if (between(grow_point->x + EIGHT_NEIGHBOR_X[i],0,GRID_SIZE) &&
```

```
              between(grow_point->y + EIGHT_NEIGHBOR_Y[i],0,GRID_SIZE) &&
              ((*(grown_map.mdata))[grow_point->x + EIGHT_NEIGHBOR_X[i]]
                        [grow_point->y + EIGHT_NEIGHBOR_Y[i]] == 255))
          {
            (*(grown_map.mdata))[grow_point->x + EIGHT_NEIGHBOR_X[i]]
                                 [grow_point->y + EIGHT_NEIGHBOR_Y[i]]
              = (*(grown_map.mdata))[grow_point->x][grow_point->y]+1;
            new_point = dequeue(free_cell_list);
            new_point->x = grow_point->x + EIGHT_NEIGHBOR_X[i];
            new_point->y = grow_point->y + EIGHT_NEIGHBOR_Y[i];
            enqueue(cspace_point_list,new_point);
          }
      }
#ifdef DEBUG
        if (!(cspace_point_list->qlen % 100))
          printf("q-len:  %d\n\r",cspace_point_list->qlen);
#endif
      if (free_cell_list->qlen == 0)
        printf("free queue is empty.\n\r");
    }
    enqueue(free_cell_list,grow_point);
  }
  for (i = 0; i < GRID_SIZE; i++)
    for (j = 0; j < GRID_SIZE; j++)
      (*(grown_map.mdata))[i][j] =((*(grown_map.mdata))[i][j] == 255)?255:0;
}

/*********************************************************************
*                                                                    *
*       make_c_space will be the routine that takes a graymap,       *
*       thresholds it, convolves it with an epsilon ball of the      *
*       appropriate radius for the robot size (currently 4 (5?) --   *
*       the robot radius is 15cm (3 cells) plus another cell or two  *
*       for the bumper.                                              *
*                                                                    *
*********************************************************************/

#define ROBOT_EPSILON 4
#define CSPACE_THRESH 100

void make_c_space()

{

  threshold_graymap(grid,CSPACE_THRESH,threshmap);
  pseudo_circle_fill(threshmap,cspace_map,ROBOT_EPSILON);

}

/*********************************************************************
*                                                                    *
*       surf takes a start position and returns a queue whose contents *
*       is the list of points on the shortest path from the specified  *
*       start to the goal point specified in the last call to flood.   *
```

```
*                                                                        *
**************************************************************************/

qhd surf(startx,starty)

int startx, starty;

{

  qhd surf_path;
  point *current;
  int currx,curry,currval;
  int flood_x_size,flood_y_size;

  surf_path = (qhd) himalloc(sizeof(struct queuehead));
  surf_path->qlen = 0;
  surf_path->head = NULL;
  surf_path->tail = NULL;

  currx = startx - flood_map.xmin;
  curry = starty - flood_map.ymin;
  flood_x_size = flood_map.xmax - flood_map.xmin;
  flood_y_size = flood_map.ymax - flood_map.ymin;
  /* if the starting point is outside the world, inside an object, or in a
     disconnected component from the goal, return the empty queue.  */
  if ((startx < flood_map.xmin) || (startx >= flood_map.xmax) ||
      (starty < flood_map.ymin) || (starty >= flood_map.ymax) ||
      ((*(flood_map.mdata))[currx][curry] == 0) ||
      ((*(flood_map.mdata))[currx][curry] == 255))
    return(surf_path);
  /* otherwise, start surfing (looking for a lower-potential (mod-wise)
     neighbor and going to it). */
  while (1) /* we'll loop indefinitely, breaking out of the loop when we */
  {         /* find ourselves at a local minimum. */
    currval = (*(flood_map.mdata))[currx][curry];
    current = dequeue(free_cell_list);
    current->x = currx + flood_map.xmin;  /* first, put the current point */
    current->y = curry + flood_map.ymin;  /* on the surf path. */
    enqueue(surf_path,current);
    printf("%3d %3d\n\r",current->x,current->y);
    /* check first to see if a 4-connected neighbor has a lower potential*/
    if ((currx > 0) && ((*(flood_map.mdata))[currx-1][curry] < currval))
      currx--;
    else if ((curry > 0) &&((*(flood_map.mdata))[currx][curry-1] < currval))
      curry--;
    else if ((currx <= flood_x_size) &&
             ((*(flood_map.mdata))[currx+1][curry] < currval))
      currx++;
    else if ((curry <= flood_y_size) &&
             ((*(flood_map.mdata))[currx][curry+1] < currval))
      curry++;
    /* if not, see if one of the other 8-connected neighbors does. */
    else if ((currx > 0) && (curry > 0) &&
             ((*(flood_map.mdata))[currx-1][curry-1] < currval))
```

```
    {
      currx--;
      curry--;
    }
    else if ((currx > 0) && (curry <= flood_y_size) &&
             ((*(flood_map.mdata))[currx-1][curry+1] < currval))
    {
      currx--;
      curry++;
    }
    else if ((currx <= flood_x_size) && (curry > 0) &&
             ((*(flood_map.mdata))[currx+1][curry-1] < currval))
    {
      currx++;
      curry--;
    }
    else if ((currx <= flood_x_size) && (curry <= flood_y_size) &&
             ((*(flood_map.mdata))[currx+1][curry+1] < currval))
    {
      currx++;
      curry++;
    }
    else
      return(surf_path);
  }
}


/**********************************************************************
*                                                                     *
*       sch_cspace is the SCHEME interface to the make_cspace routine.  *
*       It takes no arguments, and only exists as a separate          *
*       entrypoint because making the cspace seems to be the most     *
*       expensive part of the floodfill, and doesn't really need to   *
*       be redone for every goal.                                     *
*                                                                     *
**********************************************************************/

scheme_value sch_cspace(nargs,args)

long nargs;
scheme_value *args;

{
  make_c_space();
}
/**********************************************************************
*                                                                     *
*       sch_flood is the SCHEME interface to the flood_fill routine.   *
*       call it with args goalx,goaly.                                *
*                                                                     *
**********************************************************************/

scheme_value sch_flood(nargs,args)
```

```
long nargs;
scheme_value *args;

{

  int goalx,goaly;

  goalx = EXTRACT_FIXNUM(args[1]);
  goaly = EXTRACT_FIXNUM(args[0]);
  flood(goalx,goaly);
  return(SCHTRUE);

}

/*********************************************************************
*                                                                    *
*       sch_surf is the SCHEME interface to the surf routine.       *
*       call it with args startx,starty,x_vector,y_vector,vect_len, *
*       where x_vector and y_vector are scheme vectors into which to *
*       put the surf path, and vect_len is the length of the vectors. *
*                                                                    *
*********************************************************************/

scheme_value sch_surf(nargs,args)

long nargs;
scheme_value *args;

{

  qhd surf_path;
  int startx,starty;
  point *current_point;
  scheme_value x_vector, y_vector;
  int vect_len,index;

  startx = EXTRACT_FIXNUM(args[4]);
  starty = EXTRACT_FIXNUM(args[3]);
  x_vector = args[2];
  y_vector = args[1];
  vect_len = EXTRACT_FIXNUM(args[0]);

  surf_path = surf(startx,starty);
  index = 0;
  while ((surf_path->qlen > 0) && (index < vect_len))
  {
    current_point = dequeue(surf_path);
    vector_ref(x_vector,index) = ENTER_FIXNUM(current_point->x);
    vector_ref(y_vector,index++) = ENTER_FIXNUM(current_point->y);
    enqueue(free_cell_list,current_point);
  }
  while (surf_path->qlen > 0)
  {
```

```
    current_point = dequeue(surf_path);
    enqueue(free_cell_list,current_point);
  }
  return(ENTER_FIXNUM(index));

}
```

C.4 cnav.scm

```
;;; nav.scm
;;;
;;; given a decent rangefinder, use a combination of raster-oriented and
;;; geometric-based techniques to do map building and navigation

;;; define the structure "point" to be '(x y).  Define its constructor,
;;; (make-point x y), its accessors, (x point) and (y point), and its
;;; modifiers, (set-x! point xval) and (set-y! point yval).

(define (make-point xv yv) (list xv yv))
(define (x p) (car p))
(define (y p) (cadr p))
(define (set-x! point xv) (set-car! point xv))
(define (set-y! point yv) (set-car! (cdr point) yv))

;;; define the structure "edge" to be '(head tail eqn angle).  Define
;;; its constructor, (make-edge . '(head tail eqn angle)) where eqn and
;;; angle are optional.  Define its accessors, (head edge), (tail edge),
;;; (eqn edge), and (angle edge), and its modifiers, (set-head! edge head),
;;; (set-tail! edge tail), (set-eqn! edge eqn), and (set-angle! edge angle).

(define (make-edge . arglist)
  (list (car arglist) (cadr arglist) (caddr arglist) (cadddr arglist))
)
(define (head edge) (car edge))
(define (tail edge) (cadr edge))
(define (eqn edge) (caddr edge))
(define (angle edge) (cadddr edge))
(define (set-head! edge hv) (set-car! edge hv))
(define (set-tail! edge tv) (set-car! (cdr edge) tv))
(define (set-eqn! edge eqv) (set-car! (cddr edge) eqv))
(define (set-angle! edge av) (set-car! (cdddr edge) av))

;;; define the structure "eqn" to be '(slope intercept).  Define its
;;; constructor, (make-eqn slope intercept), its accessors, (slope eqn) and
;;; (intercept eqn), and its modifiers, (set-slope! eqn slope) and
;;; (set-intercept! eqn intercept).

(define (make-eqn slope intercept) (list slope intercept))
(define (slope eqn) (car eqn))
(define (intercept eqn) (cadr eqn))
(define (set-slope! eqn slv) (set-car! eqn slv))
(define (set-intercept! eqn intv) (set-car! (cdr eqn) intv))
```

```
;;; get-length takes an edge and returns its length

(define (get-length this-edge)
  (sqrt (+ (* (- (x (head this-edge)) (x (tail this-edge)))
              (- (x (head this-edge)) (x (tail this-edge))))
           (* (- (y (head this-edge)) (y (tail this-edge)))
              (- (y (head this-edge)) (y (tail this-edge))))))
)

;;;r-sin-theta and r-cos-theta compute (look up, actually) rounded versions
;;;of those functions given r & theta (theta in degrees in the range [0,359]
;;;if you please).  *** may want to change to [-179,180] ***

(define (mymodulo n r)
  (let ((mv (modulo n r)))
    (if (< (+ mv mv) r) mv (- mv r))
  )
)

(define (r-sin-theta r theta)
  (round/ (* r (fixed-sine (mymodulo theta 360))) trig-scale)
)


(define (r-cos-theta r theta) (r-sin-theta r (+ theta 90)))

(define *current-x-posn* 0)
(define *current-y-posn* 0)
(define *current-heading* 0)

(define (contact? dummy-arg) (> (read-bumpers) 0))

(define (turnto heading)
  (let ((turnamt (- heading *current-heading*))
        (current-where (rotate-where)))
    (if (> (abs turnamt) 2)
        (tolerate-impediments (rotate 'by turnamt)))
    (set! *current-heading* (+ *current-heading*
                               (- (rotate-where) current-where)))
  )
)

(define (goto place . rest)
  (let ((xdiff (- (x place) *current-x-posn*))
        (ydiff (- (y place) *current-y-posn*))
        (turnamt 0)
        (wanted-angle 0)
        (current-trans-dist (translate-where))
        (new-trans-dist 0)
        (new-angle 0)
        (distance 0))
    (if (and (= xdiff 0) (= ydiff 0)) (set! wanted-angle *current-heading*)
        (set! wanted-angle (fixed-atan2 ydiff xdiff)))
```

```
    (set! turnamt (- wanted-angle *current-heading*))
    (if (> turnamt 180) (set! turnamt (- turnamt 360)))
    (if (< turnamt -180) (set! turnamt (+ turnamt 360)))
    (if (> turnamt 180) (set! turnamt (- turnamt 360)))
    (if (< turnamt -180) (set! turnamt (+ turnamt 360)))

    (set! distance (isqrt (+ (* xdiff xdiff) (* ydiff ydiff))))
    (if (> turnamt 90) (begin (set! turnamt (- turnamt 180))
                              (set! distance (- distance))))
    (if (< turnamt -90) (begin (set! turnamt (+ turnamt 180))
                              (set! distance (- distance))))
    (if (> (abs turnamt) 2) (turnto (+ *current-heading* turnamt)))
    (if (contact? 0)
        (if (null? rest)
            (tolerate-impediments (translate 'by distance))
            (tolerate-impediments (translate 'by distance
                                             (car rest) (cadr rest)))
        )
        (if (null? rest)
            (tolerate-impediments (translate 'by distance 'until contact?))
            (tolerate-impediments (translate 'by distance (car rest)
                                       (cadr rest) 'until contact?))
        )
    )
    (set! new-trans-dist (translate-where))
    (set! new-angle *current-heading*)
    (set! *current-x-posn* (+ *current-x-posn*
                              (r-cos-theta (- new-trans-dist
                                              current-trans-dist)
                                           new-angle)))
    (set! *current-y-posn* (+ *current-y-posn*
                              (r-sin-theta (- new-trans-dist
                                              current-trans-dist)
                                           new-angle)))
  )
)

(define *cell-scale* 50) ; cell size is 5 cm

(define (cell-goto x y)   ;given a cell number, go to the center of it.
  (goto (list (* x *cell-scale*) (* y *cell-scale*)))
)

(define (follow-surf x-cell-list y-cell-list)
  (do ((index 0 (+ index 1)))
      ((= index (vector-length x-cell-list)) (whereami))
    (cell-goto (vector-ref x-cell-list index)(vector-ref y-cell-list index))
  )
)

(define (whereami) (display *current-x-posn*) (display #\space)
                   (display *current-y-posn*) (display #\space)
                   (display *current-heading*) (newline))
```

```
(define *rangescale* 54)
(define *leftrangescale* 54)
(define *rightrangescale* 54)

(define *distance-surface-to-center* 110)

;;; these are the SCHEME definitions of the external-calls necessary to use
;;; C map-making from SCHEME.

(load "~rbrown/robotics/mobot-c/extaddr.scm")

(define (map-update x-pos y-pos heading lrange rrange)
  (external-call (address->external *map-update-addr*) x-pos y-pos heading
                                               lrange rrange))

(define grid 0)
(define localization 1)     ;these are the names we give C when we want
(define threshmap 2)
(define single_vector 3)    ;something out of one of them.
(define temp 4)
(define flood 5)
(define cspace 6)

(define *transfer-vector* (make-vector 86 0)) ;86 for 256, 43 for 128,
                                              ;22 for 64
(define (grid->vector gname row-num)
  (external-call (address->external *grid->vector-addr*) gname row-num
                 *transfer-vector*)) ;86 for gridsize of 256

;;; going from SPARC to SCHEME, don't bother with vector encoding.
(define (vector->grid gname row-num vector)
  (external-call (address->external *vector->grid-addr*)
                 gname row-num vector))

(define (clear-grids)
  (external-call (address->external *clear-grids-addr*)))

;;; and the external-calls necessary for C localization from SCHEME.

(define (prepare-for-localization and-or-add thresh)
  (external-call (address->external *prepare-for-localization-addr*)
                 and-or-add thresh))
(define sch-add-vector-localization (address->external
                                     *add-vector-localization-addr*))
(define sch-get-loc-list (address->external *return-loc-list*))

(define (add-vector-localization and-or-add side range theta epsilon)
  (external-call sch-add-vector-localization and-or-add side range
                 theta epsilon))

(define left-val 0)
(define right-val 1)
(define and-val 0)
(define add-val 1)
```

```
(define (prepare-for-and-localization thresh)
  (prepare-for-localization and-val thresh))
(define (prepare-for-add-localization thresh)
  (prepare-for-localization add-val thresh))


(define (left-and-localize range theta epsilon)
  (add-vector-localization
     and-val left-val (+ 1 (round/ (+ range *distance-surface-to-center*)
                                   *rangescale*))
     theta epsilon))

(define (right-and-localize range theta epsilon)
  (add-vector-localization
     and-val right-val (+ 1 (round/ (+ range *distance-surface-to-center*)
                                    *rangescale*))
     theta epsilon))

(define (left-add-localize range theta epsilon)
  (add-vector-localization
     add-val left-val (+ 1 (round/ (+ range *distance-surface-to-center*)
                                   *rangescale*))
     theta epsilon))

(define (right-add-localize range theta epsilon)
  (add-vector-localization
     add-val right-val (+ 1 (round/ (+ range *distance-surface-to-center*)
                                    *rangescale*))
     theta epsilon))

(define (get-loc-list xvector yvector size start starty endx endy)
  (let ((num-locs (external-call sch-get-loc-list xvector yvector
                                 size start starty endx endy)))
    (do ((i 0 (+ i 1)))
      ((= i num-locs) num-locs)
      (begin
        (display "(")
        (display (vector-ref xvector i))
        (display #\space)
        (display (vector-ref yvector i))
        (display ")")
        (newline)
      )
    )
  )
)

;;; also need external calls to the flood-fill routines.

(define (cspace) (external-call (address->external *sch-cspace-addr*)))
(define (flood goalx goaly)
  (external-call (address->external *sch-flood-addr*) goalx goaly))
```

```
(define (surf startx starty maxlen)
  (let ((x-vect-raw (make-vector maxlen 0))
        (y-vect-raw (make-vector maxlen 0))
        (x-vect #f)
        (y-vect #f)
        (surf-len 0))
    (set! surf-len (external-call (address->external *sch-surf-addr*)
                               startx starty x-vect-raw y-vect-raw maxlen))
    (write surf-len)
    (set! x-vect (make-vector surf-len 0))
    (write surf-len)
    (set! y-vect (make-vector surf-len 0))
    (write surf-len)
    (do ((i 0 (+ i 1)))
        ((= i surf-len) (list x-vect y-vect))
      (vector-set! x-vect i (vector-ref x-vect-raw i))
      (vector-set! y-vect i (vector-ref y-vect-raw i))
    )
  )
)

(define (laser-update)
  (let ((ranges (range))
        (heading *current-heading*))
    (set! heading (+ *current-heading* (caddr ranges)))
    (map-update *current-x-posn* *current-y-posn* heading
                (car ranges) (cadr ranges))
  )
)

(define (sweep omega n)
  (pan-rotate omega)
  (do ((i 0 (+ i 1)))
      ((= i n) (pan-limp))
;    (display (modulo i 10))  ;note that this display is a luxury item:  it
    (illuminate 3)
    (laser-update))          ;tells us how far it's gotten but slows things
    (extinguish 3)
)                            ;down by about 5 clock ticks per range.

(define (sweep-at-positions omega n position-list)
  (let loop ((pos-list position-list))
    (if (null? pos-list)
        position-list
        (begin
          (display (caar pos-list))
          (display #\space)
          (display (cadar pos-list))
          (display #\space)
          (goto (car pos-list))
          (display (time))
          (display #\space)
          (sweep omega n)
          (display (time))
```

```
          (newline)
          (loop (cdr pos-list))
        )
    )
  )
)

(define (blinkin-sweep-at-positions omega n position-list)
  (let loop ((pos-list position-list))
    (if (null? pos-list)
        position-list
        (begin
          (extinguish 2)
          (illuminate 1)
          (goto (car pos-list))
          (extinguish 1)
          (illuminate 2)
          (sweep omega n)
          (loop (cdr pos-list))
        )
    )
  )
)

(define (several-add-localizations range-vector-list
                                   robot-angle-offset thresh)
  (prepare-for-add-localization thresh)
  (map (lambda (range-vector)
         (left-add-localize (car range-vector)
                            (+ robot-angle-offset (cadr range-vector)) 2))
       range-vector-list)
)

(define (get-location-vectors n d-theta)
  (if (> n 0)
      (begin
        (let ((range-vector (range)))
          (display (car range-vector))
          (display #\space)
          (display (cadr range-vector))
          (display #\space)
          (display (caddr range-vector))
          (newline)
        )
        (pan-relative d-theta)
        (sleep (* 2 one-second))
        (get-location-vectors (- n 1) d-theta)
      )
  )
)
```

C.5 pushingtraj.scm

```
;;; on the pushee (the steering robot)

; rotarc-aux does the one-step rotation based on the gain values and the
; base values for translate-where and rotate-where.

(define (rotarc-aux translate-base rotate-base gain-num gain-den)
  (rotate 'by (- (round/ (* gain-num (- (translate-where) translate-base))
                         gain-den) (- (rotate-where) rotate-base))
          'on-exit (lambda (r) (rotate-halt) r)))

(define (rotarc-aux2 translate-base rotate-base gain-num gain-den max-turn)
  (let ((turnamt (- (round/ (* gain-num (-(translate-where) translate-base))
                         gain-den) (- (rotate-where) rotate-base)))
(gain-sign (* gain-num gain-den)))
    (if (negative? gain-sign) (set! turnamt (min turnamt max-turn)))
    (if (positive? gain-sign) (set! turnamt (max turnamt max-turn)))
    (rotate 'by turnamt 'on-exit (lambda (r) (rotate-halt) r))
  )
)

; rotarc takes a gain value (numerator/denominator pair) and a distance.  It
; causes the base to rotate to an angle which is gain * distance-so-far,
; relative to the translate- and rotate-where values in effect when rotarc
; is invoked.  It runs the loop until the translate-where value has changed
; by distance.  This has the effect of following a circular-arc path with
; turning radius directly proportional to the gain value, until specified
; arc distance.

(define (rotarc gain-num gain-den distance)
  (let ((rotate-base (rotate-where))
        (translate-base (translate-where)))
    (translate-power (- *rotarc-translate-power*))
    (let loop ()
         (rotarc-aux translate-base rotate-base gain-num gain-den)
         (if (or (and (< distance 0)
                      (> (- (translate-where) translate-base) distance))
                 (and (> distance 0)
                      (< (- (translate-where) translate-base) distance)))
             (loop)
         )
    )
    (translate-limp)
  )
)

; rotarc-2 is like rotarc, but only updates the robot's heading if contact
; mode is _not_ on the list of bad contact modes (eg. '(0 1 15)).  If the
; current contact mode

(define (rotarc-2 gain-num gain-den distance bad-contact-modes)
  (let ((rotate-base (rotate-where))
        (translate-base (translate-where)))
```

```
    (translate-power (- *rotarc-translate-power*))
    (let loop ()
         (if (not (member (contact-mode) bad-contact-modes))
             (rotarc-aux translate-base rotate-base gain-num gain-den))
         (if (or (and (< distance 0)
                      (> (- (translate-where) translate-base) distance))
                 (and (> distance 0)
                      (< (- (translate-where) translate-base) distance)))
             (loop)
         )
    )
    (translate-limp)
  )
)

(define *min-rotarc-translate-power* 45)
(define *max-rotarc-translate-power* 60)

(define (rotarc-3 gain-num gain-den distance bad-contact-modes)
  (let ((rotate-base (rotate-where))
        (translate-base (translate-where))
        (bad-contact-distance 0)
        (previous-distance (translate-where))
        (was-bad #f))
    (translate-power (- *rotarc-translate-power*))
    (let loop ()
      (if (not (member (contact-mode) bad-contact-modes))
          (begin
            (if was-bad
                (begin
                  (set! bad-contact-distance
                        (+ bad-contact-distance (- (translate-where)
                                                   previous-distance)))
                  (say bad-contact-distance)))
            (rotarc-aux (+ translate-base bad-contact-distance)
                     rotate-base gain-num gain-den)
            (set! previous-distance (translate-where))
            (set! was-bad #f)
          )
          (set! was-bad #t)
          )
         (if (or (and (< distance 0)
                      (> (- (translate-where) translate-base) distance))
                 (and (> distance 0)
                      (< (- (translate-where) translate-base) distance)))
             (loop)
         )
    )
    (if was-bad (begin
                  (set! bad-contact-distance
                        (+ bad-contact-distance (- (translate-where)
                                                   previous-distance)))
                  (say bad-contact-distance)))
    (translate-limp)
```

```
  )
)

(define (rotarc-4 turn-angle turning-radius bad-contact-modes)
  (let ((distance (- (round/ (* (* (abs turn-angle) turning-radius) 22)
                             (* 7 180))))
        (finish-angle (+ (rotate-where) turn-angle))
        (gain-num 1)
        (gain-den 1)
        (rotate-base (rotate-where))
        (translate-base (translate-where))
        (bad-contact-distance 0)
        (previous-distance (translate-where))
        (was-bad #f)
        (current-rotarc-translate-power *rotarc-translate-power*))
    (set! gain-den (round/ distance turn-angle))
    (set-rotate-default! 'torque 110)
    (translate-power (- *rotarc-translate-power*))
    (let loop ()
      (if (not (member (contact-mode) bad-contact-modes))
          (begin
            (if was-bad
                (begin
                  (set! bad-contact-distance
                        (+ bad-contact-distance (- (translate-where)
                                                   previous-distance)))
                  (say bad-contact-distance)))
            (rotarc-aux2 (+ translate-base bad-contact-distance)
                         rotate-base gain-num gain-den
     (- finish-angle (rotate-where)))
            (set! previous-distance (translate-where))
            (set! was-bad #f)
          )
          (set! was-bad #t)
          )
      (if (zero? (contact-mode))
          (set! current-rotarc-translate-power
                (max (- current-rotarc-translate-power 5)
                     *min-rotarc-translate-power*))
          (set! current-rotarc-translate-power
                (min (+ current-rotarc-translate-power 1)
                     *max-rotarc-translate-power*))
      )
      (translate-power (- current-rotarc-translate-power))
      (if (or (and (< turn-angle 0)
   (> (rotate-where) (+ rotate-base turn-angle)))
      (and (> turn-angle 0)
   (< (rotate-where) (+ rotate-base turn-angle))))
  (loop)
      )
   )
    (if was-bad (begin
                  (set! bad-contact-distance
                        (+ bad-contact-distance (- (translate-where)
```

```
                                                    previous-distance)))
                  (say bad-contact-distance)))
    (translate-limp)
  )
)
(define *bad-contact-modes* '(1 15))
(define *rotarc-translate-power* 40)
(set-rotate-default! 'vel 325)
(set-rotate-default! 'accel 200)

;;; on the pusher (the drive robot)
; All I need is an alignment thread
; with an unwind-protect wrapped around it that does a force-control
; translate.  I can just start up the motor and then have a steering thread.
; The steering thread _does_ need to check translate encoders to see when
; it's gone the requested distance.
(set! *rotate-to-contact-torque* 140)
(define *pushing-robot-speed* 100)

(define (pusher contact-mode-hysteresis)
  (extinguish 0)
  (illuminate 3)
  (if (impediment?
        (push-track *pushing-robot-speed* (lambda ()
                          (if (> (abs (- (contact-mode)
                                         *pushing-tracking-contact-mode*))
                                 contact-mode-hysteresis)
                              (rtc-approx *pushing-tracking-contact-mode*))
                          #f)
        )
      )
      (illuminate 0)
  )
  (extinguish 3)
)
(define *default-pusher-hysteresis* 0)
(define *pushee-thread* (spawn (lambda () #t)))
(define *pusher-thread* (spawn (lambda () #t)))

(define (go-to-lab-door)
  (set! *pushee-thread*
        (spawn
          (lambda ()
            (say "Hey, lihly!  Lehts dance!")
            (map (lambda (rotarc-args)
                   (apply (lambda (angle radius)
                            (if (zero? angle)
                                (rotarc 0 1 (- radius))
                                (rotarc-4 angle radius
                                          *bad-contact-modes*)))
                          rotarc-args))
      '((0 2500) (-90 1500) (0 500) (45 1000) (-45 1000)
        (0 2500) (90 900) (0 3000)
  ))
```

```
  (translate-halt) ; make it actually _stop_ the trip.
  (sleep (* 5 one-second))
  (translate-limp))))
)

(define (go-to-teaching-lab)
  (set! *pushee-thread*
        (spawn
          (lambda ()
            (say "Hey, lihly!  Lehts dance!")
            (map (lambda (rotarc-args)
                   (apply (lambda (angle radius)
                            (if (zero? angle)
                                (rotarc 0 1 (- radius))
                                (rotarc-4 angle radius
                                          *bad-contact-modes*)))
                          rotarc-args))
                 '((0 1800) (-90 1200) (60 1000) (-60 1000)
                   (-40 2500) (40 1800) (0 3000)))
  (translate-halt) ; make it actually _stop_ the trip.
  (say "This is the end of the line.  The bucket stops here.")
  (sleep (* 5 one-second))
  (translate-limp))))
)
(define *test-rotarc-denominator* 26)

;;; I want to be able to adjust turning radius and bad-contact-modes from
;;; Tommy's buttons.  I only want to use two buttons, though, so I'm going
;;; to set things up so that when you hit the appropriate button Tommy
;;; bumps up the appropriate value, up to a certain point, after which he
;;; goes back to the minimum value.  He will also speak the new choice.

(define (say-list l)
  (let ((say-str " "))
    (let loop ((lst l))
      (if (not (null? lst))
          (begin
            (set! say-str (string-append say-str " "
                                         (number->string (car lst))))
            (loop (cdr lst))
          )))
  (say (string-append say-str ".")))
)

(define *bad-contact-mode-choices*
  '((0)
    (0 1 15)
    (0 1 2 14 15)
    (0 1 2 3 13 14 15)
    (0 1 2 3 4 12 13 14 15)
    (0 1 2 3 4 5 11 12 13 14 15)
    (0 1 2 3 4 5 6 10 11 12 13 14 15)))

(define *remaining-bad-contact-mode-choices* *bad-contact-mode-choices*)
```

```
(define *current-bad-contact-mode-choice*
  (car *bad-contact-mode-choices*))

(define (select-next-bad-contact-mode)
  (set! *current-bad-contact-mode-choice*
        (car *remaining-bad-contact-mode-choices*))
  (say "Bad contacts ")
  (say-list *current-bad-contact-mode-choice*)
  (set!  *remaining-bad-contact-mode-choices*
         (cdr  *remaining-bad-contact-mode-choices*))
  (if (null? *remaining-bad-contact-mode-choices*)
      (set! *remaining-bad-contact-mode-choices* *bad-contact-mode-choices*)
  )
)

(define *turning-radius-choices*
  '(500 750 1000 1250 1500 1750 2000 2250 2500 3000))

(define *remaining-turning-radius-choices* *turning-radius-choices*)
(define *current-turning-radius* (car *turning-radius-choices*))

(define (select-next-turning-radius)
  (set! *current-turning-radius*
        (car *remaining-turning-radius-choices*))
  (say (string-append "Turning radius "
                      (number->string *current-turning-radius*)
                      "."))
  (set!  *remaining-turning-radius-choices*
         (cdr  *remaining-turning-radius-choices*))
  (if (null? *remaining-turning-radius-choices*)
      (set! *remaining-turning-radius-choices* *turning-radius-choices*)
  )
)

;;; One task I want to try for the results chapter of the thesis is to see
;;; how tight a series of hairpin turns the robots can maneuver a given box
;;; through.  The idea is that the steerer will go through a contact-mode-
;;; bounded (eg rotarc-4) 180 degree turn to the left, then a 180-degree
;;; turn to the right, then a 180 degree turn to the right, and so on.

(define *num-hairpins* 4) ; 4 half turns (left right left right)

(define (hairpin num-turns radius bad-contact-modes)
  (let loop ((turn-num num-turns))
    (if (not (zero? turn-num))
        (begin
          (if (zero? (modulo turn-num 2))
              (rotarc-4  180 radius bad-contact-modes)
              (rotarc-4 -180 radius bad-contact-modes)
          )
          (loop (- turn-num 1))
        )
        (begin
          (translate-halt) (sleep (* 2 one-second))
```

```
          (translate-limp) (say "Enough hairpins.")
        )))
)

(define *circle-turning-radius* 1000)

(define (fourth-setup-tommys-buttons)
  (set-button-action! button/black1
                      (lambda () (select-next-bad-contact-mode)))
  (set-button-action! button/black2
                      (lambda () (select-next-turning-radius)))
  (set-button-action! button/black3
                      (lambda () (set! *rotarc-translate-power*
                                       (+ 3 *rotarc-translate-power*))
                              (if (> *rotarc-translate-power* 80)
                                  (set! *rotarc-translate-power* 40))
                              (say (string-append "rotarc translate power "
                                      (number->string
                                         *rotarc-translate-power*)))))
  (set-button-action! button/green1
                      (lambda ()
                        (kill-thread *pushee-thread*)
                        (extinguish 5) (extinguish 6) (extinguish 7)
                        (stop) (say "end of dance.")))
  (set-button-action! button/green2
                      (lambda ()
                        (kill-thread *pushee-thread*)
                        (set! *pushee-thread*
                              (spawn
                               (lambda ()
                                 (hairpin *num-hairpins*
                                          *current-turning-radius*
                                          *current-bad-contact-mode-choice*)
                               )))
                        (say "Now vee dahnce.")))
  (set-button-action! button/green3
                      (lambda ()
                        (kill-thread *pushee-thread*)
                        (set! *pushee-thread*
                              (spawn
                               (lambda ()
                                 (say "lihly go round in circles!")
                                 (rotarc-4 -1440 *current-turning-radius*
                                           *bad-contact-modes*)
                                 (translate-halt)
                                 (say "end of dahnce"))))))
)

(if (= (system-id) 2)
    (begin
      (set-button-action! 3 (lambda () (kill-thread *pusher-thread*)
                              (set! *pusher-thread*
                                    (spawn (lambda () (pusher 0))))))
      (set-button-action! 2 (lambda ()
```

```
                              (set! *pushing-robot-speed*
                                    (+ *pushing-robot-speed* 25))))
      (set-button-action! 0 (lambda () (set! *pushing-robot-speed* 100)))
    )
)

(if (= (system-id) 1) (fourth-setup-tommys-buttons))

(define (repeated-rtc-approx mode)
  (passive-rtc-approx mode) (repeated-rtc-approx mode))

(define (align-and-rotate mode)
  (push-until-contact)
  (rotate-to-contact mode)
  (rotate-halt)
  (translate-power (- *rotarc-translate-power*))

)
```

Bibliography

[ABN93] R.C. Arkin, T. Balch, and E. Nitz. Communication of behavioral state in multi-agent retrieval tasks. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 2, pages 588–594, Atlanta, Ga, 1993.

[AH91] Sami Atiya and Greg Hager. Real-time vision-based robot localization. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 639–644, Sacramento, CA, 1991.

[AHU83] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley, 1983.

[AS85] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[AST92] P. K. Agarwal, M. Sharir, and S. Toledo. Applications of parametric searching in geometric optimization. In *Proc. Third ACM-SIAM Symp. on Discrete Algorithms*, pages 72–82, 1992.

[Bal93] Z. Balorda. Automatic planning of robot pushing operations. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 1, pages 732–737, Atlanta, Ga, 1993.

[BCD93] R. G. Brown, L. P. Chew, and B. R. Donald. Localization and map-making algorithms for mobile robots. In *IASTED International Conference on Robotics and Manufacturing*, pages 185–190, Oxford, England, September 1993.

[BD94] A. J. Briggs and B. R. Donald. Automatic sensor configuration for task-directed planning. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation, San Diego, CA*, pages 1345–1350, May 1994.

[Bes88] Paul J. Besl. Active, optical range imaging sensors. *Machine Vision and Applications*, **1**:127–152, 1988.

[BHM85] G. Bickel, G. Hausler, and M. Maul. Triangulation with expanded range of depth. *Optical Engineering*, **24**(6):975–979, 1985.

[BL91] J. Barraquand and J. C. Latombe. Robot motion planning: A distributed representation approach. *International Journal of Robotics Research*, **10**(6):628–649, 1991.

[BL93] J. Barraquand and J. C. Latombe. Nonholonomic multibody mobile robots: Controllability and motion planning in the presence of obstacles. *Algorithmica*, **10**:121–155, 1993.

[BR91] J.R. Beveridge and E.M. Riseman. Hallway navigation in perspective. Technical report, University of Massachusetts at Amherst, Computer and Information Sciences Department, June 1991.

[BRD91] F. Blais, M. Rioux, and J Domey. Optical range image acquisition for the navigation of a mobile robot. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 2574–2580, Sacramento, CA, 1991.

[Bri95] A. J. Briggs. *Task-Level Strategies for Robot Sensing and Control.* Ph.D. dissertation, Department of Computer Science, Cornell University, Ithaca, NY, January 1995.

[Bro85] R. C. Brost. Planning robot grasping motions in the presence of uncertainty. Carnegie-Mellon Robotics Institute technical report CMU-RI-TR-85-12, Computer Science Department and The Robotics Institute, July 1985.

[Buc86] S. J. Buckley. *Planning and Teaching Compliant Motion Strategies.* Ph.D. dissertation, Massachusetts Institute of Technology, 1986.

[Can86] J.F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):34–43, 1986.

[CC92] Frédéric Chenavier and James L. Crowley. Position estimation for a mobile robot using vision and odometry. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 2588–2593, Nice, France, 1992.

[CGH+93] L.P. Chew, M.T. Goodrich, D.P. Huttenlocher, K. Kedem, J.M. Kleinberg, and D. Kravets. Geometric pattern matching under Euclidean motion. In *Proc. Fifth Canadian Conference on Computational Geometry*, pages 151–156, Waterloo, Ontario, August 1993.

[CK92] L.P. Chew and K. Kedem. Improvements on approximate pattern matching problems. In O. Nurmi and E. Ukkonen, editors, *Proc. Third Scandinavian Workshop on Algorithm Theory.* Lecture Notes in Computer Science 621, Springer-Verlag, 1992.

[CMW+93] J.M. Cameron, D.C. MacKenzie, K. R. Ward, R.C. Arkin, and W.J. Book. Reactive control for mobile manipulation. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 3, pages 228–235, Atlanta, Ga, 1993.

[Cox88] Ingemar J. Cox. Blanche: An autonomous robot vehicle for structured environments. In *Proc. of the 1988 IEEE International Conference on Robotics and Automation*, pages 978–982, Philadelphia, PA, 1988.

[CR92] W. Clinger and J. Rees. Revised^4 report on the algorithmic language scheme. Technical report, Cornell University Department of Computer Science, 1992.

[Deu82] F. Deumlich. *Surveying Instruments*. Walter de Gruyter, Berlin, Germany, 1982.

[DJ91] B. R. Donald and James S. Jennings. Sensor interpretation and task-directed planning using perceptual equivalence classes. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 190–197, Sacramento, CA, 1991.

[DJR93] B. R. Donald, James S. Jennings, and D. Rus. Experimental information invariants for cooperating autonomous mobile robots. In *Proceedings of the 1993 International Joint Conference on Artificial Intelligence, Workshop on Dynamically Interacting Robots*, Chambery, Fr., 1993.

[DJR94a] B. R. Donald, James S. Jennings, and D. Rus. Analyzing teams of cooperating mobile robots. In *Proceedings of the 1994 IEEE International Conference on Robotics and Automation*, pages 1896–1903, San Diego, CA, 1994.

[DJR94b] B. R. Donald, James S. Jennings, and D. Rus. Information invariants for distributed manipulation. In *Proceedings of the 1994 Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, 1994.

[Don87] B. Donald. *Error Detection and Recovery for Robot Motion Planning with Uncertainty.* Ph.D. dissertation, Massachusetts Institute of Technology, 1987.

[Don89] B. Donald. *Error Detection and Recovery in Robotics*, volume 336 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1989.

[Don95] B.R. Donald. Information invariants in robotics. *Artificial Intelligence*, **72**, 1995.

[Dru87] Drumheller. Mobile robot localization using sonar. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-9**(2), 1987.

[Elf87] A. Elfes. Sonar-based real-world mapping and navigation. *IEEE Journal of Robotics and Automation*, **RA-3**(3), 1987.

[Erd84] M. Erdmann. On motion planning with uncertainty. Masters dissertation, Massachusetts Institute of Technology, 1984.

[Erd93] M. Erdmann. Action subservient sensing and design. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 2, pages 592–598, Atlanta, Ga, 1993.

[Fly93] A. Flynn. *Mobile Robots: Inspiration to Implementation.* A. K. Peters, Wellesley, Massachusetts, 1993.

[GCV83] Georges Giralt, Raja Chatila, and Marc Vaisset. An integrated navigation and motion control system for autonomous multisensory mobile robots. *Robotics Research*, **1**:191–214, 1983.

[GLP85] W. Eric L. Grimson and Tomas Lozano-Pérez. Recognition and localization of overlapping parts from sparse data in two and three dimensions. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 61–66, St. Louis, MO, 1985.

[GLP87] W.E.L. Grimson and T. Lozano-Pérez. Localizing overlapping parts by searching the interpretation tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 9(4):469–482, July 1987.

[GMR91] L. J. Guibas, R. Motwani, and P. Raghavan. The robot localization problem in two dimensions. In *Symposium on Discrete Algorithms*, pages 259–268, 1991.

[GRDSV89] P. GrandJean and A. Rober De Saint Vincent. 3-d modeling of indoor scenes by fusion of noisy range and stereo data. In *Proceedings of the 1989 IEEE International Conference on Robotics and Automation*, pages 681–687, Scottsdale, AZ, 1989.

[Gro87] Ultrasonic Components Group. *Ultrasonic Ranging System.* Polaroid, Inc., 1987.

[GSO92] Javier Gonzalez, Anthony Stentz, and Anibal Ollero. An iconic position estimator for a 2d laser rangefinder. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 2646–2651, Nice, France, 1992.

[HK90] D. P. Huttenlocher and K. Kedem. Efficiently computing the hausdorff distance for point sets under translation. In *Proc. of the Sixth ACM Symposium on Computational Geometry*, pages 340–349, 1990.

[HK93] D.P. Huttenlocher and K. Kedem. Distance metrics for comparing shapes in the plane. In *Symbolic and Numerical Computation for Artificial Intelligence*, pages 201–219. Academic Press, 1993.

[HLR94a] D.P. Huttenlocher, M.E. Leventon, and W.J. Rucklidge. Visually-guided navigation by comparing two-dimensional edge images. Technical Report 1407, Cornell University, Department of Computer Science, 1994.

[HLR94b] D.P. Huttenlocher, M.E. Leventon, and W.J. Rucklidge. Visually-guided navigation by comparing two-dimensional edge images. In *Proc. Computer Vision and Pattern Recognition*, pages 842–847, Seattle, Washington, 1994.

[HMB92] Alois A. Holenstein, Markus A. Müller, and Essam Badreddin. Mobile robot localization in a structured environment cluttered with obstacles. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 2576–2581, Nice, France, 1992.

[Hog88] N. Hogan. Impedance control of industrial robots. *Robotics and Computer-Integrated Manufacturing*, **1**, 1988.

[Hor93] Ian D. Horswill. *Specialization of Perceptual Processes.* Ph.D. dissertation, Massachusetts Institute of Technology, 1993.

[HR93] K. Hughes and N. Ranganathan. A model for determining sensor confidence. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 3, pages 136–141, Atlanta, Ga, 1993.

[HTP+91] Jiawei Hong, Xiaonan Tan, Brian Pinette, Richard Weiss, and Edward M. Riseman. Image-based homing. COINS Technical Report 91-15, University of Massachusetts, Amherst, February 1991.

[HU90] D.P. Huttenlocher and S. Ullman. Recognizing solid objects by alignment with an image. *International Journal of Computer Vision*, 5(2):195–212, 1990.

[IDD90] Tsutomu Ito, Daniel DeMenthon, and Larry S. Davis. Fast range scanner using an optic ram. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 1503–1508, Cincinnati, OH, 1990.

[IRST92] M. Iagolnitzer, F. Richard, J. F. Samson, and P. Tournassoud. Locomotion of an all-terrain mobile robot. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 104–109, Nice, France, 1992.

[IST90] Hiroshi Ishiguro, Patrick Stelmaszyk, and Saburo Tsuji. Acquiring 3-d structure by controlling visual attention of a mobile robot. In *Proceedings of the 1990 IEEE International Conference on Robotics and Automation*, pages 755–760, Cincinnati, OH, 1990.

[IYT91] Hiroshi Ishiguro, Masashi Yamamoto, and Saburo Tsuji. Panoramic representations of scenes around a point. In *Proceedings of the 1991 IEEE/RSJ International Conference on Intelligent Robot Systems*, 1991.

[Jaz70] A. H. Jazwinski. *Stochastic Processes and Filtering Theory*. Academic Press, 1970.

[Jen95] J. S. Jennings. *Distributed Manipulation with Mobile Robots*. Ph.D. dissertation, Cornell University, 1995.

[JR93] J.S. Jennings and D. Rus. Active model acquisition for near-sensorless manipulation with mobile robots. In *IASTED International Conference on Robotics and Manufacturing*, pages 179–184, Oxford, England, September 1993.

[KGC91] T. Kanade, A. Gruss, and L. R. Carley. A very fast vlsi rangefinder. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 1322–1329, Sacramento, CA, 1991.

[Kha80] O. Khatib. *Commande Dynamique dans l'Espace Opérationnel des Robots Manipulateurs en Présence d'Obstacles*. Ph.D. dissertation, École Nationale Supérieure de l'Aeronautique et de l'Espace, 1980.

[Kho87] D.E. Khoditscheck. Exact robot navigation by means of potential fields. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation*, pages 1–6, Raleigh, NC, 1987.

[KK88] Eric Krotkov and Ralf Kories. Adaptive control of cooperating sensors: Focus and stereo ranging with an agile camera system. In *Proc. of the 1988 IEEE International Conference on Robotics and Automation*, pages 548–553, Philadelphia, PA, 1988.

[KK91] In So Kweon and Takeo Kanade. Extracting topographic features for outdoor mobile robotics. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 1992–1997, Sacramento, CA, 1991.

[Kle92] Lindsay Kleeman. Optimal estimation of position and heading for mobile robots using ultrasonic beacons and dead-reckoning. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 2582–2587, Nice, France, 1992.

[KP92] Stephen M. Killough and François G. Pin. Design of an omnidirectional and holonomic wheeled platform prototype. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 84–90, Nice, France, 1992.

[KTB87] D. J. Kriegman, E. Triendl, and T.O. Binford. A mobile robot: Sensing, planning, and locomotion. In *Proceedings of the 1987 IEEE International Conference on Robotics and Automation*, pages 1402–408, Raleigh, NC, 1987.

[Kuc90] R. Kuc. A spatial sampling criterion for sonar obstacle detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **PAMI-12**(7), 1990.

[KV88] P. Khosla and R. Volpe. Superquadric artificial potentials for obstacle avoidance and approach. In *Proceedings of the 1988 IEEE International Conference on Robotics and Automation*, pages 1178–1184, Philadelphia, PA, 1988.

[Lat92] J.-C. Latombe. *Robot Motion planning*. Kluwer, 1992.

[LDWC90] J. Leonard, H. Durrant-Whyte, and I. J. Cox. Dynamic map building for an autonomous mobile robot. In *Proceedings of the 1990 IEEE/RSJ International Conference on Intelligent Robot Systems*, 1990.

[LM94] Kevin M. Lynch and Matthew T. Mason. Stable pushing: Mechanics, controllability, and planning. In *Proceedings of the 1994 Workshop on the Algorithmic Foundations of Robotics*, San Francisco, CA, 1994.

[LP83] T. Lozano-Pérez. Spatial planning: A configuration space approach. *IEEE Transactions on Computers*, C–32(2):108–120, 1983. Also MIT A.I. Memo 605, Dec. 1982.

[LPMT84] T. Lozano-Pérez, M. T. Mason, and R. H. Taylor. Automatic synthesis of fine-motion strategies for robots. *International Journal of Robotics Research*, **3**(1), 1984.

[LRDG90] J. Lengyel, M. Reichert, B. Donald, and D. Greenberg. Real-time robot motion planning using rasterizing computer graphics hardware. Department of Computer Science Technical Report TR 90-1122, Cornell University Department of Computer Science, May 1990.

[Lum86] V.J. Lumelsky. Continuous motion planning in unknown environment for a 3d cartesian robot arm. In *Proceedings of the 1986 IEEE International Conference on Robotics and Automation, San Francisco, CA*, pages 1569–1574, April 1986.

[Lyn92] Kevin M. Lynch. The mechanics of fine manipulation by pushing. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2269–2276, Nice, France, 1992.

[Mar82] D. Marr. *Vision*. W. H. Freeman and Company, New York, 1982.

[Mat90] M. J. Mataric. A model for distributed mobile robot environment learning and navigation. Masters dissertation, Massachusetts Institute of Technology, 1990.

[Mat93] Maja J. Mataric. Kin recognition, similarity, and group behavior. In *Proc. of the Fifteenth Annual Cognitive Science Society Conference*, Boulder, Colorado, 1993.

[MDW93] J.M. Manyika and H.F. Durrant-Whyte. A tracking sonar sensor for vehicle guidance. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 2, pages 424–429, Atlanta, Ga, 1993.

[ME85] H. P. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the 1985 IEEE International Conference on Robotics and Automation*, pages 116–121, St. Louis, MO, 1985.

[Meg83] N. Megiddo. Applying parallel computation algorithms in the design of serial algorithms. *JACM*, 30:852–865, 1983.

[MH79] D. Marr and E. Hildreth. Theory of edge detection. Artificial Intelligence Laboratory Memo 518, Massachusetts Institute of Technology Artificial Intelligence Laboratory, April 1979.

[Mor89] H. P. Moravec. Sensor fusion in certainty grids for mobile robots. In A. Casals, editor, *Sensor Devices and Systems for Robotics*. Springer-Verlag, 1989.

[MR88] Clare D. McGillem and Theodore S. Rappaport. Infra-red location system for navigation of autonomous vehicles. In *Proc. of the 1988 IEEE International Conference on Robotics and Automation*, pages 1236–1238, Philadelphia, PA, 1988.

[MS85] Matthew T. Mason and J. Kenneth Salisbury. *Robot Hands and the Mechanics of Manipulation*. MIT Press, London, England, 1985.

[Nor93] Fabrice R. Noreils. Toward a robot architecture integrating cooperation between mobile robots: Application to indoor environment. *International Journal of Robotics Research*, **12**:79–98, 1993.

[NP82] J. Neivergelt and F. P. Preparata. Plane-sweep algorithms for intersecting geometric figures. *CACM*, 25(10):739–747, 1982.

[OS89] A. V. Oppenheim and R. W. Schafer. *Discrete Time Signal Processing.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[OW83] A. V. Oppenheim and A. S. Willsky. *Signals and Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1983.

[OY92] Yoshikuni Okawa and Ken Yokoyama. Control of a mobile robot for the push-a-box operation. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 761–766, Nice, France, 1992.

[Par94] Lynne E. Parker. *Heterogeneous Multi-Robot Cooperation.* Ph.D. dissertation, Massachusetts Institute of Technology, 1994.

[PP93] N. Pears and P. Probert. An optical range sensor for mobile robot guidance. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 3, pages 659–664, Atlanta, Ga, 1993.

[PS85] F. P. Preparata and M. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, 1985.

[RD92] J. A. Rees and B. R. Donald. Program mobile robots in scheme. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, Nice, France, 1992.

[RG92] Anil S. Rao and Kenneth Y. Goldberg. Orienting generalized polygonal parts. In *Proceedings of the 1992 IEEE International Conference on Robotics and Automation*, pages 2263–2268, Nice, France, 1992.

[RG93] S. Ratering and M. Gini. Robot navigation in a known environment with unknown moving obstacles. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 1, pages 25–30, Atlanta, Ga, 1993.

[Rio84] M. Rioux. Laser range finder based upon synchronized scanners. *Applied Optics*, **23**(21):3837–3844, 1984.

[RJB+91] D. B. Reister, J. P. Jones, P. L. Butler, M. Beckerman, and F. J Sweeney. Demo 89 – the initial experiment with the hermies-iii robot. In *Proceedings of the 1991 IEEE International Conference on Robotics and Automation*, pages 2562–2567, Sacramento, CA, 1991.

[Row86] J. R. Rowland. *Linear Control Systems: Modeling, Analysis, and Design.* John Wiley and Sons, New York, 1986.

[Ruc93] W.J. Rucklidge. Lower bounds for the complexity of the Hausdorff distance. In *Proc. Fifth Canadian Conference on Computational Geometry*, pages 145–150, Waterloo, Ontario, August 1993.

[Ruc95] W.J. Rucklidge. *Efficient Computation of the Minimum Hausdorff Distance for Visual Recognition.* Ph.D. dissertation, Cornell University, January 1995.

[RWA+92] Yuval Roth, Annie S. Wu, Remzi H. Arpaci, Terry Weymouth, and Ramesh Jain. Model-driven pose correction. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 2625–2630, Nice, France, 1992.

[Sar89] K. B. Sarachik. Visual navigation: Constructing and utilizing simple maps of an indoor environment. Artificial Intelligence Laboratory Technical Report 1113, Massachusetts Institute of Technology Artificial Intelligence Laboratory, March 1989.

[SB93] Daniel J. Stilwell and John S. Bay. Toward the development of a material transport system using swarms of antlike robots. In *Proceedings of the 1993 IEEE International Conference on Robotics and Automation*, pages 766–771, Atlanta, GA, 1993.

[SF93] K. Singh and K. Fujimura. Map making by cooperating mobile robots. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 1, pages 254–259, Atlanta, Ga, 1993.

[SHD86] T.M. Silberberg, D.A. Harwood, and L.S. Davis. Object recognition using oriented model points. *Computer Vision, Graphics and Image Processing*, 35:47–71, 1986.

[SKS88] S. Shekbar, O. Khatib, and M. Shimojo. Object localization with multiple sensors. *International Journal of Robotics Research*, **7**(6):34–44, 1988.

[TL92] H. Takeda and J.C. Latombe. Planning the motions of a mobile robot in a sensory uncertainty field. Stanford CS Technical Report STAN-C-92-1424, Stanford University, Department of Computer Science, April 1992.

[VTGAS93] S.J. Vestli, N. Tschichold-Gürman, M. Adams, and S. Sulzberger. Integration of path planning, sensing, and control in mobile rootics. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 3, pages 243–248, Atlanta, Ga, 1993.

[WA92] Mark West and Haruhiko Asada. Design of a holonomic omnidirectional vehicle. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 97–103, Nice, France, 1992.

[YH88] Kazuhiro Yoshida and Shigeo Hirose. Laser triangulation range finder available under direct sunlight. In *Proc. of the 1988 IEEE International*

Conference on Robotics and Automation, pages 1702–1708, Philadelphia, PA, 1988.

[YNY92] Yasushi Yagi, Yoshimitsu Nishizawa, and Masahiko Yachida. Map-based navigation of the mobile robot using omnidirectional image sensor copis. In *Proc. of the 1992 IEEE International Conference on Robotics and Automation*, pages 47–52, Nice, France, 1992.

[YY93] Y. Yamamoto and X. Yun. Control of mobile manipulators following a moving surface. In *Proc. of the 1993 IEEE International Conference on Robotics and Automation*, volume 3, pages 1–6, Atlanta, Ga, 1993.