NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

SONAR-BASED REAL WORLD MAPPING SYSTEM

by

Louis-Paul Normand

A Thesis Submitted to The School of Graduate Studies and Research in Partial Fulfillment of the Requirements for the Degree of Master of Applied Science

Ottawa-Carleton Institute of Electrical Engineering

Department of Electrical Engineering
Faculty of Engineering

University of Ottawa

ISBN 0-315-60013-6

UNIVERSITÉ D'OTTAWA
UNIVERSITY OF OTTAWA

I hereby declare that I am the sole author of this thesis. I authorize the University of Ottawa to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Louis-Paul Normand

I further authorize the University of Ottawa to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Louis-Paul Normand

Abstract

Autonomous robot navigation is a complex processing task which can be broken down in several levels. These levels are Robot Control, Sensor Interpretation, Sensor Integration, Real-World Modelling, Navigation, Global Planning and Control. The work done in this thesis addresses the first four of these levels in order to generate an accurate environment model from sensory data.

A real world mapping system was developed and implemented for a mobile platform evolving in an unstructured environment. The system uses sonars as the source of sensory data. A certainty-grid based mapping method is used to map and merge the sonar data onto a pixel representation. For the purpose of this research, a user-friendly application is used to provide a test platform and demonstrate the system capabilities.

Acknowledgements

Grateful thanks to Dr. Emil M. Petriu for his advice, guidance and friendship during the course of my research.

Special thanks to Mr. D. Green, Mr. R. Liscano, Mr. S. Lang, Mr. L. Korba and Mr. N. Burtnyk from the National Research Coucil of Canada for their much appreciated support.

Finally, I would like to thank my fiancee, Lianne Sauve, for her support throughout the course of this masters degree program. I would also like to thank my family and friends for their moral support.

Table of contents

List of Figures

1. Introduction

Robotics, has in the past few years, progressed very rapidly. Many theoretical applications of a few years ago have now been implemented in the industry. Industrial applications range from welding to assembly tasks. Most of these tasks involve a stationary robot that carries out its function on objects that are brought up to the working position. However, to widen the spectrum of activity of robotic application, it became essential to provide the robots with a greater autonomy. Such an autonomy refers to the ability for a robot to operate in an unstructured environment. To accomplish this, a robot must be able to acquire information about its environment and model that information so that tasks such as navigation and object localisation can be carried out. Through these two capabilities, robot autonomy can be increased by allowing the robot to find the object on which it is to perform some activity and move from its current position to that object. Furthermore, navigation in itself opens up a wide range of applications from transportation of dangerous materials to disabled patient transportation.

The main problem for reliable navigation systems come from the sensing devices and the interpretation of their data. First of all, due to the inherent uncertainties of any kind of sensor, it is important to compose the information from different sensor types and from multiple readings. Once this is accomplished, a rich model of the environment has to be produced. This model has to support functions such as path planning, obstacle avoidance, landmark identification and position and motion estimation. This world model provides the most essential element of a navigation system as it represents the starting point from which the higher level representations are derived.

For the purpose of this research, it was decided that a sonar-based system was to be implemented to provide a world mapping capability. The aim was to

provide a mobile platform with a mapping system capability capable of supporting navigation data representation requirements.

In order to reduce the complexity of a sonar-based mapping system, the processing task is broken down in processing levels which correspond to a series of activities that are to be carried out to achieve a navigation capability for a mobile robot. For each of these processing levels, a data representation scheme has to be implemented to optimise processing. At the lower levels, the data has to support deterministic algorithms while at higher levels the data has to support heuristic tasks. Several data representation schemes are discussed but the certainty grid method was selected because of its ability to support compositions of information. The details on how this method applies to sonar and how it resolves some of the sonar's main limitations are also revealed. In order to structure the data representation scheme used to support the various processing levels, a multiple axis of representation is introduced. This multiple axis includes a geometrical axis, an abstraction axis and a resolution axis.

The architecture of a complete navigation system is introduced from which the mapping system architecture is derived. The mapping system laboratory configuration is also presented. Many design decisions are directly related to this configuration.

The development of the mapping system followed a life cycle approach which takes the designer into a requirements analysis phase, a design phase, a coding phase and finally into a testing phase. The essence of each of these phases is also discussed.

2. Autonomous Robot Navigation

2.0 Introduction

Autonomous robots could, in the very near future, have a significant impact on various areas including transportation, exploration or manipulation. Some direct applications include mining, cleaning work, wheelchair control and many more. Autonomous means that the robot must rely on its "on board" resources to accomplish a required high level activity. In a mobile robot, one of the basic but non-trivial activities required is navigation. Navigation can be defined as the task that finds an obstacle-free path between a start and an end point.

Autonomous navigation in itself poses quite a complex problem. The robot cannot rely on a priori knowledge since the environment that it evolves in is typically quite dynamic. An autonomous navigation system must therefore use on-board sensors to acquire the necessary data. It must then model that data and represent it in such a way that meaningful information can be easily processed to compute a path between the robot's current position and an intermediate or an end goal destination. This destination is typically derived from a set of heuristics that are applied at various levels of data abstraction in the data representation hierarchy. Data abstraction refers to the levels of data refinement that are required to optimise computations inherent in each processing level. The planning requirements vary with the task required from the robot. For example, a local obstacle avoidance task is processed at a lower level than a global path planning task and therefore requires more detail at the map level but doesn't require a symbolic representation necessary in non-deterministic computations.

Mobile robot researchers have used a variety of world model representations. The two main ones are boundary-based and volumetric or

space-filling. In the boundary-based category we find line segment techniques (shown in figure2.4) as described by Crowley and Drumheller in [8] and [6] and polygon techniques presented by Kuan in [10]. As for the volumetric category, we find the occupancy grid and the quadtree representations shown in figure 2.3. By fitting data to parameterized models, boundary-based representations impose strong geometric assumptions on the sensor data. Doing this early in the interpretation process is problematic because it commits the system to rigid descriptions based on insufficient evidence. Because of the inherent inaccuracies of any type of sensor, early geometric interpretation is difficult. It also underscores the importance of redudant data for noise reduction and the importance of representing confidence in the raw data and the derived representations.

The projects studied for this thesis include: the Microbe [1] robot from Munich University which implements a navigation system for an artificial environment using a pixel-based representation; the Dolphin [2] system from Carnegie Mellon University which provides a mobile platform with a mapping system using the certainty-grid representation; the Intelligent Mobile Platform (IMP) introduced by Crowley in [8], again from Carnegie Mellon University, which uses line segments; the Mobile Autonomous Robot Stanford (MARS) which uses a hierarchy of models ranging from the motion model to the world model and finally, the mobile robot system introduced in [36] from Holland which uses line segments.

In this chapter the various conceptual processing levels of autonomous navigation are defined which in essence provide the framework for this research. Representation schemes are discussed for various levels of data abstraction. Sensor modelling principles are also presented using sonar as the source of sensor data. Sonars are very attractive because of their simplicity and low implementation costs, however their resolution is rather limited. The

principles presented in this chapter can be easily adapted to other sensor types. The various aspects of composing information from different sensors or from the same sensor source but different perspective are also discussed. Finally an architecture for a complete autonomous navigation system is presented.

2.1 Conceptual Processing Levels

It is essential to break down a navigation system into processing levels in order to optimise the use of computing resources. The levels introduced by Alberto Elfes [2] (Figure 2.1) constitute an attempt at breaking down the various functions to be carried out in a navigation system. Each level commands the execution of its predecessor and uses that predecessor's output to do its own processing.

In this section, the evolution of the data representation will be shown along with the activities that can be initiated from each level. The processing levels are robot control, sensor interpretation, sensor integration, real world modelling, navigation, global planning and control. Each will be elaborated on in the following paragraphs.

The robot control level is primarily concerned with the set of primitives that are used to control the various motors, actuators and sensors on the mobile robot. These primitives are implemented by the manufacturers and provide direct control of the robot's capabilities in terms of motion and sensor control.

VII. Control

Global Supervision of System Behaviour
User Interface
Scheduling of Activities
Integration of Plan-driven with Data-driven Activities

VI. Global Planning

Task-level Planning to provide sequences of sensory, actuator and processing (software) actions
Error-recovery and Replanning in case of failure or unexpected events

V. Navigation

Navigation Modules provide services such as Path-planning and Obstacle Avoidance

IV. Real-world Modelling

Integration of local pieces of correlated information into a Global Real-world Model that describes the robot's environment of operation
Matching acquired information against stored maps
Object Identification
Landmark Recognition

III. Sensor Integration

Information provided by different Sensor Modules is correlated and abstracted
Common representations and compatible frames of reference are used

II. Sensor Interpretation

Acquisition of Sensor Data (Vision, Sonar, Laser Rangefinder, etc.)
Interpretation of Sensor Data

I. Robot Control

Set of primitives for Robot Operation
Actuator Control (e.g. Locomotion)
Sensor Control
Internal Sensor Monitoring

figure 2.1 Processing levels

This level, in fact, provides the interface between the robot and its sensor systems and the remaining processing levels. Motion control typically includes drive control and steering actuator control while sensor control deals with the on-board external sensors such as sonars. The command routines which are triggered by primitives are normally stored in the sensor controller system where they also run. The only interaction with the controller system can be via standard parallel or serial interface thus freeing up the main processing resources for higher level computations. Internal sensor monitoring deals with the robot's internal status sensors such as dead reckonning.

The sensor interpretation level is responsible for two main functions, the acquisition of sensor data and the transformation of that data into a common sensor representation. This level is unique to each type of sensor but its output should be of the same format regardless of the sensor used. Acquisition of data involves, for example, firing patterns in the case of a sonar, or "snap shot" in the case of a vision system. It also involves high level handshaking activities with the sensor control system of the robot control level. The data transformation at this level depends on the system. If a mobile robot only has one type of sensor, it becomes very easy to find a common sensor data representation. On the other hand, if the platform possesses many different types of sensors, the sensor data will have to be refined to a higher level before a common representation can be found.

The sensor integration level uses the sensor data acquired and formatted by the sensor interpretation level for each sensor type and integrates it into one representation. Again the extent of processing required here depends on the representation level of the data passed from the sensor interpretation level. Data can be further refined at this level, however, if the common representation required by the sensor interpretation level is already sufficiently refined, the task becomes a merging task rather than a refining one.

The real world mapping level is required to integrate the representations generated by the sensor integration level into a meaningful framework. Such a framework is implemented by incrementally composing information obtained from different perspectives. This representation can then be used for matching purposes with previously stored maps which in turn provide capabilities such as position update and landmark recognition. Up to this level, algorithms are normally deterministic, involving straight forward data transformation. The next levels deal with decision-making and often require algorithms emulating human behaviour using a heuristic approach to resolve problems.

The navigation level deals with tasks such as local obstacle avoidance, intermediate path planning and detection of emergencies. Although some of the aspects of the navigation level can be computed deterministically, such as path planning, the determination of intermediate goals will be heuristic if insufficient data is available to execute deterministic path planning algorithms.

Global planning is responsible for achieving the global goal as commanded. This level generates a task level plan that converges towards the end goal. This plan is updated every time new data is available. This level is also responsible for error detection, diagnosis and recovery.

The final level is the control level and is responsible for the scheduling of all the activities whether they are plan driven or data driven. Scheduling has to obey heuristic rules so that the robot's behaviour is logical, taking into consideration newly acquired data that reflects the current dynamic environment. The control level also provides supervisory control over the whole system and assumes the interface responsibilities with the outside world.

The levels described above provide a framework for a mobile robot navigation system software architecture. However, they only serve as guidelines

for breaking down the various navigation functions and for that reason, every implementation is different depending on the type of application, types of sensor, the type of mobile platform, etc. The various data representation schemes associated with each processing levels are discussed next.

2.2 Data Representation

It is possible to divide the inter-level means of communication into two categories as illustrated in figure 2.2. First the commands or function calls which trigger the execution of the subordinate level routines and second the newly generated sensor data passed from a processing level to the next level up. Data representation deals with the second category since commands reflect the program function design and do not carry any navigational information. The choice of data representation has a direct impact on the processing done at a particular level. The primary goal of a data representation scheme should be to optimise the processing task. Let's look at each level again describing a typical data representation hierarchy for a navigation system.

At the robot control level, sensor data is in its purest form. In the case of sonar this corresponds to a range reading and a sonar position. In the case of vision, it correspond to a pixel map with its associated intensity levels. Very little data manipulation is done by the processor as most of it is produced by the sensor system hardware.

The sensor interpretation level is the level that controls the sensor data acquisition scheme and therefore has to interface with the external sensor controller system. The processing task is identical for identical types of sensors and is responsible for transforming the data into a common data representation for all sensor types. The lower the level of the common representation, the less processing is required. For example, in a system using only sonar data, the data

representation that is passed up to the next level could consist of a simple array containing the sonars' relative positions and their associated range readings. In this case, the integration into a pixel-type representation is done using that array at the next level. If vision data is used, pixel-type transformation has to be carried out for each sensor individually at this level and then integrated or merged at the next level.

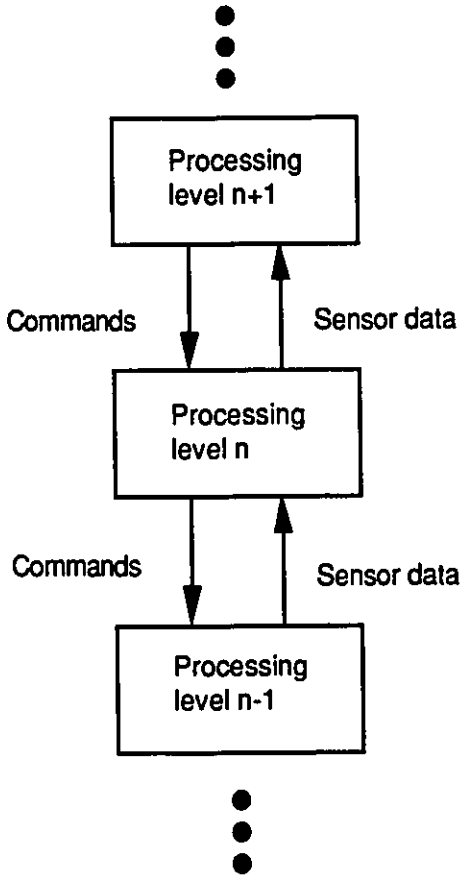


figure 2.2 Inter-level communication

At the sensor integration level, more elaborate data representation schemes emerge. The most common techniques include maps of pixel or line segments. The pixel maps include schemes such as certainty grids elaborated by Moravec[3] and quadtrees presented by Kambhampati and Dairs[4]. For line segment techniques, various approaches have been introduced, in particular by Crowley[7], Drumheller[6] and Miller[5]. The essence of these methods is discussed next.

The certainty grid method consists of overlaying an array of imaginary grids over the robot's environment. Each grid in the array covers a certain area of the environment and, depending where that area lies in relation with the sensor coverage, an occupancy probability is assigned to the grid. In order to accomplish this it is essential to possess an accurate probability distribution model of the sensor used. Almost any kind of sensor can be mapped into a certainty grid. The strength of the certainty grid method lies in the fact that the probabilities are reinforced every time new probabilities are computed from subsequent readings that cover parts of the same area. The grids are initialised to an unknown state at the beginning of a session and the new probabilities computed from the last reading are merged according to probabilistic principles which are discussed in section 2.5. The modelling of the sensor data is unique to each type of sensor and provides occupancy probabilities based on the inherent characteristics of the sensor. The model itself is determined experimentally, although statistical research using Bayesing reasoning[3] has managed to clarify some aspects of the experimental behaviour. Certainty grids are very attractive for integration of information from various types of sensor as they can adapt to the intrinsic uncertainties and resolution limits of most types of sensor. Details on how this method applies to sonars will be discussed in sections 2.3 and 2.4.

The quadtree representation is based on the recursive decomposition of a 2-dimensional picture. It starts with the whole picture. If it is homogeneous

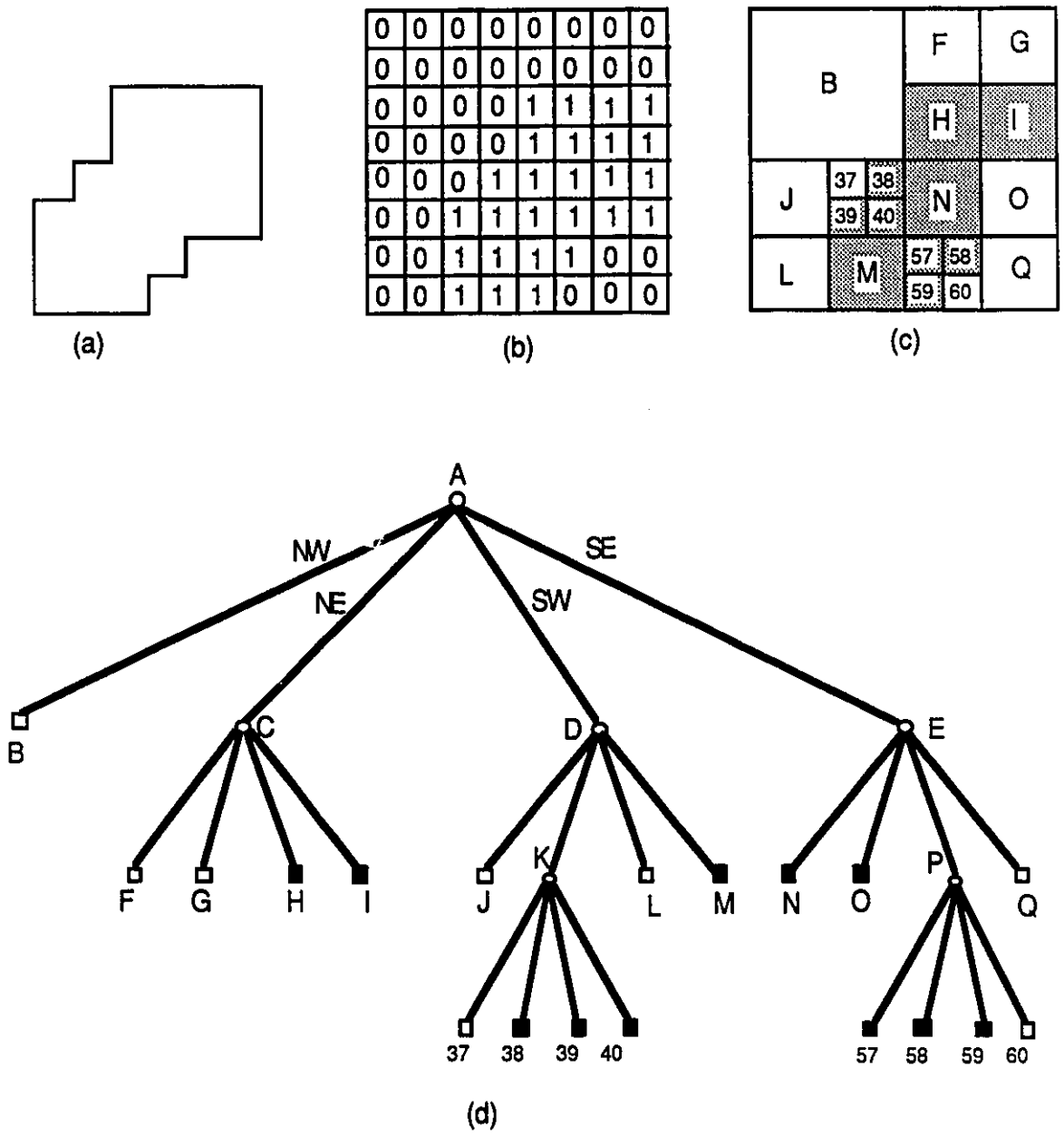


figure 2.3 Quadtree Representation: Sample image (a), its pixel map (b), its block decomposition (c) and the corresponding quadtree representation (d)

(all black or occupied, or all white or empty) the decomposition stops. If not, the picture is broken down into four equal size quadrants on which the same operation is carried out recursively. Figure 2.3 illustrates this. The decomposition can be stopped at a predetermined level or when all blocks in the quadtree are homogeneous. Quadtrees offer a significant save in data volume and, for that reason, algorithms operating on them do not have to manipulate as much data and execute considerably faster. Unfortunately, direct mapping of sensor data into quadtree representation is difficult as quadtrees are generated from a raster-type pixel array. This means that an intermediate stage, such as certainty grid mapping, is required to produce the raster-type pixel array required. Samet in [7] proposes an algorithm to convert rasters into quadtrees.

Line segment representation constitutes an interesting option to pixel based representations. The mapping process consists of extracting straight line segments from the sensor data as shown in Figure 2.4 in the case of sonar data. Line segments support easier implementation of fast localisation algorithms. These algorithms typically use matching techniques to compare the sensor data to a priori knowledge of the environment. Such algorithms are presented in [6] and [8]. Line segments are also easy to convert into a grammar-like representation in which different types of segment configurations are catalogued. This conversion is used by some matching techniques such as the ones presented in [8], the two representations that are to be matched are bumped against the grammar rules and if the set of rules they assert is the same, then a match exists. However, line segments require substantially more sensor readings than is required by other methods such as certainty grid thus does not optimise the use of the sensors. Furthermore, the task of extracting straight lines from inaccurate sensor readings, such as sonars, can be a tedious one.

The next level is the real world modelling level. This level uses the data formatted at the sensor integration level and, through matching algorithms,

carries out localisation and landmark recognition. It also integrates local pieces of correlated information into a global model. The data representation requirements for the integration function do not significantly differ from the sensor integration level. However for matching or object extraction purposes, a geometric representation describing boundaries and line segments is often more appropriate.

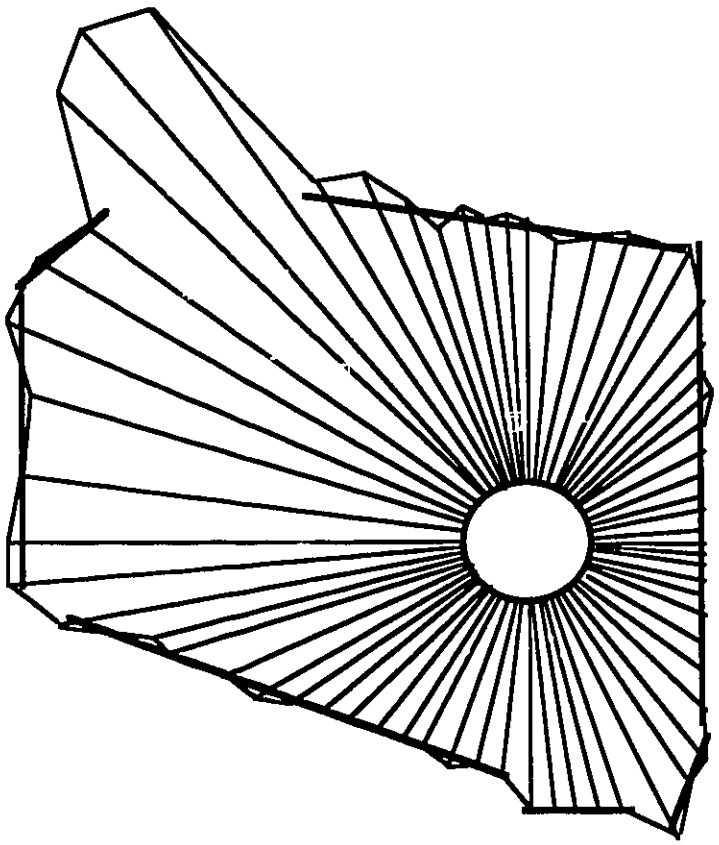


figure 2.4 Line segment representation of sonar data

The navigation level operates on the global model and therefore uses the same data representation to carry out low level path planning and obstacle avoidance tasks as the sensor integration level. However, the requirement for using a heuristic approach in high level path planning algorithms where a high level plan is passed down from the global planning level, forces a refinement of the data in order to use more appropriate languages. Representations such as semantic networks, production rules, or any other representation supported by an inference mechanism of some sort can be used. Tanimoto introduces various knowledge representation schemes in [9] some of which could be applied to this kind of application.

The next two levels, the control level and the global planning level, operate on a data set that is gradually refined to include only information pertinent to the level. This information progresses by inference starting at the navigation level up to the control level. The control level uses inferences to schedule plan driven and data driven activities. Here heuristics are required so that the robot exhibits a "rational" behaviour and saves on the total number of iterations required to attain the end goal. Global planning generates a task level plan based on the current situation and a set of heuristics to generate sequences of actuator, sensor and processing actions. The choice of representation depends on the processing task and on the type of data required. The inference engine desired also influences the type of data.

2.3 Sonar Based Mapping

As briefly mentioned in section 2.0, the rest of this chapter focuses on sonar as the means of acquiring sensor data. Sonar are the most widely used sensors for robotic applications and have been studied extensively by many researchers, in particular Moravec[3] and Elfes[2].

In a dynamic, unstructured environment, active sensing devices can provide reliable data about the surroundings with a relatively low complexity. Other sensor types such as the those using lighting techniques or triangulation-based rangefinders are not as efficient when operating in such a non-ideal environment. Amongst active sensing devices we find ultrasonic (sonar) and laser rangefinders. Sonars provide the advantage of being much less expensive than the laser rangefinders, but the latter provide better resolution and precision. Stereo vision also provides reliable information however, the computational expense is very high and, in a real-time environment, the time required to obtain adequate precision is often not available. A stereo vision system is also quite expensive to implement since it requires the support of a computer to handle the heavy data processing task. For these reasons, the popularity of sonar systems as the primary source of range data for navigation systems has grown rapidly. Today, very accurate models exist that help maximise the amount of information that can be obtained from sonars.

Sonar-based mapping for a mobile robot typically assumes that all the data must be obtained via the sonar system and therefore the navigation system cannot rely on a priori knowledge of the constantly changing surroundings. The data representation that best supports the mapping of sonar data is the certainty grid because it tends to optimise the information delivered by every reading.

Sonars give the range of the nearest reflecting object within their beamspan. Because of this, the position of the object is only known to be somewhere in a certain area at the range reading. Furthermore, the volume of the sonar cone up to the range reading can be thought of as being probably empty. As the certainty grid is laid over the sonar cone, each grid is assigned a probability of being empty and a probability of being occupied. These probabilities are modelled by probability profiles which will be discussed in the next section.

Once one sonar reading has been mapped, another reading is acquired from another perspective, possibly overlapping parts of the area covered by the beam of the first sonar. The next step consists of merging the mapped results of this newly acquired reading with the previous results. Composing information serves to reinforce overlapping occupancy probabilities so that after a few readings from different vantage points, a respectable map is built. The empty probabilities are used to enhance the boundaries of the occupied probabilities. The various considerations in composing information will be discussed in section 2.5.

2.4 Sonar Modelling

As mentioned before, modelling is essential for translation of sonar information into a certainty grid-based representation. Therefore, the accuracy of the model has a direct impact on the overall accuracy of the system. In the case of the sonar, the most complete model comes from experimentation taking into consideration known sonar characteristics and limitations.

In order to describe the experimental model introduced by Elfes in [2], let's look at some inherent problems associated with sonar readings:

a. the timing circuitry limits the range precision,

b. the detection sensitivity varies with the angle of the reflecting objects with the main beam axis,

c. multiple reflections or echoes from other sonars when fixed in salvo often produce false readings,

d. because of the wide beamwidth, the position of the object returning the echo is not precisely known.

Because of these problems, direct interpretation of sonar data is impossible reinforcing the requirements for a probabilistic approach such as the certainty grid which tends to minimise the error on data integrated from several readings taken from different vantage points.

As mentioned before, two types of probabilities are used in the case of sonars. The first one is associated with the probability for each grid located in the sonar cone to be occupied. The second one is the probability for the same grid to be empty. These probabilities are calculated from the following known sonar parameters and from data about the grid location in relation to the sonar cone:

R	range measurement as returned by the sonar
e	relative sonar error
W	sonar beamwidth
S	position of the sonar sensor S(x,y,z)
d	distance between the position P(x,y,z) for which the probabilities are sought and the sonar position S
q	angle formed by P,S and the sonar cone main axis
R_{min}	minimum measurable range from the sonar

These parameters are illustrated in Figure 2.5 along with the occupancy probability density function graphs. These density function graphs represent the two probabilities p_e and p_o, the empty probability and the occupied probability respectively. The empty probabilities are non-null for any grid that falls inside of the probably empty region delimited by the following parameters :

2.1 $R_{min} < d < R - e$ and $-W/2 \leq q \leq W/2$

The density function f_e for the probably empty region is a function of d and q thus $p_e = f_e(d,q)$. The somewhere occupied region is contained within the following boundaries:

2.2 $R - e < d < R + e$ and $-W/2 \leq q \leq W/2$

The density function f_o for the somwhere occupied region is also a function of d and q thus $p_o = f_o(d,q)$.

The empty probability density function can be written as a product of a function of d, G(d), by a function of q, H(q). Thus

2.3 $p_e = f_e(d,q) = G(d) \cdot H(q)$

where

2.4 $G(d) = 1 - ((d - R_{min}) / (R - e - R_{min}))^2$

for

2.5 $R_{min} \leq d < R - e$

and

2.6 $G(d) = 0$ elsewhere

and

2.7 $H(q) = 1 - (2q/W)^2$

for

2.8 $-W/2 \leq q \leq W/2$

and

2.9 $H(q) = 0$ elsewhere

The occupied probability density function can also be written as a product of a function of d, P(d) by a function of q, Q(q). Thus

$$2.10 \quad p_o = f_o(d,q) = P(d) \cdot Q(q)$$

where

$$2.11 \quad P(d) = 1 - ((d - R/e)^2$$

for

$$2.12 \quad R - e \leq d \leq R + e$$

and

$$2.13 \quad P(d) = 0 \text{ elsewhere}$$

and

$$2.14 \quad Q(q) = 1 - (2q/W)^2$$

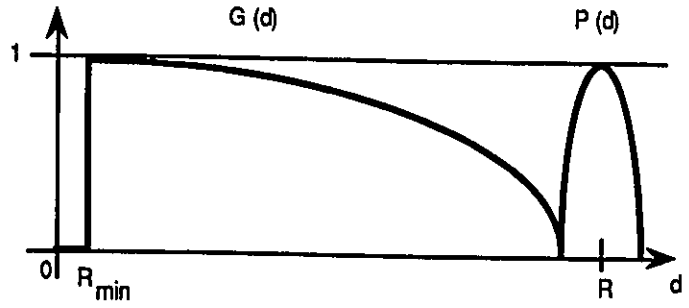
for

$$2.15 \quad -W/2 \leq q \leq W/2$$

and

$$2.16 \quad Q(q) = 0 \text{ elsewhere.}$$

The two probabilities, p_e and p_o are thresholded to produce the final map. The values for the occupancy probabilities range from 0 to 1 where 0 is unknown and 1 is occupied or empty. The probabilities correspond to the degree of certainty for the grid of being empty or occupied.



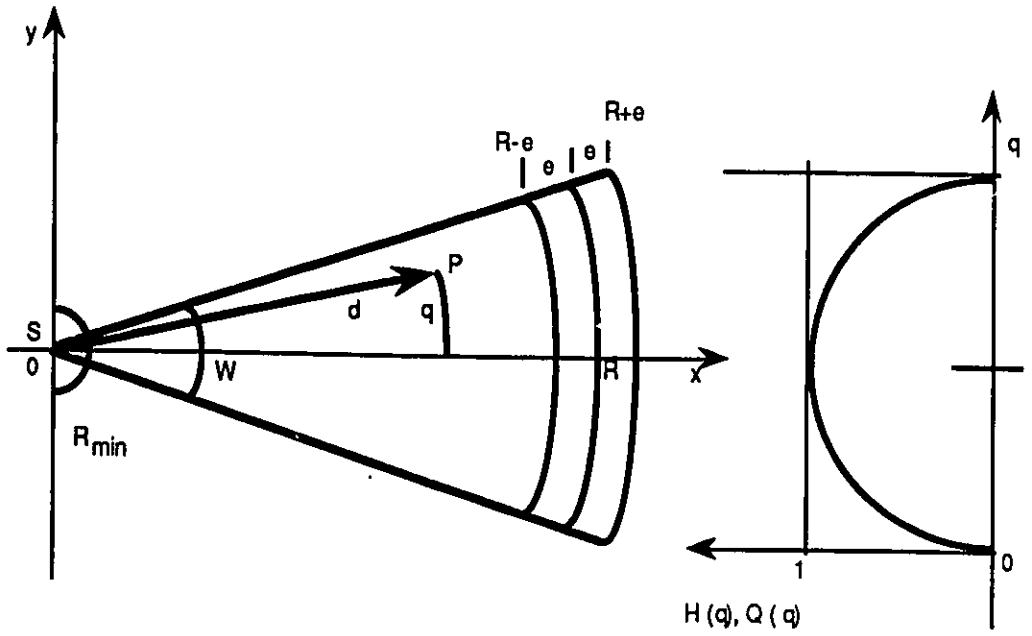


figure 2.5 Occupancy probabilities distribution

2.5 Composing Information

In order to achieve higher goals in the navigation system hierarchy such as sensor fusion, motion planning, landmark identification and to reduce the uncertainties inherent to sonar readings, it is important to have a scheme to integrate the mapped results of several sonars together. Furthermore, integration of information is also required to extend the "awareness" of the robot about its environment by merging several maps together.

The goal of information composition is to increase the occupancy probabilities where readings confirm each other and decrease them where readings are contradictory. The operations performed on the occupied and empty probabilities are different. It is possible to reduce the range reading uncertainties of one sonar by using the empty probabilities. Empty probabilities represent a space whose totality is probably empty and are simply added using the following probabilistic addition formula:

$$2.17 \quad p_e(\text{grid}) = p_e(\text{grid}) + p_e(\text{reading}) - p_e(\text{grid}) \cdot p_e(\text{reading})$$

where $p_e(\text{grid})$ is the already stored value to be enhanced by $p_e(\text{reading})$ the newly acquired value.

Occupied probabilities represent a lack of knowledge concerning the location of a reflecting object and are first weakened by conflicting information using the equation:

$$2.18 \quad p_o(\text{reading}) = p_o(\text{reading}) \cdot (1 - p_e(\text{grid}))$$

where $p_o(\text{reading})$ is the occupied probability obtained from the new reading and

decreased by the p_e(grid) as computed by equation 2.17. The result is then added using the same probabilistic addition formula as for the empty probability thus:

$$2.19 \quad p_o(\text{grid}) = p_o(\text{grid}) + p_o(\text{reading}) - p_o(\text{grid}) \cdot p_o(\text{reading})$$

With this merging scheme, it is possible to refine the occupancy probabilities for the environment by moving the robot to various locations and taking new readings. Correct information will tend to be enhanced as newly acquired information is merged while wrong information will be cancelled out with the incremental integration of the correct data.

Matthies and Elfes in [17] introduce a less arbitrary approach to update the certainty grid estimates which sheds some light on the above techniques. Let's look at this approach derived from the Bayes theorem.

Each cell, C, of a map has an associated state variable, s(C). s(C) is a discrete random variable with two states, occupied and empty, denoted OCC and EMP. Since the states are exclusive and exhaustive, **P(s(C) = OCC) - P(s(C) = EMP) = 1**. Updating the occupancy grid is done with a Bayesian estimation model. A new sensor reading R_{i+1} introduces additional information about the state s(C) of a cell C in the form of the conditional probability density function, p(Ri+1/s(C)). This information is combined with the most recent probability estimate stored in the cell P(s(C)/{R}i), based on the current set of readings, {R}i = {Ri,...,R0}, to give a new estimate P(s(C)/{R}i+1), which is based on all the information available so far. In this way, occupancy grids provide a natural way to update the world model by composing information from sensor readings taken from different views.

Building maps from range data involves reasoning with uncertain pieces

of information. This can be accomplished in the following manner. For a given sensor, a probabilistic sensor model is derived in the form of the conditional distribution **p(sensor reading R/ world is in state S_R)**. The state S_k of the world model is described by the set of states of all cells in the map. In the two-dimensional case, for a map with n^2 cells, each with two possible states, it would be necessary to specify 2^{n2} cells, conditional probabilities for each reading. To avoid combinatorial explosion, it is assumed that the cell states are independent discrete random variables, so that $P(S=S_k) = P_i\ P(s(C_i))$ and the state of the map is determined by estimating the state of each cell individually.

For the updating procedures we start by applying Bayes theorem:

$$P(S_i/e) = \frac{P(e/s_i)P(s_i)}{\Sigma_j P(e/s_j)P(s_j)}$$

where s_i is one of n disjoint states being estimated, e is the relevant evidence, $P(s_i)$ is the priori probability that evidence e would be present given that the system is in state s_i. $P(s_i/e)$ is what we need for decision making, namely the conditional probability that the system is in state i in light of evidence e. In our case, the evidence is given by a sensor range reading R and the desired probabilities are P(s(C) = OCC/R) and P(s(C) = EMP/R), which we abbreviate to P(OCC/R) and P(EMP/R). Since **P(OCC/R) = 1 - P(EMP/R)**, we need to specify only one of the probabilities. For all cells in the field of view of the sensor we can express Bayes theorem as:

$$P(OCC/R) = \frac{p(R/OCC)P(OCC)}{p(R/OCC)p(OCC) + p(R/EMP)P(EMP)}$$

and for sequential updating of the map based on multiple readings, we write:

$$P(OCC/\{R\}_{k+1}) = \frac{p(R_{k+1}/OCC)\ P(OCC/\{R\}_k)}{p(R_{k+1}/OCC)\ P(OCC/\{R\}_k) + p(R_{k+1}/EMP)\ P(EMP/\{R\}_k)}$$

A similar formula can be derived for the combination of estimates provided by different sensors. Notice that $P(OCC/\{R\}_k)$ and $P(EMP/\{R\}_k)$, the current prior probabilities that a cell is occupied or empty, are taken from the existing map. The conditional probabilities p(R/OCC) and p(R/EMP) are determined from the probabilistic sensor model. Formal definitions of these conditionals may not be easy to formulate since, in general, the reading R depends on the state of the world as a whole, and cannot be specified independently for each cell.

To initialize a map we assign equal probability to each state, in other words, **P(OCC) = P(EMP) = 0.5**, which implies that the state of cell C is unknown. For this particular experimentation, the following assumption was made: **p(R/EMP) = 1-p(R/OCC).** This can be justified in some cases, but is not true in general. However, it provides us with a very simple updating formula:

$$P(OCC)/\{R\}_{k+1}) = \frac{p(R_{k+1}/OCC)P(OCC/\{R\}_k)}{p(R_{k+1}/OCC)P(OCC/\{R\}_k)+(1-p(R_{k+1}/OCC))(1-P(OCC/\{R\}_k))}$$

This formula has several useful properties:

a. It is cumulative and associative, which means that data in a multisensory system can be incorporated in any order.

b. Combining evidence, E=p(R/OCC), with the initialization probability, P(OCC) = 0.5, gives E as a result.

c. Conflicting measurements cancel: combining occupied evidence, E^+, with empty evidence, E^-, (of the same strength) produces unknown.

As mentionned before, this approach was not implemented because formal definitions of the conditional properties are difficult to formulate. Futhermore, the results obtained by Moravec using this technique in [3] are not better or worse that the probabilistic addition approach.

2.6 Map Representation Considerations

It is essential to structure information about the environment in a map hierarchy that provides adequate resolution for nearby areas and sufficient coverage for longer range planning. Therefore the mapping scheme described in sections 2.3 through 2.5 has to provide the capability to generate maps with various grid sizes. Furthermore it is preferable, for planning purposes, to have the same amount of information on each side of the robot. The robot must therefore be centered in each map and the maps must be scrolled and rotated as it moves.

The fact that the robot occupies a volume in space provides additional

information about the occupancy probability of the area it occupies. Sonar information can therefore be overwritten to reflect a minimum occupied probability and a maximum empty probability.

In order to integrate various readings or maps together, the robot relies on its dead reckoning sensors. Unfortunately the position error in a dead reckoning system is cumulative. Compensation for this shortcoming can be achieved via a blurring task which decreases the occupancy probabilities of grids that are not updated by new readings, until they vanish.

The maps created by a mapping system are the starting point for the representation hierarchy discussed in section 2.2. Elfes in [2] proposes three axis of representation for sonar mapping information; these axis are illustrated in figure 2.6.

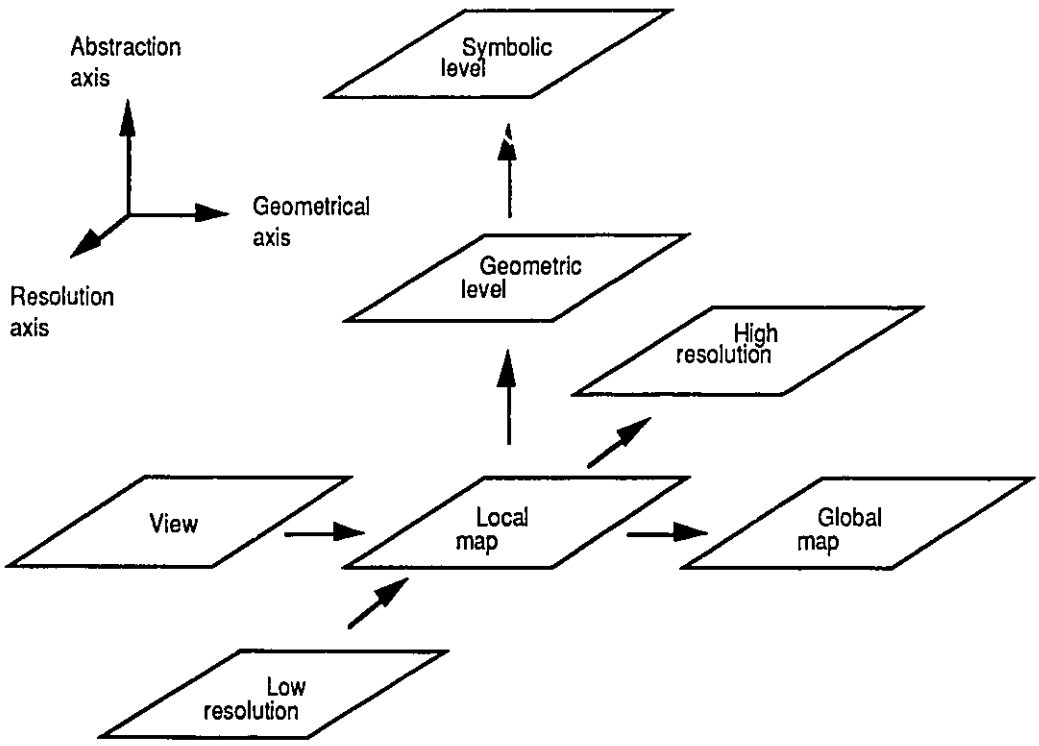


figure 2.6 Multiple axis of representation of a sonar map

Along the abstraction axis appear the abstraction levels of the data representation starting with the low level, data intensive local map. The data then migrates to the geometric level which is used to represent boundary information. The last level is the symbolic level which represents relationships between objects within the environment.

The geographical axis defines various scale of maps starting with local views obtained from one robot location. Local maps, created by incrementally integrating views together, provide information about the robot's immediate environment. Finally the global map regroups local maps together to provide

immediate and distant environment data.

The last axis is the resolution axis and includes various levels of resolution in terms of grid size, for the local map. Typically high resolution is used for local obstacle avoidance while low resolution is used to represent information kept at the global map level.

2.7 System Architecture

In order to provide a processing environment for the various functions and representations discussed in the previous sections, an architecture for a mobile robot system is introduced in figure 2.7. This architecture is based on the Dolphin Sonar-Based Mapping and Navigation System developed at Carnegie Mellon University by Alberto Elfes and Hans Moravec. It was however modified to offer a better traceability to the processing levels introduced in section 2.1 and to the data representations presented in section 2.2. It assumes a mobile platform equipped with an array of sonar sensors and with a dead reckoning system. It also assumes that motion control commands are available at higher levels to control the various robot drives and actuators.

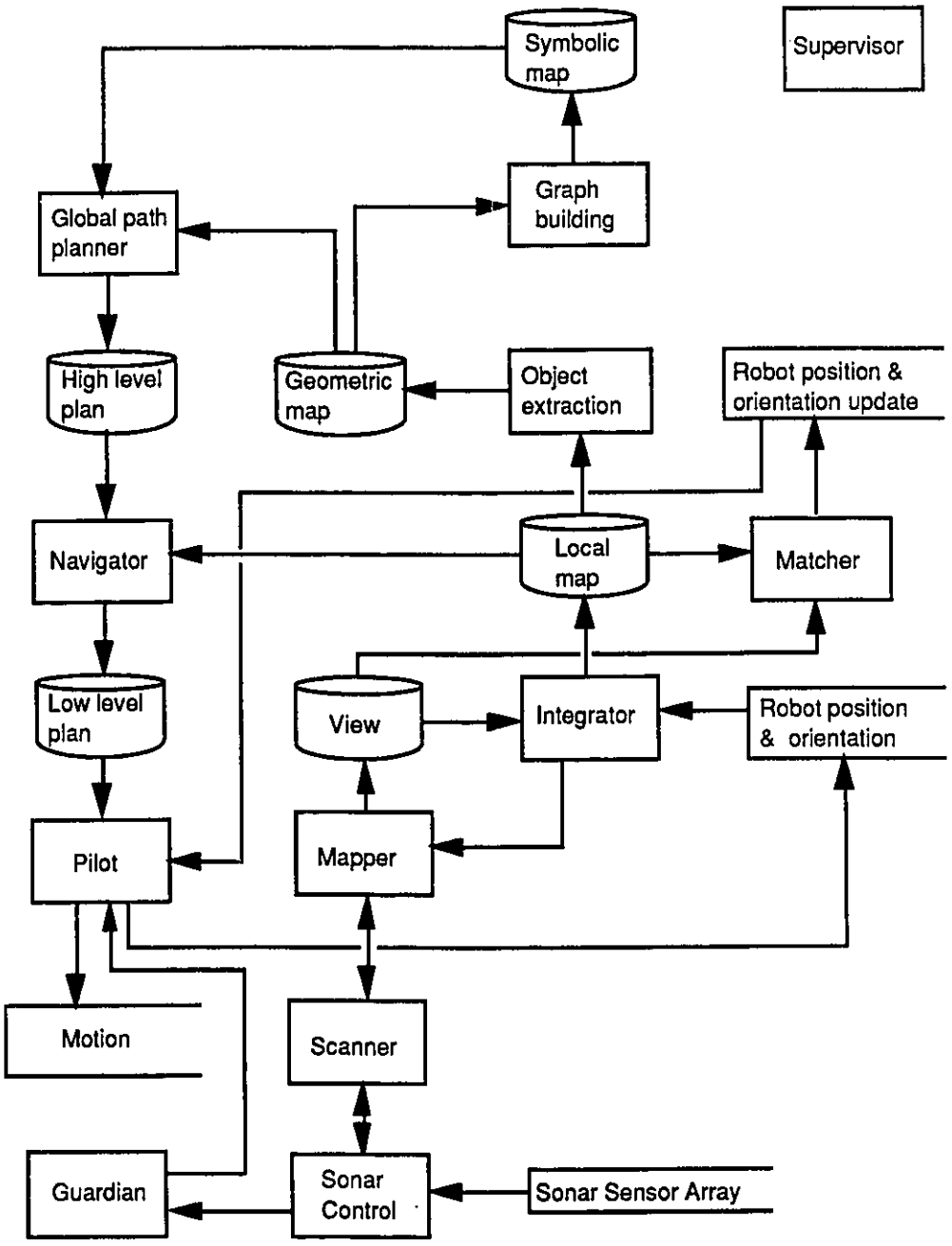


figure 2.7 System architecture

At the robot control level is the pilot module which is responsible for translating low level path information into a sequence of motion commands steering the robot along the required path. It also monitors changes in the environment via the guardian module and reacts accordingly. Finally, it monitors the dead reckoning sensors and provides position and orientation data to the rest of the system. Also at the robot control level is the sonar control module which interfaces to and runs the sonar array system and returns range readings to both the scanner and the guardian modules for interpretation.

The sensor interpretation level consists of the scanner module and the guardian module. The scanner module preprocesses and filters sonar data and sets up the various sonar control parameters in order to optimise the readings. It also commands the execution of several readings for each location and averages them. The guardian module simply monitors the sonar looking in the direction of motion and reports any changes in the environment to the pilot module.

At the sensor integration level, preprocessed and averaged readings from each of the sensors in the array are integrated by the mapper module into a view corresponding to the current robot position.

Three functions are carried out at the real world mapping level. The integrator module takes the newly acquired view and integrates it into a local map. The matching module computes a position and orientation update for the dead reckoning system. Finally the object extraction module generates geometric information about occupied areas and builds a geometric map reflecting that information.

The navigation level capabilities are provided by the the navigator module. It generates a low level plan consisting of pilot level commands to steer the robot to intermediate or end goals. The intermediate goals are generated via

local obstacle avoidance algorithms using input from the high level plan and the local map.

At the global path planning level are the graph builder module and the global path planner module. The graph builder module generates a symbolic description of the environment using the geometric map as input. The global path planner uses inputs from the symbolic and geometric maps to generate a high level plan that consists of intermediate or end goals that converge toward the overall task completion.

The supervisor module corresponds to the control level and provides the scheduling functions for plan driven or data driven activities. In order to accomplish this it uses input from the outside world and system status information acquired via supervisory functions.

This architecture is theoretical and does not constitute the only method of achieving sonar-based navigation. Although geared towards sonar exclusively, by funnelling readings through a certainty grid, it could be easily adapted to other types of sensors. Indeed the system would be unique to each sensor type up to the sensor integration level but from then on would be common rendering, the execution independent of the sensing.

3. Design and Implementation

3.0 Introduction

The design and implementation of the mapping system reflects the study and analysis that were carried out to identify the requirements for a system providing a mobile platform with the first four processing levels of a navigation system. These requirements are discussed in section 3.2 as they relate to the laboratory configuration identified in section 3.1.

The system design through development followed a strict software development methodology which is described in appendix A. Such a method provided a very useful tool in converting the requirements into an accurate design and in turn, converting that design into reliable code. Structured program design was achieved through effective program decomposition, the reasons for a modular design are also discussed in appendix A. The resulting system architecture appears in section 3.2 and the design of its main modules is described in sections 3.3 through 3.6. Finally the application framework in which the mapping system was implemented is described in section 3.7.

3.1 Laboratory Configuration

The implementation of the mapping system is part of a project at the National Research Council to improve the mobility of disabled persons by designing a platform that can navigate independently and have the ability to transport a person to a desired destination. This section describes the laboratory configuration available at the National Research Council facilities for the development effort. The configuration is responsible for many decisions made in the design phase. It was also closely tied in with the requirements discussed in the next section. The configuration can be broken down in two parts; the first

one being the computer architecture and the second one the computer controlled platform.

To achieve an adequate level of performance and safety, the National Research Council decided to control the sensor-based mobile vehicle with a real-time computing system. Such a system is characterised by an ability to respond in a rapid and predictable way to unexpected events that may occur within the environment. In order to be useful in an experimental environment the system must be very flexible and, as new sensors are added, it must also be expandable to meet the computing requirements.

Research at the National Research Council over the last few years has led to the development of the Harmony real-time multi-processing operating system. Harmony was designed specifically to address the issues of real-time control. Two of its principal advantages are that it offers transparent multi-processing and that it supports the custom interfacing of peripherals.

The actual hardware configuration on which Harmony runs is a 68020-based single board computers operating on a VME bus as shown in figure 3.1. A separate bus architecture based on the VSB subsystem bus is used for communication with the various sensor systems that are being developed. The initial development of the sensor-based application is done on MacIntosh II computers which are directly interfaced with the sensor system. When development is completed, the code is re-hosted on one or more of the single-board processors in the parallel computer architecture.

The computer controlled platform purchased by the National Research Council is the K2A vehicle shown in figure 3.2, this platform was specifically designed and manufactured by Cybermation for the research community. The

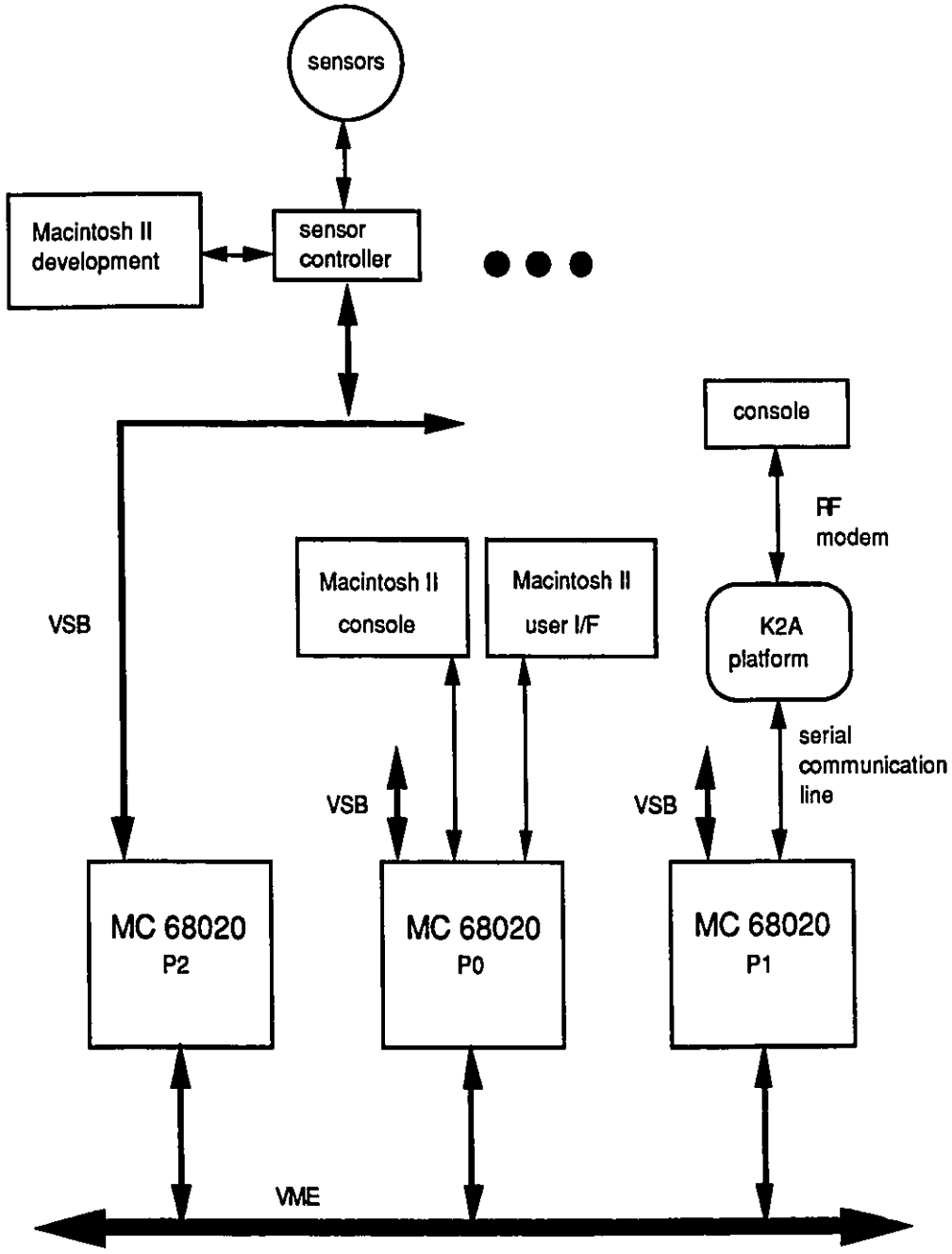


figure 3.1 Harmony-based parallel system architecture

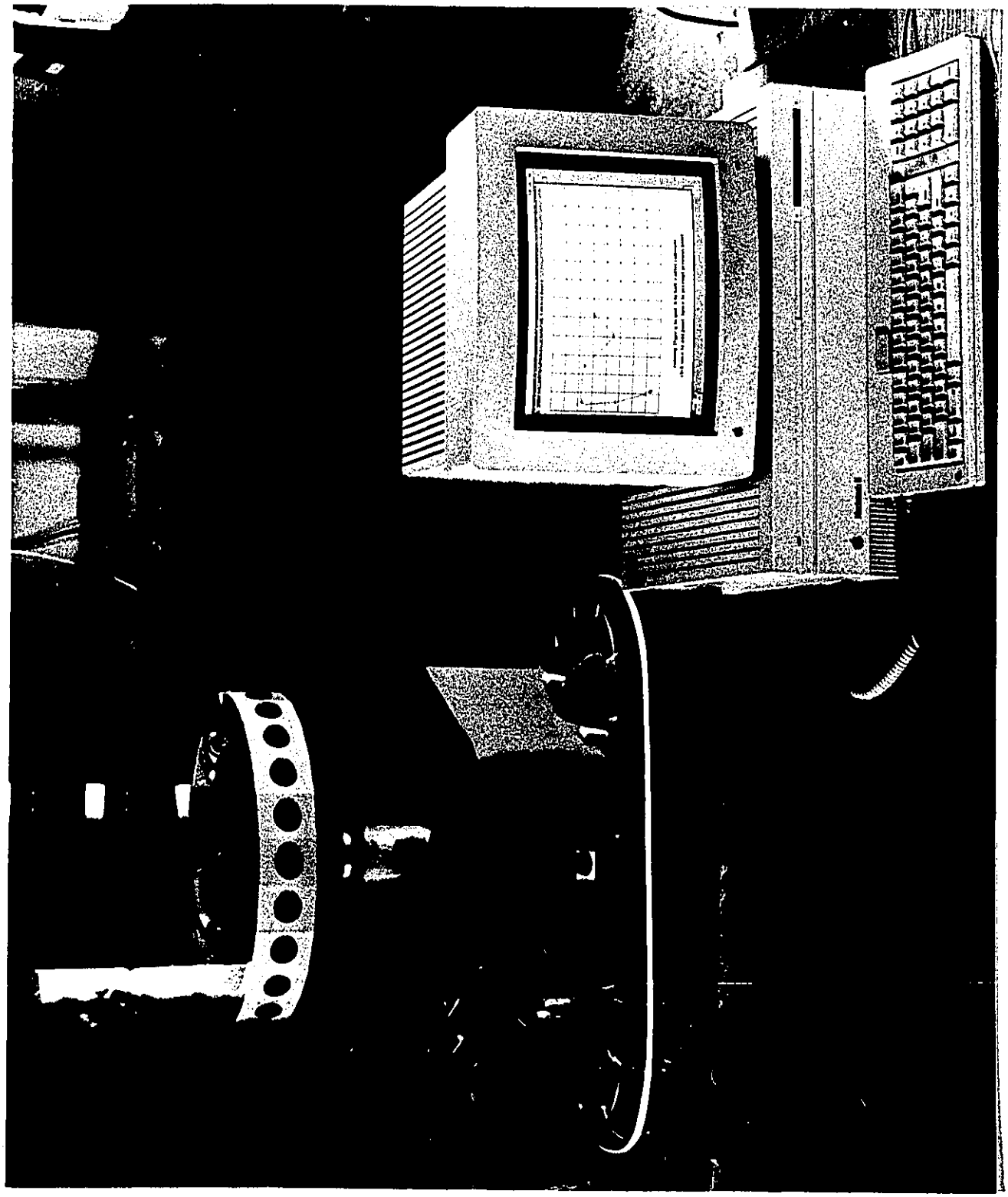


figure 3.2 Mobile platform and sonar ring

platform has a synchro-drive mechanism where the three wheels are synchronously driven or steered. This drive mechanism has the advantage that the body does not rotate with the vehicle, giving it a zero drive turning radius capability. The disadvantage is that the body of the vehicle has no front or back and for that reason there is a protruding vertical turret that rotates with the wheels but not with the body. Figure 3.3 provides a sketch of the platform. The synchro-drive has been implemented using a direct drive shaft coupling minimising the errors that are encountered through slip and misalignment normally experienced when using chains or belts for coupling.

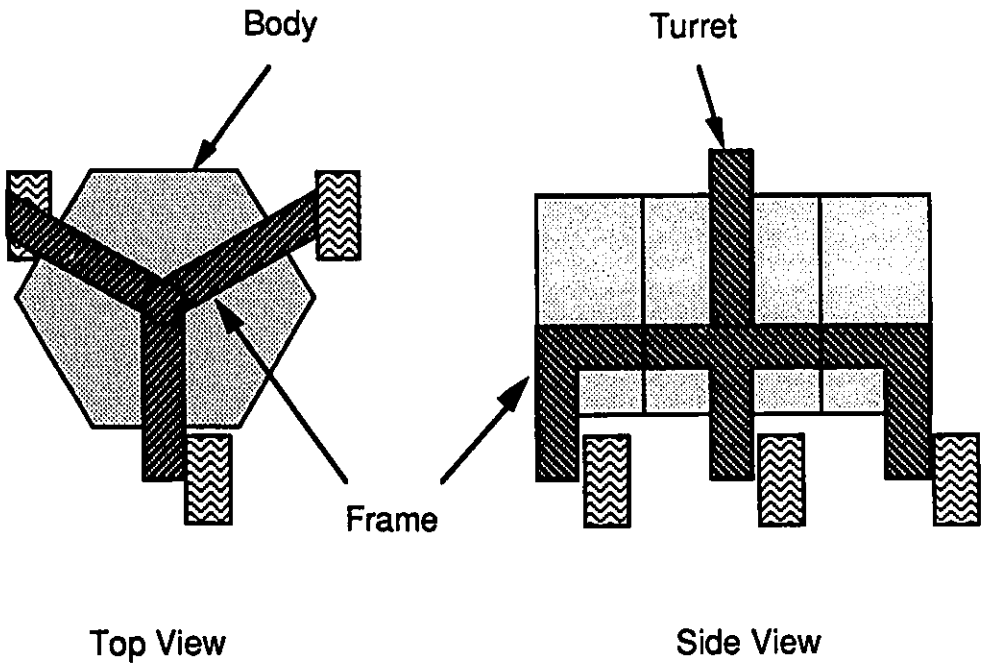


figure 3.3 K2A vehicle body layout

The platform comes equipped with a very accurate dead reckoning system due in part to the type of drive mechanism. It also reports drive and steering information in addition to the dead reckoning data to a console. The console consists of a personal computer with a colour monitor which communicates with the vehicle via an RF modem and provides a joystick interface for user motion control (see Figure 3.1).

The platform was fitted with a ring of twenty-four ultrasonic range sensors (sonars) spaced evenly around the girth as shown in figure 3.2. The sonars are grouped in three banks of eight sonars and each bank can be controlled separately allowing preprogrammed firing sequences for each bank to fire simultaneously. The sonars used are polaroid transducers with an adjustable maximum range between one and seven meters, depth/gain compensation, near/far mode and a 50 kilohertz CW burst (0.16ms or 1.28ms). The controllers consist of a printed circuit board in an aluminum chassis providing the capability to control and interface with the sonars. The sonar ring is physically located on top of the vehicle at approximately 1.3 meters from the ground.

There is currently an on-going effort to integrate a laser slit scanner with the vehicle to overcome the problems associated with the ultrasound sensors namely specular reflection and wide beamwidth.

3.2 Design Requirements

According to the software development methodology described in appendix A any software project should start with an analysis phase resulting in a set of well-defined requirements. Such an analysis was carried out to scope the size of the effort that would be involved in designing a complete navigation system for a mobile robot that would fulfil the National Research Council's requirements.

These requirements were as follows:

a. the mobile robot must be able to navigate in a known or unknown environment,

b. the mobile robot must have the ability to assess its environment so that the desired task can be performed, and

c. the mobile robot must be able to locate a human "commander".

The scope of these high level requirements covered the seven processing levels identified in section 2.1 and after evaluation of the options, it was decided that a sonar-based real world mapping system implementing the first four processing levels would be developed. Such a system would provide a basis for the implementation of a navigation system while helping to assess potential risk areas. The requirements for that scoped down effort were then further refined to the following:

a. the mapping system will be easily adaptable to various types of sensor data (mainly for eventual integration of laser slit scanner data),

b. the mapping system will not use a priori knowledge about its environment,

c. the mapping system will implement a hierarchy of map as discussed in section 2.6,

d. the mapping system will use a certainty grid based representation.

Because of the requirement in (a) and because of the technical advantages described in chapter 2, certainty grids were felt to be the best representation to adapt to uncertainties inherent to sonars,

e. the mapping system will be implemented using the application development configuration as described in the previous section,

f. the mapping system won't provide any object extraction functions. The main reason for that decision is that since the requirements for a higher level module that would use that information are not defined, it was diffucult to define requirements for object extraction,

g. since the goal of the mapping system is to translate sonar data into a real world map of the robot's environment, there is no requirement to implement robot motion control functions. This means that development at the robot control level will only include the sensor control functions.

Based on these requirements the modular architecture in figure 3.4 was designed. This architecture is in fact a subset of the overall navigation architecture presented in section 2.7 where the functions carried out by each module are the same.

Since development is to be done on a Macintosh II computer directly interfaced to the sonar ring, there is no interface with the platform. The dead reckoning data will therefore have to be read from the console and manually fed into the mapping system. Furthermore, the system will use existing Sonar Control routines developed by the National Research Council that control the

execution of the various sonar system modes of operation. These routines were developed using the C language and, for compatibility reasons, the mapping system will also be developed in C. Finally, the development effort will not include integration of the end product into the Harmony based system however it will include integration in a user-friendly Macintosh II application which shall provide visibility of the data at every level of processing.

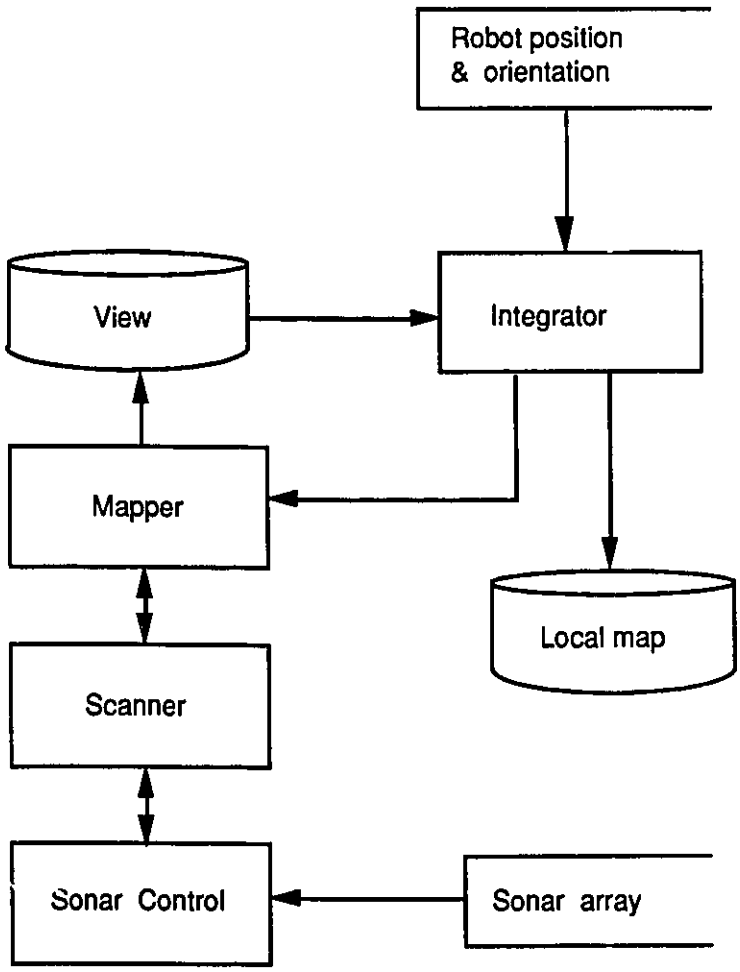


figure 3.4 Mapping system architecture

3.3 Sonar Control Module

The Sonar Control module is responsible for carrying out the sensor control functions of the robot control processing level. The motor and actuator control functions that will eventually be required in a full navigation system were not considered within the scope of the mapping system, due to the development system configuration used which does not interface with the platform. However the existing motion control functions that allow user control of the robot will be used independently of the mapping system to assist robot positioning, which is essential for operation and testing purposes. The internal sensor monitoring functions, such as dead reckoning, were not included in the development effort but will also be used during operation to provide data for manual user input.

The development effort concentrates on the functions required to trigger the sonar controller routines that are necessary to acquire data for the mapping system. The National Research Council has developed a set of routines aimed at providing interactive user-friendly control of the sonar controller functions. These functions are implemented by the DoCommand routine and are as follows:

1. Stop auto scan
2. Start auto scan
3. Perform a single scan
4. Set the maximum depth setting
5. Set the excitation delay
6. Set the mode and gain
7. Set the firing pattern (used on a full scan)
8. Find the closest object
9. Resend data from last full scan

10. Excite a single transducer
11. Warning beep
12. Get the operating parameters
13. Flush the data transmit queue
14. Reset the operating parameters of the controller to the default values
15. Do a hard reset of the controller

Of these functions, only the setup functions (functions 4 to 7) and the "perform a single scan" function (function 3) are relevant to the mapping system. The DoCommand routine simply consists of the DoCommand function which in turn consists of a case statement that carries out the processing task associated with the function (or command) number passed as an argument in the function call. The command number is the only argument passed to the function. The DoCommand routine does not communicate directly with the sonar controllers, instead it calls up the ControllerInterface routine to handle that low level task. After processing a command at the DoCommand routine level, the calling routine must make sure that the buffer used to transfer data between the ControllerInterface and the sonar controllers is cleared. Results, set up parameters and handshaking parameters are propagated using global data declaration and are therefore available for monitoring and/or modification at every processing level which executes the appropriate external declaration routine.

The second routine in the Sonar Control module is the ControllerInterface routine which consists of several low level sonar controller functions. These functions are ReadSerial, WriteSerial, ReadData, ReceiveAck, ReadSingle, ReadScan, and GetStatus. ReadSerial reads a 16 bit word from the serial port and returns TRUE if successful and FALSE if not. WriteSerial writes a 16 bit word to the serial port and returns the same status as ReadSerial. ReadData performs the

same function as ReadSerial but breaks down the 16 bit word so that bits 11 to 15, which represent the transducer number, get stored in the first cell of a row of a 2 x 24 array and bits 0 to 10, which correspond to the depth count, get dumped into the second cell of the same row. ReceiveAck returns TRUE if the acknowledgement of the transmitted command is received and false if not. ReadSingle reads a single sensor value. ReadScan reads the data received on a full scan which consists of one reading per sonar. ReadSingle returns TRUE if data is read and FALSE if not. ReadScan does not return anything. GetStatus gets the four status words from the serial controller. As in the DoCommand routine, the ControllerInterface routing provides more than what is required by the mapping system but since it was possible to use these two routines as developed, it was decided not to modify them. Since these routines were "off-the-shelf", development details will not be discussed any further.

3.4 Scanner Module

The Scanner module is responsible for invoking the Sonar Control module routines to set up the required sonar controller parameters and trigger scans in both far field and near field modes. In each of the two modes, three readings are taken for each sonar in the ring. These readings are formatted, averaged and stored into 24 x 2 arrays which represent the sonar position and the range reading for each sonar. The two arrays obtained (one in each mode) are then processed to select the best possible reading.

The Scanner module is made up of two routines, the main routine is called Scan. The top level flowchart for the Scan routine is shown in figure 3.5. First the gain profile is set up for maximum return. Then the maximum depth is also set to its maximum (7.02 meters). The firing delay is set to half of its maximum. Better results are obtained from a bigger delay value but acceptable readings can be obtained with the mid-range setting. The values chosen for these

parameters were determined experimentally and yield the most accurate results. Once all the parameters have been set, the range mode parameter is first set to near mode and a single scan is triggered by calling up the TrigScan routine.

TrigScan is the other routine in the Scanner module. It is responsible for acquiring the results of three scans along with the number of not out-of-range returns for each sonar. The top level flowchart for the TrigScan routine is illustrated in figure 3.6. It begins by initialising the arrays for storing the results of the three scans. It then goes into a loop in which a single scan is executed and sonar readings from the same sonar are added if they are not out of range otherwise the not-out-of-range return counter is incremented by one. Before it exits, TrigScan computes the average reading for each sonar and returns a 24 x 2 array containing the averaged readings and their associated sonar numbers. The 24 x 1 array that contains the number of not-out-of-range counts is also returned.

Scanner transfers the two arrays produced by TrigScan into near mode specific arrays. It then sets the range mode parameter to far mode and performs the same processing as for the near mode. The resulting arrays are transferred into far mode specific arrays. The next step consists of selecting between the far and near mode values. In order to do this, the following conditions are applied:

a. The reading value obtained in the mode that returned the most readings is selected
b. If in both modes, the resulting range value is out of range, the maximum range value is attributed for that sonar reading.
c. If the number of counts is the same then the near mode reading is selected if the range reading is below 400 centimetres. otherwise the far mode reading is chosen.

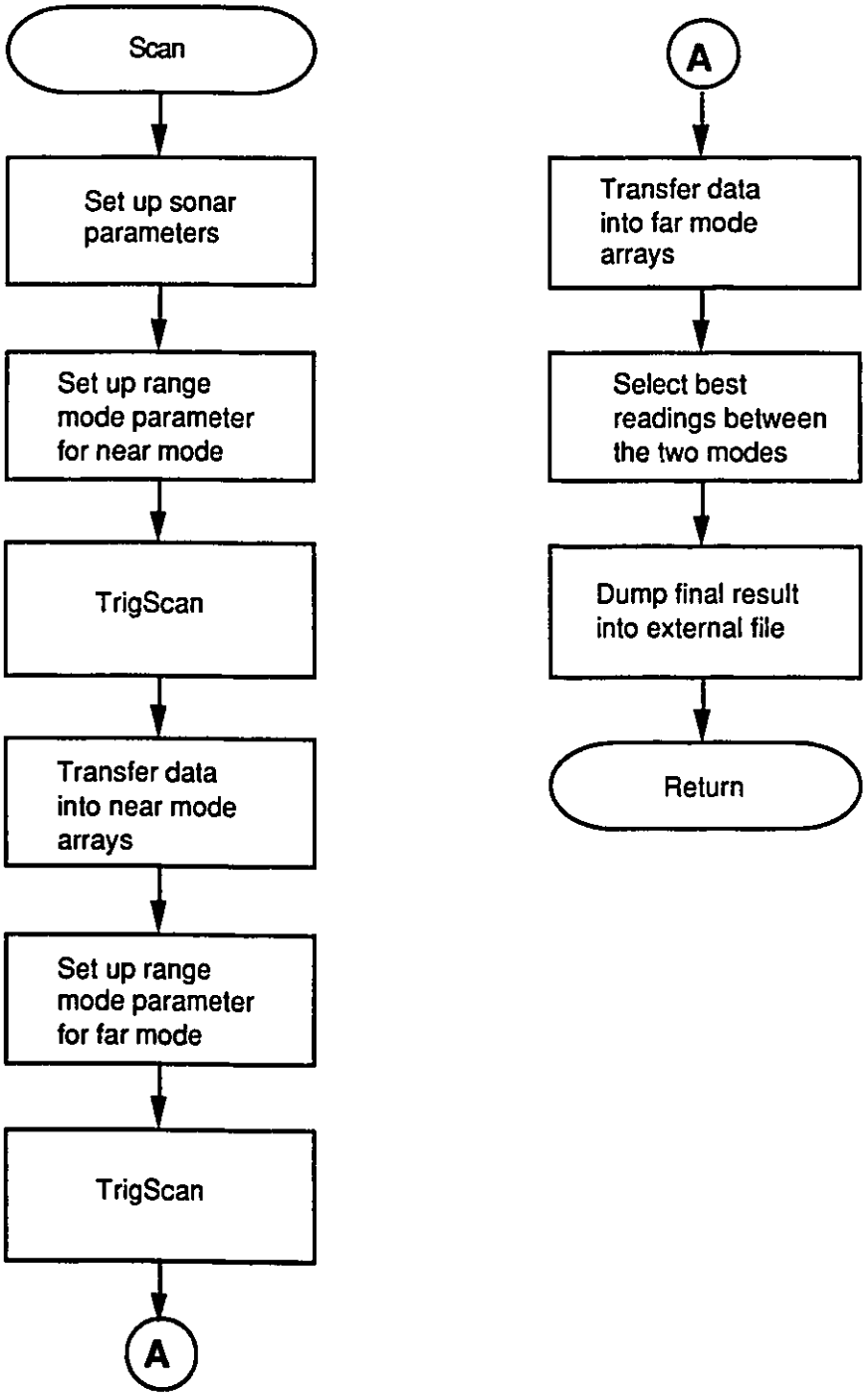


figure 3.5 Scan top level flow diagram

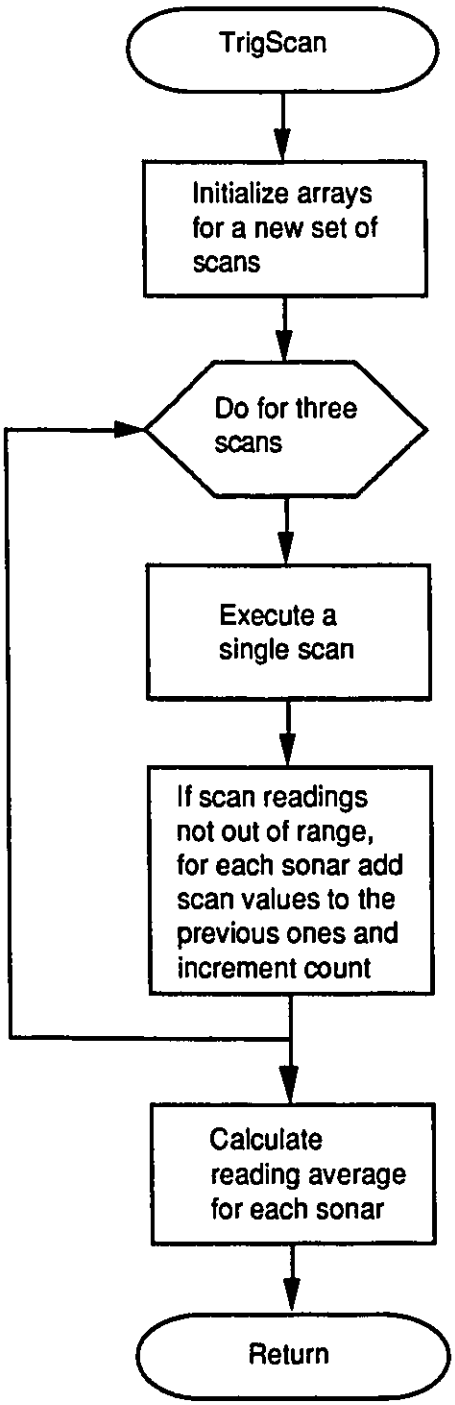


figure 3.6 TrigScan top level diagram

It is important that these conditions be applied in the order presented. This order corresponds to heuristics determined experimentally that yield the best results. The resulting 2 x 24 array is then dumped into an external file for post operation analysis. The results in the array appear in centimetres for the range reading and in degrees for the sonar relative position.

Both the Scan routine and the TrigScan routine invoke the sonar controller DoCommand routine for setting up the parameters and triggering the scans. The parameters are declared as global variables with the command number passed as a parameter. Communication between the Scan routine and the TrigScan routine is also done via global variables and does not require parameter passing. Finally, every time the DoCommand routine executes, Scan and TrigScan must clear the Sonar Control buffers upon reception of the acknowledgement signal.

3.5 Mapper Module

The Mapper is at the heart of the mapping system. It is responsible for generating a certainty grid based view of the robot's environment for a specific robot location. In order to implement a variable data resolution capability along the resolution axis (figure 2.6), the Mapper module operates with a selectable grid size. The final view is produced from two arrays: the probably empty array and the somewhere occupied array. The arrays are first initialised to the unknown state and then each array cell, which corresponds to a grid in the view, is updated based on its position in relation to the sonar cones in the sonar ring system and the area occupied by the robot.

The Mapper module consists of four routines. The top level flowchart for the main routine which is called Mapper is shown in figure 3.7. Mapper starts by initialising three arrays (Probably Empty, Somewhere Occupied and Final

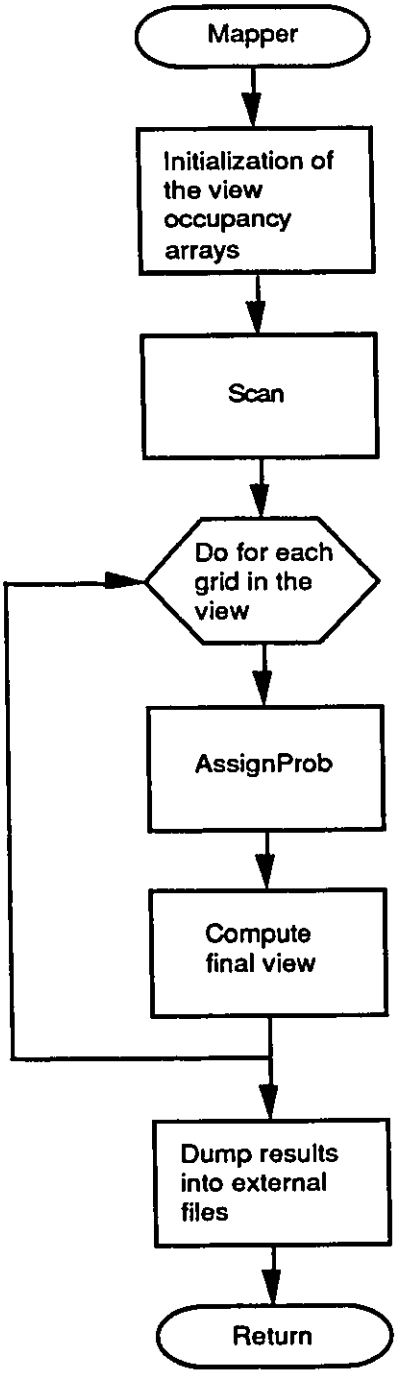


figure 3.7 Mapper top level diagram

View) to the unknown state. It then invokes the Scan routine of the Scanner module to acquire the scan results for the current robot position. When Scan returns Mapper executes a two-index loop to visit each grid in the view. The number of grids in the view is passed down by the Integrator module and is selected to optimize processing time while obtaining sufficient sensor coverage. After experimentation, it was found that a 17 by 17 view accomplishes the desired objectives. Each of the three arrays is indexed so that the view is centered within a Cartesian x-y coordinate system in which x and y correspond to the loop indices. Within the loop the AssignProb routine is called. AssignProb returns the empty and occupied probabilities for the grid whose coordinates correspond to the current loop indices. The Final View is also computed within that loop simply by comparing the occupied and empty probabilities returned by AssignProb. The larger of the two values is assigned to the Final View array. Mapper then dumps the Somewhere Occupied and Probably Empty arrays into an external file along with a symbol representation of the Final View array.

AssignProb accepts the loop indices passed as parameters from the Mapper routine. It first transforms the loop indices into centimetres and then sets up a default beam value which is used only if the grid lies between two sonar cones. It then verifies if the resulting coordinates fall within a grid area occupied by the robot. If so, the Somewhere Occupied value is set to 0 and the Probably Empty value is set to 1. If not, the appropriate sonar beams are selected. If the grid lies directly on one of the sonar beam main axes then only that sonar is selected. If not, the two sonars on each side of the grid are selected. ComputeProb is then called to compute the empty and occupied probabilities in relation to the selected sonar beams. The top level flowchart for AssignProb is shown in figure 3.8.

The ComputeProb parameters consist of the loop indices and their centimetre equivalents. If two beams were selected by AssignProb, ComputeProb

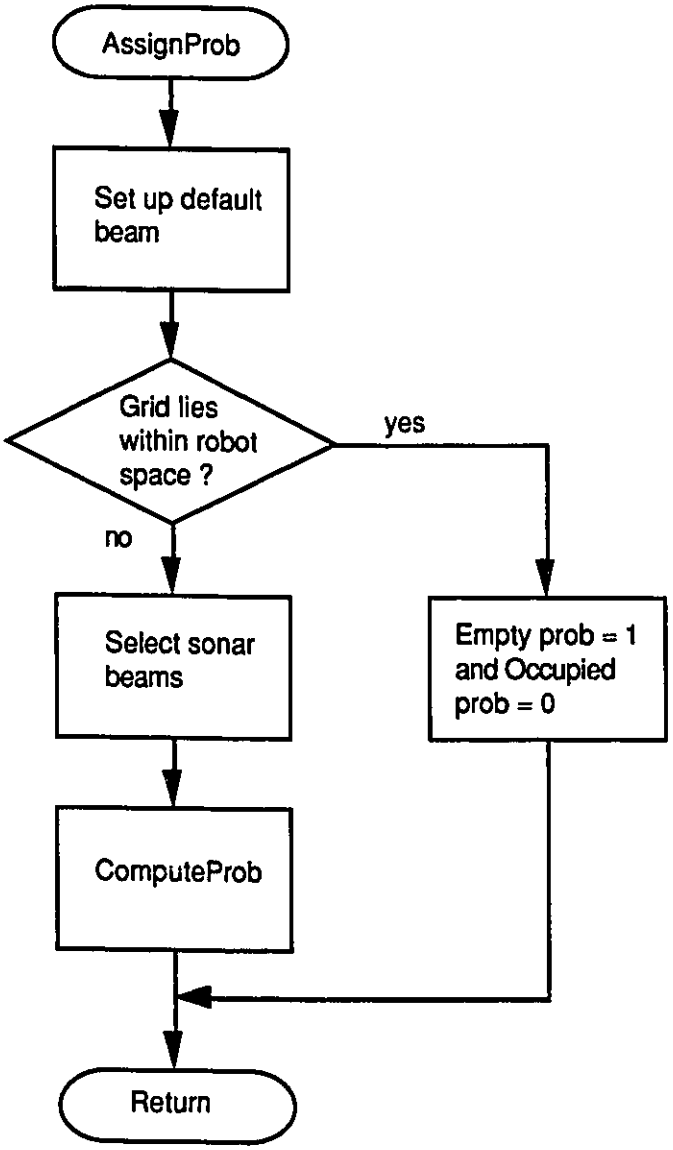


figure 3.8 AssignProb top level diagram

calls BeamProb to calculate the occupied and empty probabilities of the grid being processed in relation to the first beam. It directly assigns these probabilities to the Somewhere Occupied and Probably Empty arrays. It then invokes BeamProb again with the same grid but in relation to the second beam and

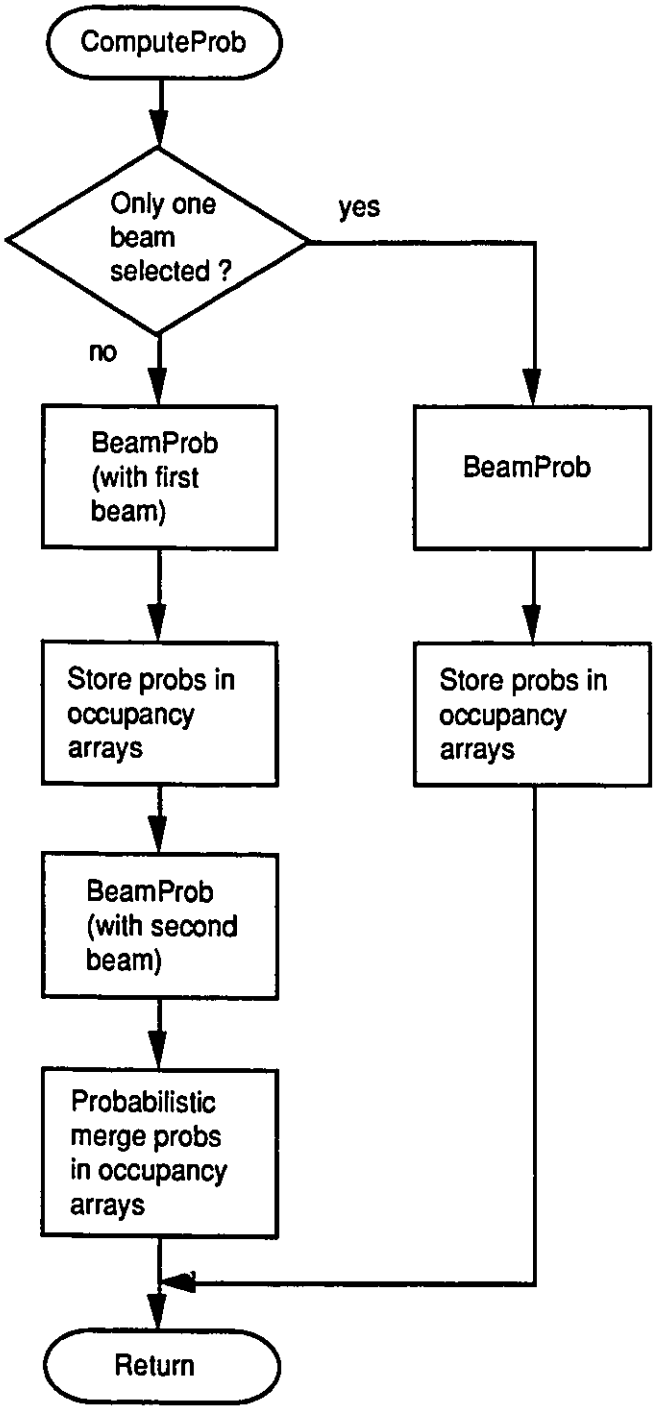


figure 3.9 ComputeProb top level diagram

probabilistically merges the new occupied and empty probabilities in the occupacy arrays in accordance with equations 2.17, 2.18 and 2.19. If only one beam was selected by AssignProb, BeamProb is called with the selected beam and the probabilities are directly assigned to the occupacy arrays. Figure 3.9 shows the top level flowchart for the ComputeProb routines.

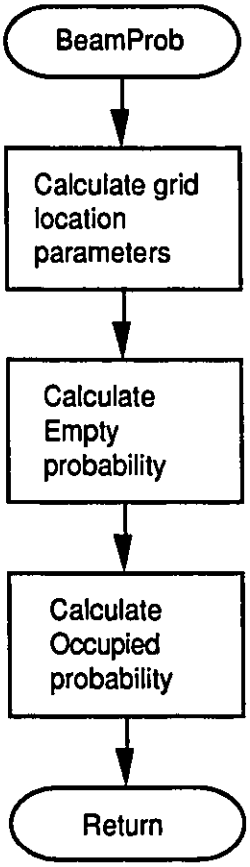


figure 3.10 BeamProb top level diagram

BeamProb accepts as parameters the number of the sonar selected and the grid coordinates in centimetres. It starts by calculating the various grid location

parameters as required by equations 2.3 and 2.10. These are: d, the distance between the sonar position and the grid, q, the angle formed by the grid, the sonar and the sonar main axis, and finally e, the relative error. With these parameters, the empty probability and the occupied probability for that grid in relation to the selected sonar beam are calculated. Figure 3.10 shows the top level diagram for the BeamProb routine. The beamspan and the minimum range values from equations 2.3 and 2.10 were chosen as constants based on data from the manufacturer, however they could be implemented as functions of other sonar parameters. The value for the relative error was set to 10% instead of the 5% claimed by the manufacturer. This yields a wider occupied region and therefore increases the safety margin. The resolution obtained using such a relative error provides enough accuracy for navigation purposes. If more is required, the relative error could also be implemented as a function.

3.6 Integrator Module

The Integrator module performs a dual role in the mapping system. First of all, it is responsible for merging new views produced by the Mapper module into a local map. As illustrated in figure 2.6, the local map is the only representation common to all three axes and therefore a significant amount of information can be extracted from it. The robot always remains centered with the local map and its orientation decides the layout of the local map x and y coordinates. The grid size is selectable but once set, it remains constant for the duration of the session. The Integrator module's second function is inherent in the fact that it is the top level module of the mapping system and therefore is responsible for the user interface which in fact is handled by calls to some of the application routines discussed in the next section.

The Integrator module consists of two routines, the main one is called Integrator. Integrator starts by requesting a grid size from the user. It then

initialises three arrays that are to be used to define the local map. These arrays are the Probably Empty array, the Somewhere Occupied array and the Final View array. Their cells are of the same format as the views but their dimensions are larger. The initialisation only takes place at the beginning of the session in order to allow the incremental build up of the local map from different views. Integrator then starts its main loop which continues until the user does not want to add another view. In the loop body, the user is requested to enter the robot position and orientation. The position data is relative to the robot's origin position which ideally should be set to 0,0 at the beginning of a session to help visualising the mapped results. The position translation is expressed in robot counts for the x and y coordinates as displayed on the K2A platform console. The orientation variations are also expressed in robot counts in relation to the origin. At the moment the sonar ring is mounted on the robot body and, as mentioned in section 3.1, always bears the same orientation, however Integrator was designed to process rotational shifts. The translation and rotation counts are transformed into centimetres and degrees respectively. The position data, read from the console is provided by the dead reckoning system. In normal operating mode, this information would be read by the mapping system directly from the internal sensor interface port but since the robot is not interfaced with the sensor development system, this task is performed manually.

Next the Mapper module is called for the acquisition of the view for the current robot position. Integrator sets up the value for the number of grids in both the local maps and in the view. The local map size should completely cover the robot's directly accessible environment. Processing time is however proportional to the size selected. When Mapper returns, the MapGridProb routine is called to merge the new view and the local map into temporary occupancy arrays. When MapGridProb returns, these arrays are transferred back to the local map arrays and the Final View array is calculated by comparing the empty and the occupied probabilities. If the empty probability is larger than

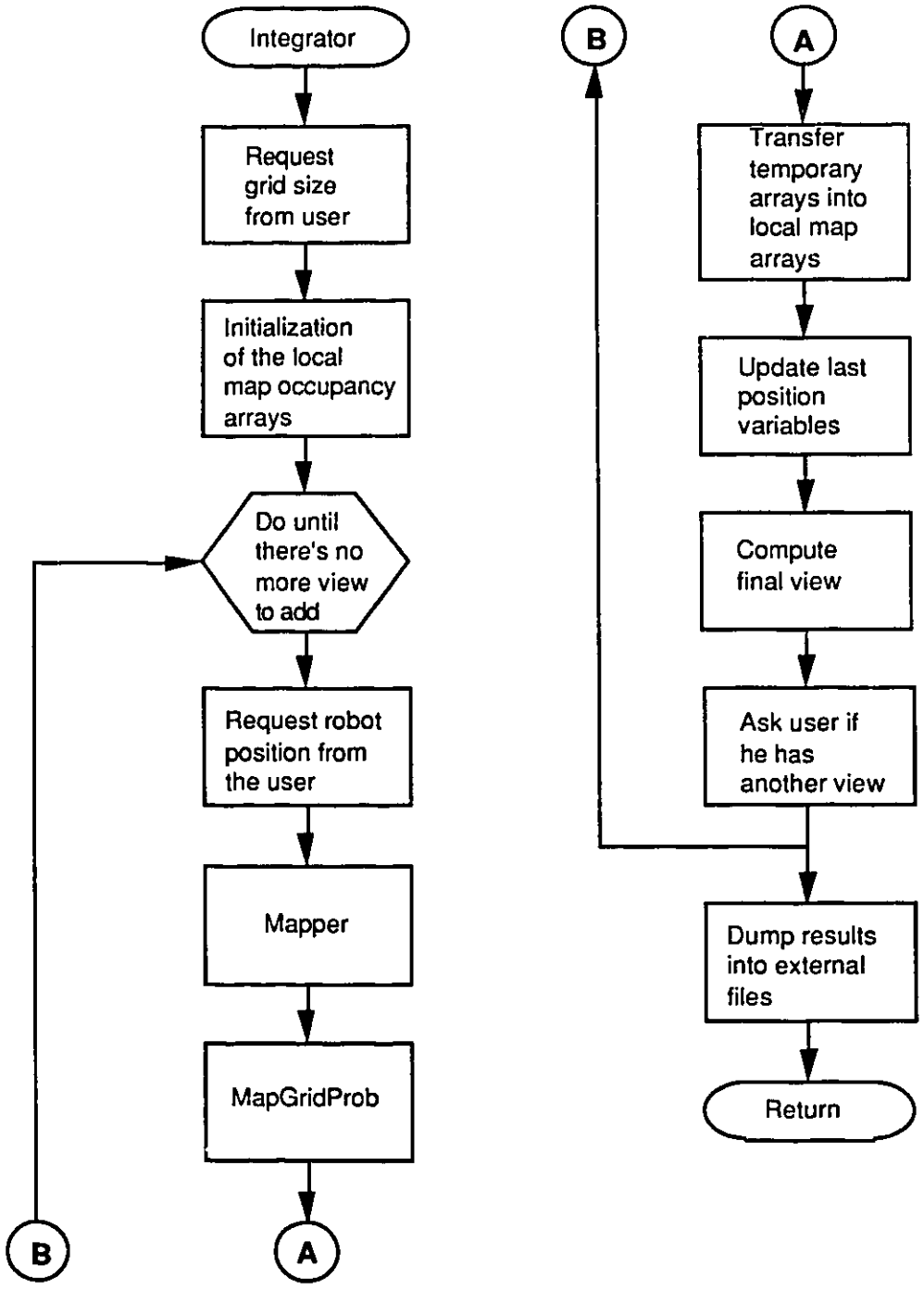


figure 3.11 Integrator top level diagram

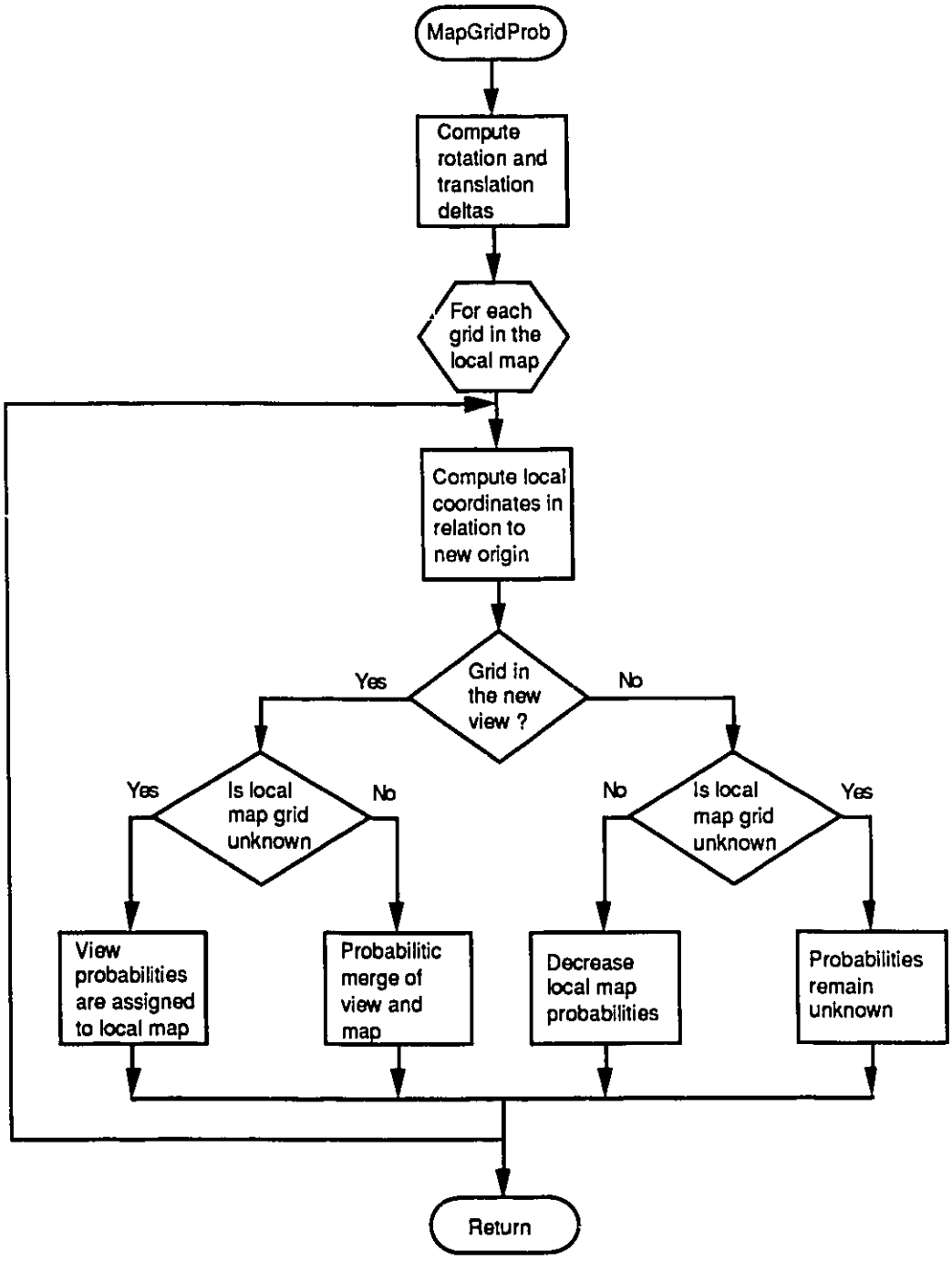


figure 3.12 MapGridProb top level diagram

the occupied one, its sign is reversed and it is assigned to the Final View array. This procedure is identical to the one that is used by the Mapper module. Before exiting or starting the loop again, Integrator updates the screen display, dumps it into a file and then asks the user if another view is required.

Integrator concludes its processing by dumping both the Somewhere Occupied and the Probably Empty arrays into external files along with a symbolic representation of the Final View array.

MapGridProb is the second routine in the Integrtor module. It is responsible for merging the new view and the existing local map into temporary arrays. It begins by computing the translation and rotation deltas from the last robot position. It then enters a loop that visits every grid in the local map. In the loop body, the array indices are computed from the local map as they register to the new origin. The view array indices do not need to be adjusted as the view always registers with the robot axes. Once the indices have been calculated MapGridProb proceeds to merge the view and the local map into temporary arrays. Four possibilities exist. First, if the local map shows an unknown status for grids in the view that show an occupancy status, the occupancy values are directly assigned to the arrays. If both the view and the local map show an occupancy status, then the probabilities are merged probabilistically as per equations 2.17, 2.18 and 2.19. On the other hand, if the view shows an unknown status for a grid location that shows an occupancy status in the local map, the local map probabilities are decreased by ten percent. This implements the blurring operation described in section 2.6. Finally, if both the view and the local map show an unknown status, the probabilities remain unknown. The top level flowcharts for the Integrator and the MapGridProb routines are shown in figures 3.11 and 3.12 respectively.

3.7 Application description

The Integrator, the Mapper and the Scanner modules were implemented to operate in an environment developed by the National Research Council to assist user interface functions. The application implementing this enviroment, including the mapping system, is shown in figure 3.13.

At the top level is the Sonar module which is responsible for global initialisation functions including graphic and printer set up and data structure memory allocation. As the top level module, Sonar invokes its sub-level modules for handling the lower level processing tasks. These sub-level modules are SonarInit, DoMenu and the system routines. Sonar is also responsible for tracking mouse and keyboard inputs from the user and for translating them into the appropriate action. The specific initialisation processing is performed by the SonarInit module which, among other things, is responsible for the sonar buffer allocation. The various routines in SonarInit are called up from Sonar as part of the global initialisation function.

There are two types of events reported by the tracking routines in Sonar. The first type includes all the application related events and the second type consists of all the system related events. The system events are the events that call the Macintosh system routines and therefore are handled by the system. The application related events result in a call to routines in the DoMenu module. DoMenu consists of a collection of utility routines that handle the menu selection functions. It also processes and distributes data that is passed back to it from its sub-level modules. Among these sub-level modules is the DoCommand module discussed in section 3.4 and the Scanner, Mapper and Integrator modules from the mapping system. Data transactions between the Scanner module and the DoCommand module are direct and do not go through the DoMenu module. The execution of all the DoMenu sub-level modules can be commanded from one or

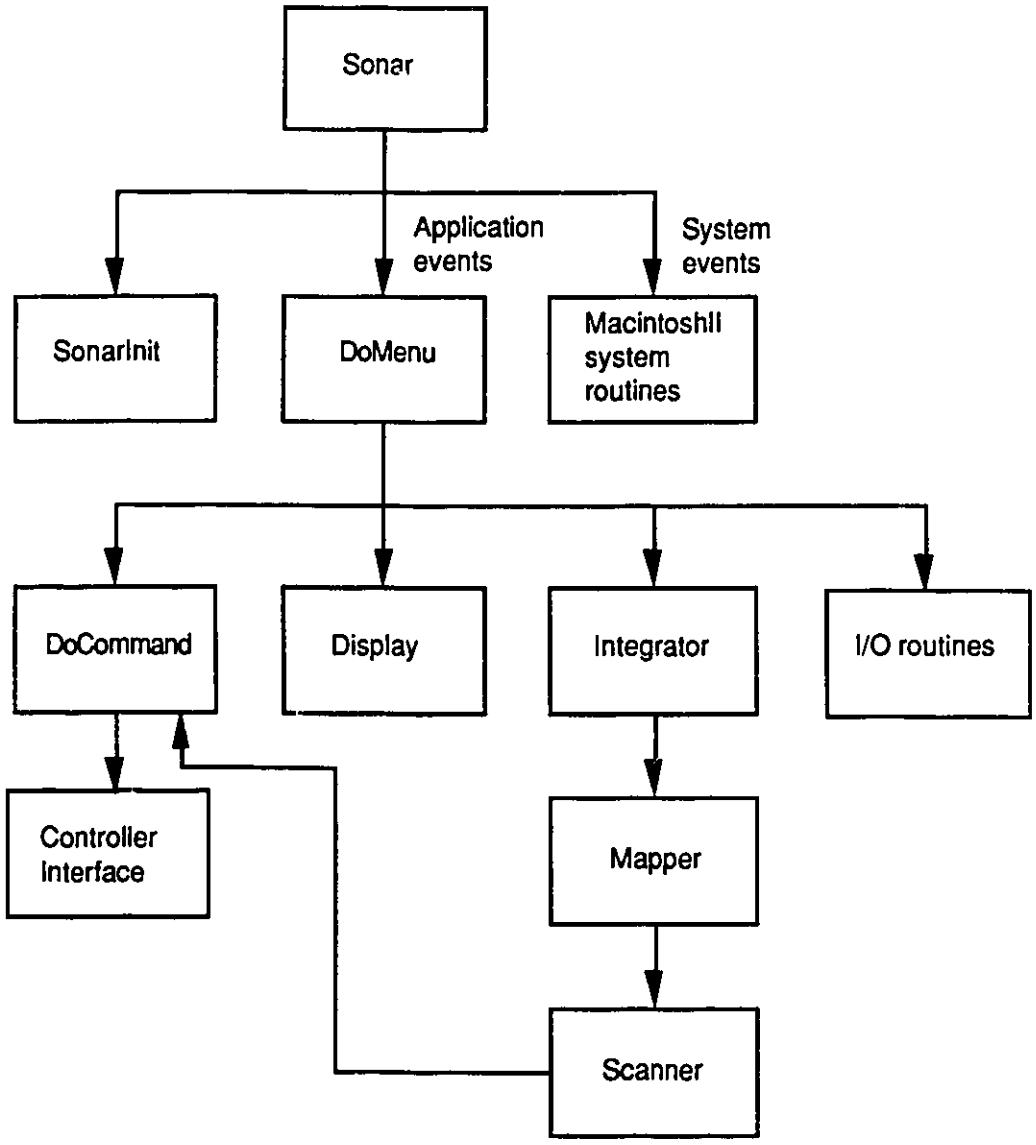


figure 3.13 Application organisation

more menu item, therefore a module can be invoked alone from a menu selection or it can be called up from another module . The two other DoMenu sub-level modules are the Display and the I/O routines modules. Display is responsible for all the data display functions available within the application such as the ones

used to display the results of the mapping system. Display also includes the routines required to handle the dialog boxes like the ones used for user input in the Integrtor module. The I/O routines module consists of three routines which handle disk I/O functions. The first routine called GetMap, copies a room map in a MacDraw format from the disk and feeds the data back to DoMenu which displays it calling up routines in the Display module. The second routine is called SpoolPicture and allows the user to spool the output graphics window, generated by the Display module, to a disk file. Finally, the FindSaveFile routine gets the name of the file in which the data from SpoolPicture is stored.

This application provides the user with great visibility and control over the various sonar system modes of operation available. The output capabilities will be used extensively to generate the test results presented in the next chapter.

4. Test Results

4.0 Introduction

Software testing and its implications with respect to software quality is very important. It represents the ultimate review of specifications, design and coding. Testing is the last activity in the software development method presented in appendix A. The testing approach described was followed as much as possible, yielding maximum error detection at every software development level. This chapter will not present all the tests that were carried out at each level but rather will concentrate on the tests that were performed to validate the requirements presented in section 3.2. These tests were selected as the ones that best show the mapping system capabilities and limitations. Samples of data produced by the scanner, the mapper and the integrator modules are used to present the test results using application generated displays. These results are regrouped in section 4.2 and their descriptions is included in section 4.1.

4.1 Test Results Description

Three tests are presented, each with a different grid size. The first test consists of a series of five views integrated in the local map one after the other using a 25 centimetre grid size. The second test consists of one view using a 40 centimetre grid size. Finally, the third test consists of a series of three views integrated in a local map using a 15 centimetre grid size. Figures 4.1 through 4.14 represent the results of the first test, figure 4.15 and 4.16 show the results of the second test and figures 4.17 through 4.22 correspond to the results of the third test.

For the first test, figures 4.1, 4.3, 4.5, 4.7 and 4.9 represent an overlay of the room layout, the sonar cones and the grid-based representation of the local

map. The light grey squares represent "probably empty" grids while the dark grey squares indicate the "somewhere occupied" grids. Unknown grids are represented by empty squares. The local map grids are initialised to unknown at the beginning of every session. It if helpful to remember from the past chapter that the view size was set to 17 x 17 and the local map size to 33 x 33. The sonar cones range values represented on the maps correspond to the values in figures 4.2, 4.4, 4.6, 4.8 and 4.10. These values correspond to the final range reading produced by the scanner module. The robot position in figure 4.1 is initialised to (0,0) and corresponds to the origin of the coordinate system that is used by the mapping system for the rest of the session. One important thing that should be mentioned about all the figures representing the room is that some of the imprecision is introduced by the room model. Even though carefully built, there are some inaccuracies about the object dimensions and positions. In fact, for the sonar readings where the reflection is generated by a perpendicular object in relation to the beam main axis, the range value obtained is more accurate than the robot position within the room layout which is physically calculated using a measuring tape. Keeping this in mind, the mapping results for the first test are described in the following paragraphs.

The first thing that can be observed about figure 4.1 is the size of the view. Since the local map was initialised to unknown, all that appears are the known grid from the first view acquired from the origin position. The empty squares immediately surrounding the nine squares occupied by the robot are unknown because of the minimum range value chosen at 27.4 centimetres as recommended by the manufacturer. So any grid falling between that range and the robot radius is left unknown.

The inherent inaccuracy of the sonar cone can be observed in the two occupied grids in the lower left corner of the view. These two grids were set to occupied by the sonar #8 cone which was returning the echo from the corner of

the table on the other side of the main lobe axis. The empty probabilities generated by the sonar #9 cone were not large enough to cancel them out. This situation is generally the cause for the presence of occupied grids which appear shifted from the area they represent.

The error percentage chosen (10%) is responsible for the occupied grids in the top left and bottom right corners that do no exactly correspond to the perimeter of the object that returned the echo. As mentioned in chapter 2, 10% was chosen to provide a certain safety margin around the detected obstacles.

Figure 4.2 gives the range reading values of the sonar cones represented in figure 4.1. The discrimination process done by the scanner module is responsible for the fact that there are no out of range readings. The results are quite reliable except for sonar #17 which seems to have been reflected a few times before the echo was received and sonar #10 echo seems to have been reflected by the wall rather than the chair at that location.

The overall results of figure 4.1 are relatively good in terms of the accuracy of the empty areas. These empty areas are in fact what is required by a navigation system to come up with an obstacle-free path to a goal destination. It would however be difficult to carry out any object extraction function from that first view.

In figure 4.3 the robot was moved in the first quadrant of the local map coordinate system (origin determined by the first view). The unknown grids around the robot in figure 4.1 have been filled by the acquisition of the new view. It is important to note that the grids around the robot's new location are not unknown since they were assigned a value in the first view which was not cancelled out by the new values.

The apparent shift of the local map in relation to the room representation is again due to inaccuracies in the room model and also because of the fact that the coordinate system is based on the robot position which does not necessarily register with a grid centre from the previous view thus creating an offset that could be as high as half the grid size. However, an improvement in terms of boundary definition can already be seen as most of the objects on the right side of the local map are represented as occupied with the exception of the grids in the top right corner. This is due to the fact that sonar #21 echo was not reflected by the table at that location but by the wall behind it. Another error is introduced by the range reading from sonar #15 which, again, can be attributed to multiple reflections of the signal. The effects of that error can be observed in the top left corner as most of the third row of grids should have been mapped as occupied.

For figure 4.5, the robot was moved to the second quadrant where it managed to start rectifying the offset occupied grid generated by the first view. The occupied grids close to the robot happen to fall below the minimum range value and therefore were left unchanged since the occupancy probabilities for these grids as per the new view are unknown. The other grids that still show an erroneous occupied status do so because their occupied value was greater than the newly reported empty values. A greater accuracy in the boundary definition for the left side of the map can be observed. However, another faulty range reading (#20) is at the source of all the false grid occupancy status along the top side of the local map. The erroneous readings along that side are probably due to a fabric covered divider represented by the long, thin rectangle. It is believed that the fabric does not constitute a good reflective material.

In figure 4.7, the robot was moved to the third quadrant. From this position the limits on the size of the local map does not permit mapping beyond the straight line of empty grids at the top. Mapping information for that area was lost and would therefore be acquired from scratch if the robot moves back to

cover that area. However for the first time, mapping information on the bottom of the local map is available. The data from the new view provided empty probabilities of sufficient magnitude to correct the occupancy status of the two grids that were set to occupied at the bottom left corner of figure 4.1. A faulty reading from sonar #19 has no bearing on the local map since its effect is beyond the limits of the map.

Finally, in figure 4.9, the robot is moved to the fourth quadrant relatively close to the origin. The empty grid at the top of the map did not change because the sonar readings from sonars #16, #17, #18, and #19 did not provide information beyond the existing boundaries. Empty grids such as these which are not bounded by occupied grids also provide accurate information because the grids beyond them could not be set to occupied, they can be assumed to be occupied. In fact if a smaller error percentage is selected, fewer occupied grids are found but the empty grids will still be bounded just as if the occupied grids were there.

Figures 4.11 and 4.12 show the values stored in the somewhere occupied and the probably empty arrays. If we consider the occupied grids in the top left corner, another one of these grids changed from occupied to empty following integration of the last view. If we look in the arrays for these positions, we observe that the occupied probabilities are only slightly superior to the empty probabilities and with more views they would also toggle to empty values. Incidently, the arrays presented in figures 4.11 and 4.12 are produced by the integrator module after a view is integrated. Figures 4.13 and 4.14 show the somewhere occupied and the probably empty arrays for the last view (fifth robot position). These two arrays are produced by the mapper module for each view acquired.

Figure 4.9 represents the results of integrating five views one after the

other in a local map. A 25 centimetre grid size was chosen because it allowed sufficient coverage to map the complete environment. Figure 4.15 shows results obtained from one view approximately centred within the environment using a 40 centimetre grid size and the same error factor (10%). It is interesting to note that since the grids are bigger, the centre of the grids around the robot did not fall in the unknown area as in figure 4.1. The overall results shown in figures 4.9 and 4.15, in terms of the empty area representation, are good however it would be significantly more difficult to extract object information from figure 4.15 than for figure 4.9. Path planning algorithms would however operate considerably faster on the representation shown in figure 4.15 since fewer grids need to be processed to find a path over a certain distance.

In figures 4.17, 4.19 and 4.21, a 15 centimetre grid size was selected in order to try to accurately map the perimeter of an object such as the table perpendicular to the row of tables in the room model. The problem encountered in this test was that sonar echoes would propagate above the table and be reflected by the object behind it. Because of the rather small coverage of the local map, scans have to be recorded from relatively close to the target object. In figure 4.17 we can see that sonar #3 did not report the echo from the table which caused the occupancy status of some of the grids covering the table to be computed as empty. This is also observed in figure 4.19 with sonars #6 and #7 and in figure 4.21 with sonars #7, #8, #9 and #10. In some of these cases the sonars were reporting the objects on the table rather than the object behind it.

It is obvious from this test that an inaccurate reading impacts more grids than it did when using bigger grid sizes. Furthermore, when using a smaller grid size, the occupied region size, which is relative to the range, is smaller and therefore less occupied grids are found. However, as mentioned before, the pattern of the empty grids provides just as much information about the occupied area as the occupied grids themselves. Those unknown grids beyond empty grids

could not be computed because their range is beyond the one for the occupied grids.

Overall we can see that mapping with a smaller grid is more accurate but also significantly more susceptible to the inherent inaccuracies of the sonar. Another limitation comes from the fact that if we do not want to lose previously acquired information, the robot's movement has to be limited to a rather small area. However, providing good quality returns, a high resolution would certainly make object extraction tasks significantly more precise.

4.2 Test Results

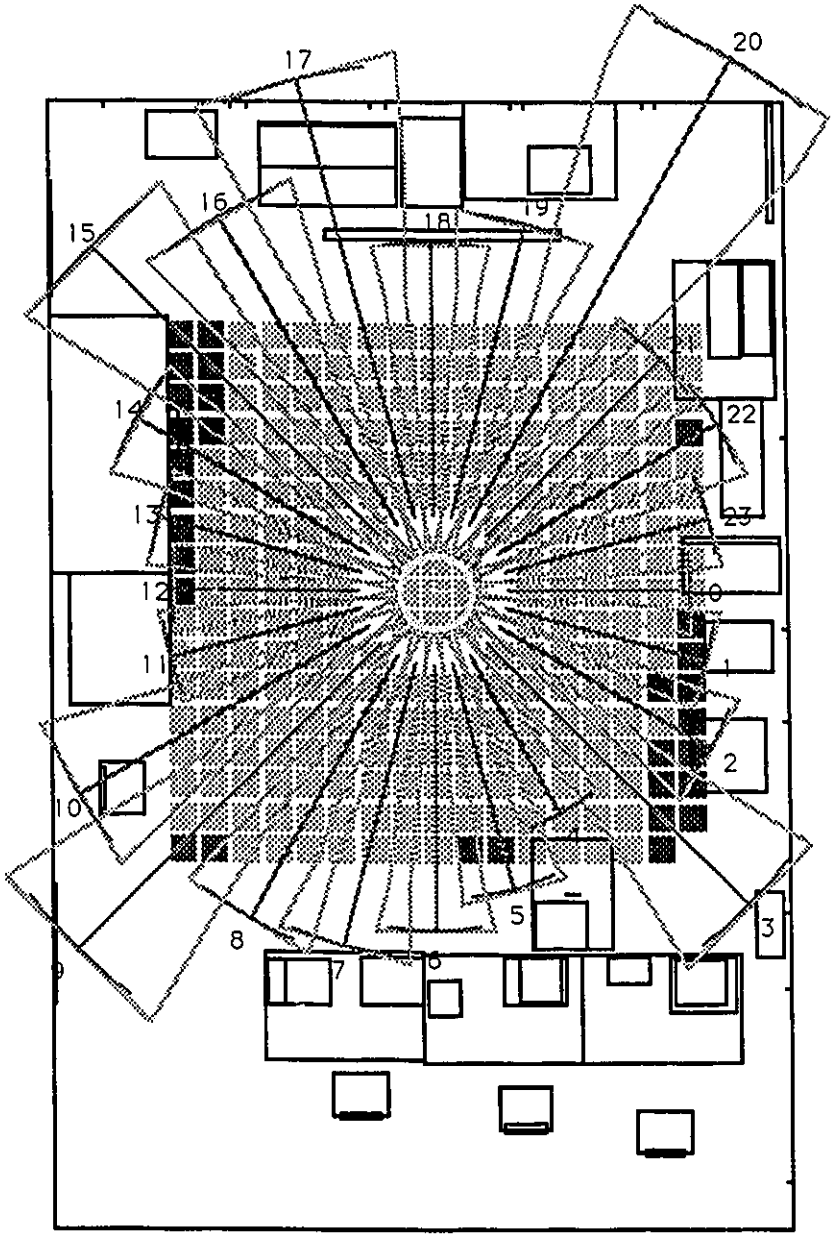


figure 4.1: Local map after the first view (25 cm grid)

Sensor #	Range (CM)
0	174
1	191
2	221
3	317
4	171
5	213
6	236
7	257
8	265
9	367
10	292
11	184
12	175
13	191
14	236
15	346
16	303
17	381
18	240
19	264
20	447
21	225
22	229
23	194

figure 4.2: Range reading values for figure 4.1

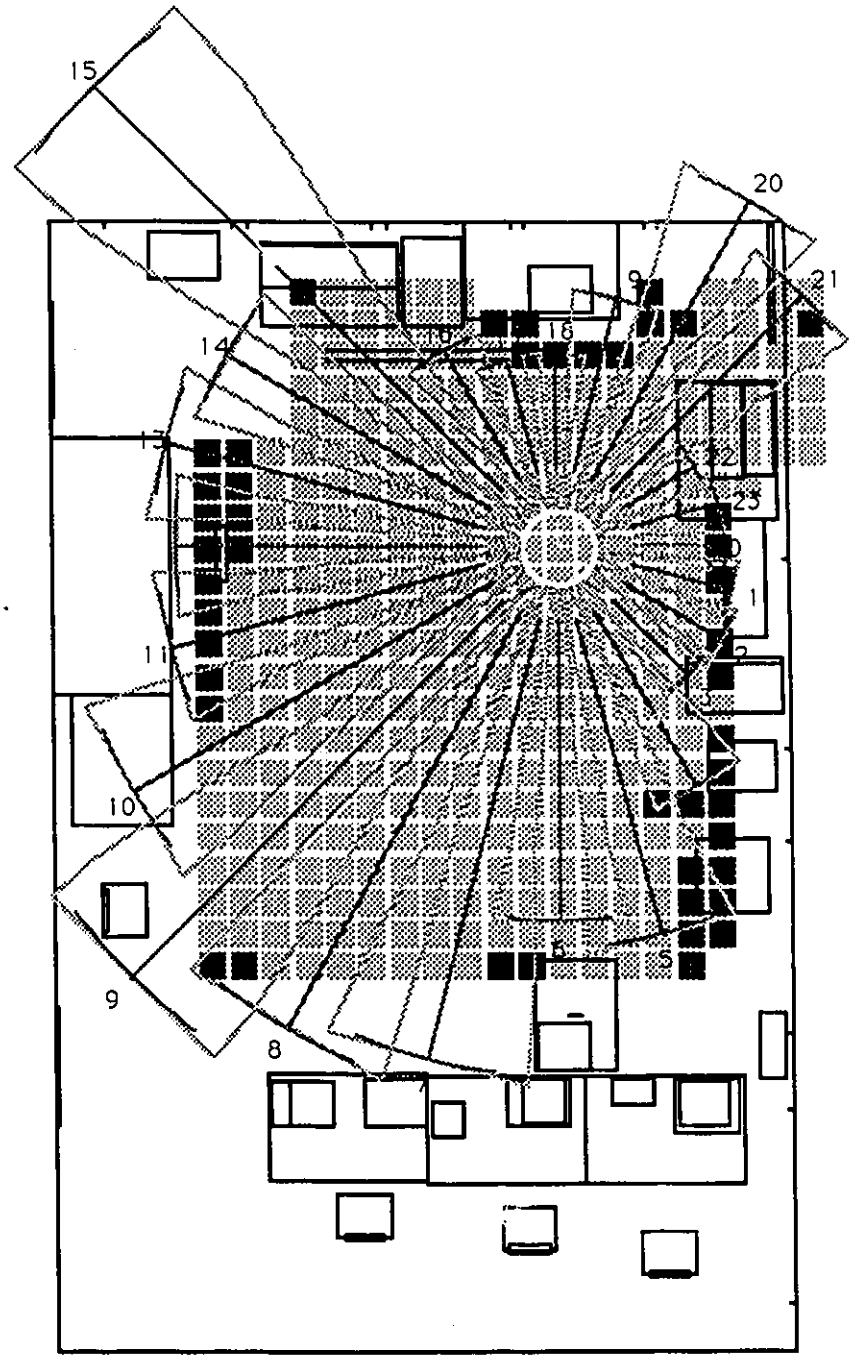


figure 4.3: Local map after the second view (25 cm grid)

Sensor #	Range (CM)
0	93
1	113
2	120
3	115
4	188
5	282
6	263
7	382
8	402
9	444
10	355
11	284
12	268
13	286
14	266
15	477
16	145
17	120
18	120
19	169
20	282
21	247
22	97
23	102

figure 4.4: Range reading values for figure 4.3

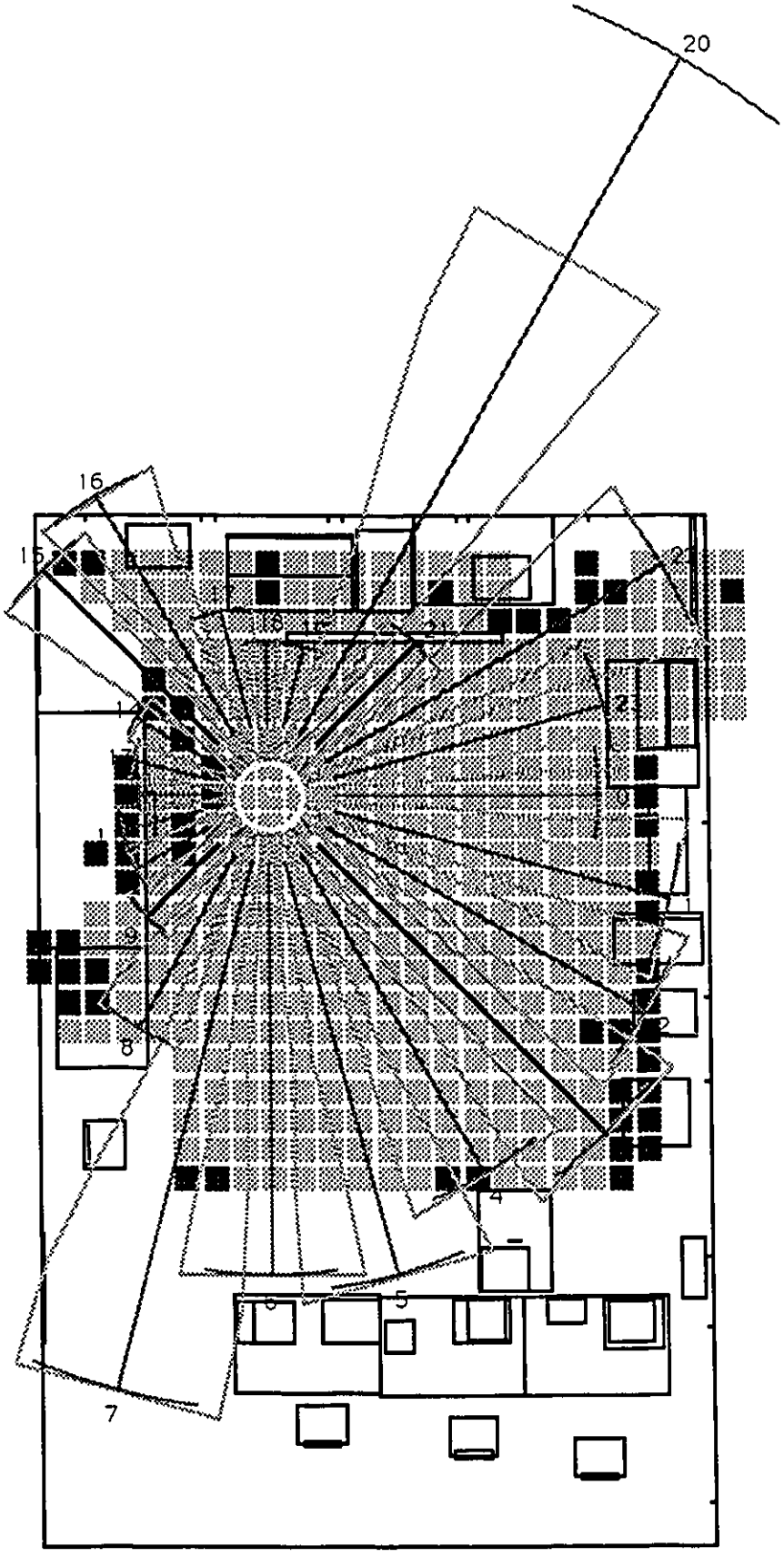


figure 4.5: Local map after the third view (25 cm grid)

Sensor #	Range (CM)
0	258
1	327
2	344
3	379
4	342
5	393
6	378
7	493
8	200
9	119
10	102
11	100
12	81
13	90
14	100
15	240
16	263
17	132
18	100
19	101
20	698
21	152
22	364
23	270

figure 4.6: Range reading values for figure 4.5

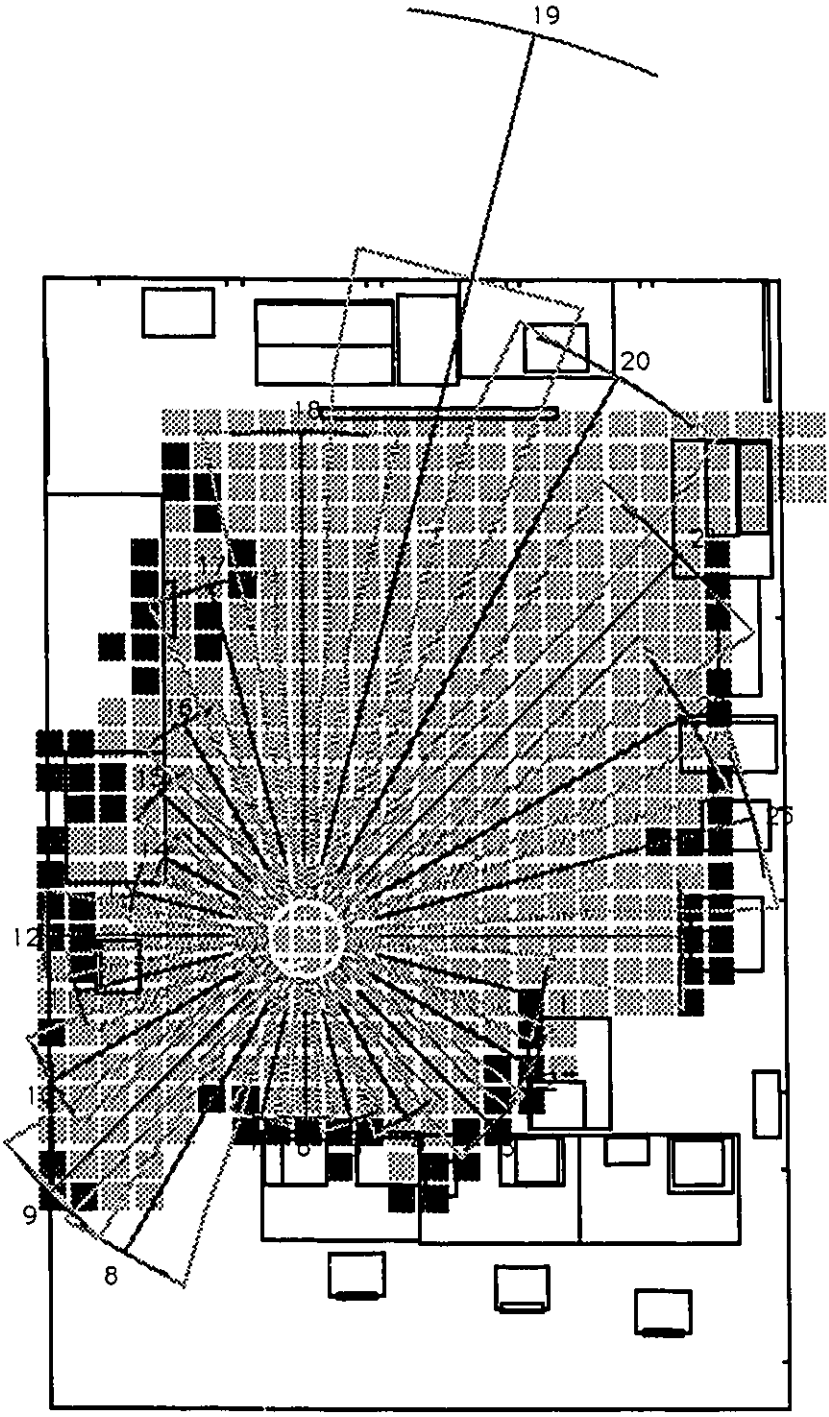


figure 4.7: Local map after the fourth view (25 cm grid)

Sensor #	Range (CM)
0	266
1	163
2	176
3	179
4	134
5	120
6	112
7	116
8	255
9	252
10	198
11	154
12	174
13	106
14	95
15	127
16	159
17	251
18	363
19	698
20	472
21	389
22	317
23	334

figure 4.8: Range reading values for figure 4.7

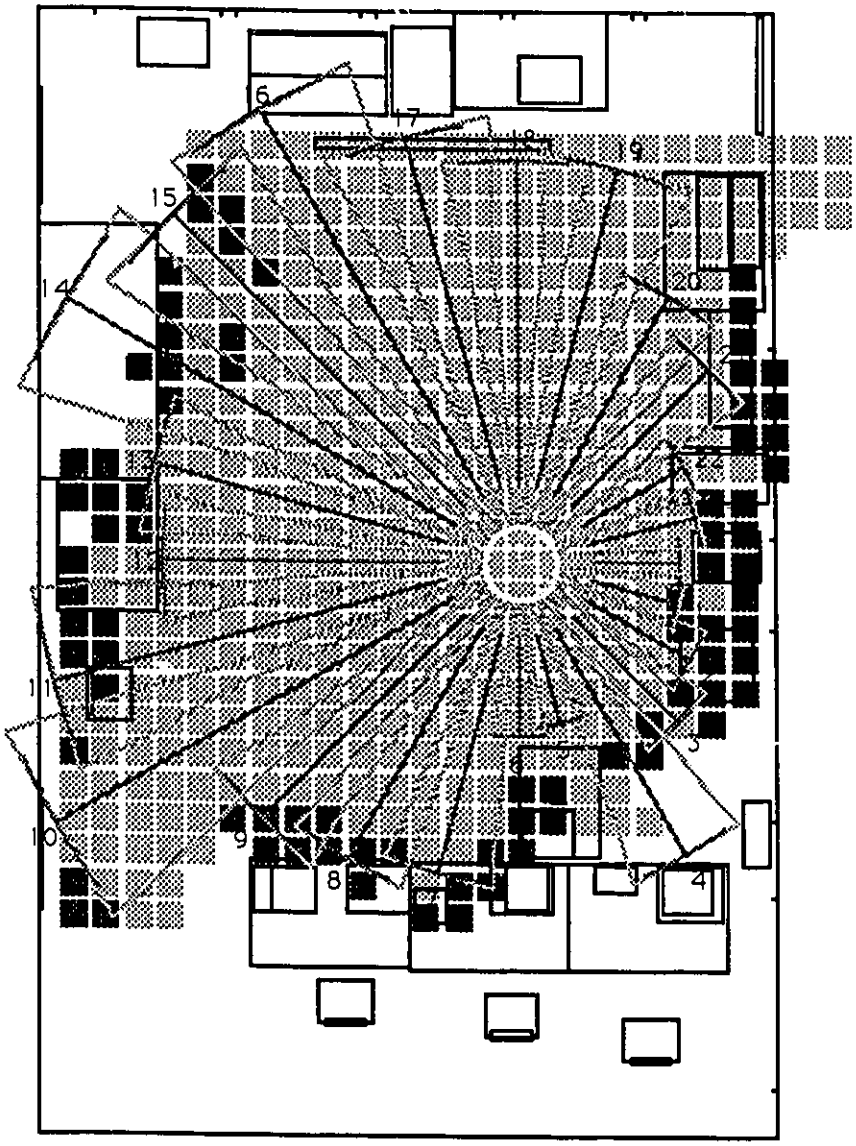


figure 4.9: Local map after the fifth view (25 cm grid)

Sensor #	Range (CM)
0	98
1	101
2	126
3	147
4	234
5	103
6	109
7	221
8	238
9	255
10	386
11	345
12	249
13	264
14	377
15	351
16	374
17	312
18	282
19	287
20	210
21	184
22	121
23	115

figure 4.10: Range reading values for figure 4.9

0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.25	0.24	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.02	0.06	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.56	0.44	0.46	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.67	0.19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.09	0.00	0.21	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.48	0.00	0.42	0.39	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.14	0.29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.53	0.00	0.70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.28	0.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.06	0.54	0.00	0.61	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.56	0.54	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.74	0.00	0.80	0.02	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.81	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.10	0.00	0.61	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.12	0.44	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.11	0.16	0.00	0.00	0.00	0.24	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.31	0.18	0.00	0.55	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.11	0.80	0.01	0.00	0.07	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.64	0.49	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.38	0.56	0.04	0.00	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.83	0.81	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.60	0.11	0.04	0.10	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.35	0.88	0.65	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.90	0.00	0.00	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.64	0.00	0.70	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.55	0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.59	0.41	0.44	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.14	0.47	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.00	0.22	0.46	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.14	0.00	0.00	0.64	0.69	0.45	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.06	0.00	0.00	0.00	0.00	0.00	0.03	0.02	0.38	0.76	0.24	0.00	0.44	0.19	0.34	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.20	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.40	0.00	0.31	0.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.47	0.02	0.00	0.00	0.00	0.00	0.00	0.00	0.18	0.84	0.00	0.14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.87	0.81	0.79	0.55	0.20	0.00	0.00	0.00	0.00	0.69	0.45	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.58	0.89	0.45	0.62	0.11	0.00	0.47	0.85	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.37	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.13	0.00	0.00	0.70	0.79	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.87	0.26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.47	0.36	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Grid Size = 25.00 cm

figure 4.11: Local map occupied probabilities

0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.32	0.63	0.64	0.68	0.56	0.50	0.52	0.56	0.77	0.81	0.76	0.63	0.32	0.20	0.29	0.37	0.51	0.53	0.47	0.37	0.16	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.45	0.69	0.77	0.73	0.65	0.65	0.73	0.81	0.78	0.76	0.65	0.63	0.50	0.55	0.53	0.63	0.57	0.47	0.34	0.20	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.34	0.72	0.75	0.81	0.79	0.79	0.81	0.80	0.80	0.78	0.77	0.80	0.65	0.65	0.71	0.67	0.68	0.46	0.28	0.15	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.46	0.77	0.24	0.67	0.74	0.81	0.80	0.80	0.81	0.81	0.75	0.73	0.39	0.73	0.70	0.70	0.59	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.05	0.57	0.09	0.81	0.81	0.81	0.78	0.81	0.81	0.81	0.78	0.81	0.81	0.73	0.58	0.70	0.61	0.35	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.45	0.75	0.55	0.96	0.97	0.96	0.97	0.98	0.98	0.97	0.96	0.97	0.96	0.92	0.87	0.88	0.74	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.42	0.64	0.54	0.96	0.97	0.98	0.97	0.98	0.98	0.98	0.97	0.98	0.97	0.94	0.87	0.86	0.56	0.12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.04	0.41	0.60	0.95	0.78	0.96	0.97	0.98	0.98	0.98	0.99	0.99	0.97	0.98	0.91	0.94	0.85	0.63	0.25	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.81	0.93	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.97	0.93	0.85	0.65	0.29	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.08	0.39	0.63	0.75	0.96	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	0.93	0.86	0.61	0.21	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.15	0.36	0.66	0.86	0.96	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.98	0.92	0.80	0.55	0.13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.46	0.78	0.83	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	1.00	0.96	0.93	0.64	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.27	0.65	0.79	0.85	0.99	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	0.88	0.56	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.24	0.57	0.72	0.76	0.85	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99	0.98	0.86	0.51	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.08	0.34	0.42	0.33	0.78	1.00	0.94	0.95	0.97	1.00	1.00	0.98	1.00	1.00	1.00	0.88	0.90	0.72	0.27	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.11	0.43	0.62	0.79	0.84	0.91	1.00	1.00	1.00	0.95	0.99	1.00	0.96	0.93	0.91	1.00	0.84	0.56	0.25	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.29	0.58	0.77	0.89	0.83	1.00	1.00	1.00	0.90	1.00	1.00	0.99	1.00	0.99	0.98	0.86	0.53	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.06	0.25	0.52	0.68	0.82	0.81	0.54	1.00	1.00	1.00	0.90	0.99	1.00	0.96	0.91	0.97	0.88	0.76	0.47	0.15	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.05	0.30	0.51	0.68	0.77	0.78	1.00	0.54	0.65	0.72	1.00	0.97	0.90	0.81	0.77	0.93	0.90	0.60	0.41	0.26	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.20	0.44	0.60	0.72	0.70	0.85	0.76	0.86	0.97	0.92	0.92	0.94	0.70	0.47	0.24	0.77	0.79	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.28	0.44	0.58	0.68	0.75	0.53	0.48	0.62	0.77	0.82	0.85	0.78	0.60	0.42	0.03	0.42	0.59	0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.21	0.43	0.53	0.59	0.53	0.00	0.02	0.14	0.32	0.44	0.56	0.60	0.47	0.19	0.00	0.25	0.36	0.24	0.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.12	0.29	0.38	0.41	0.14	0.00	0.00	0.00	0.00	0.00	0.05	0.27	0.16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.12	0.22	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.06	0.03	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Grid Size = 25.00 cm

figure 4.12: Local map empty probabilities

0.00	0.17	0.43	0.60	0.69	0.70	0.64	0.75	0.80	0.69	0.39	0.43	0.41	0.36	0.54	0.65	0.65
0.00	0.00	0.32	0.65	0.75	0.76	0.74	0.80	0.86	0.74	0.57	0.62	0.54	0.61	0.74	0.75	0.65
0.00	0.12	0.31	0.60	0.79	0.81	0.80	0.84	0.92	0.79	0.70	0.70	0.70	0.80	0.83	0.71	0.47
0.01	0.16	0.41	0.59	0.76	0.90	0.83	0.87	0.95	0.83	0.79	0.82	0.84	0.90	0.78	0.55	0.10
0.00	0.04	0.38	0.67	0.80	0.79	0.90	0.96	0.98	0.94	0.87	0.80	0.95	0.83	0.60	0.34	0.15
0.00	0.13	0.43	0.64	0.74	0.93	0.88	0.88	1.00	0.88	0.87	0.98	0.79	0.75	0.60	0.51	0.31
0.00	0.11	0.43	0.68	0.80	0.84	0.99	0.00	0.00	0.00	1.00	0.86	0.84	0.77	0.67	0.52	0.32
0.00	0.28	0.55	0.70	0.88	0.86	0.00	1.00	1.00	1.00	0.00	0.88	0.93	0.81	0.72	0.54	0.26
0.00	0.29	0.56	0.77	0.91	0.98	0.00	1.00	1.00	1.00	0.00	0.99	0.92	0.79	0.61	0.38	0.08
0.40	0.61	0.78	0.86	0.96	0.88	0.00	1.00	1.00	1.00	0.00	0.86	0.87	0.66	0.40	0.12	0.00
0.61	0.72	0.79	0.83	0.90	0.88	1.00	0.00	0.00	0.00	0.99	0.84	0.80	0.68	0.42	0.09	0.00
0.69	0.76	0.83	0.91	0.80	0.95	0.87	0.88	0.99	0.86	0.87	0.92	0.74	0.64	0.43	0.13	0.00
0.72	0.78	0.82	0.80	0.85	0.82	0.86	0.94	0.93	0.87	0.89	0.79	0.76	0.65	0.37	0.04	0.00
0.66	0.71	0.68	0.69	0.77	0.81	0.79	0.82	0.83	0.69	0.73	0.88	0.74	0.53	0.34	0.09	0.00
0.53	0.47	0.48	0.62	0.66	0.69	0.69	0.74	0.69	0.57	0.42	0.77	0.76	0.54	0.20	0.00	0.00
0.23	0.21	0.38	0.48	0.51	0.60	0.56	0.60	0.50	0.37	0.05	0.53	0.69	0.58	0.20	0.00	0.00
0.00	0.05	0.20	0.27	0.37	0.40	0.37	0.39	0.26	0.18	0.00	0.35	0.57	0.52	0.35	0.13	0.00

Grid Size = 25.00 cm

figure 4.13: View empty probabilities

0.60	0.77	0.17	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.92	0.26	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.30
0.29	0.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.71
0.58	0.57	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11
0.96	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.84	0.08	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.71	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.28	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.31	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.61
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.15	0.80
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.96
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.62	0.61
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.31	0.63
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.36	0.00	0.00	0.00	0.94	0.61
0.68	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.53	0.94	0.45	0.00	0.03	0.41	0.53	0.00

Grid Size = 25.00 cm

figure 4.14: view occupied probabilities

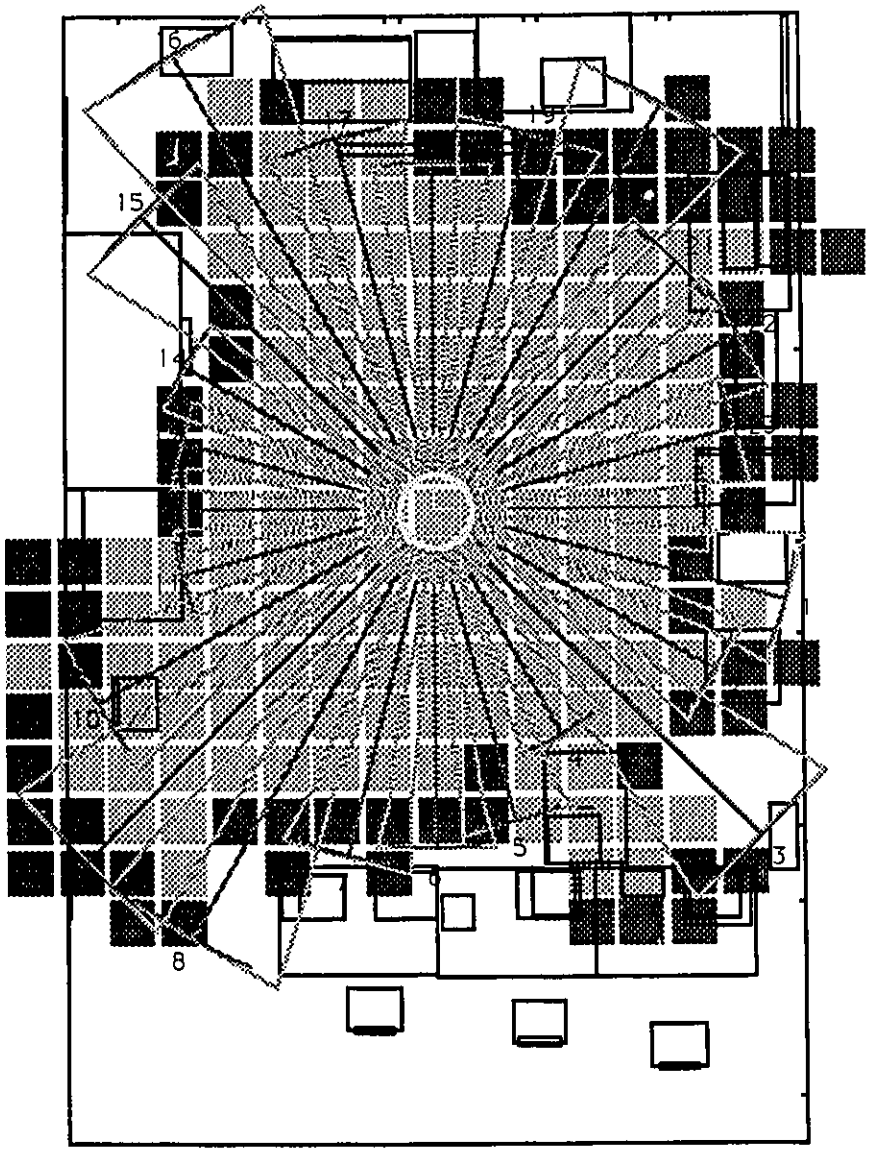


figure 4.15: Local map after one view (40 cm grid)

Sensor #	Range (CM)
0	184
1	255
2	225
3	331
4	174
5	221
6	233
7	249
8	355
9	359
10	275
11	173
12	165
13	174
14	194
15	293
16	378
17	268
18	241
19	273
20	333
21	241
22	244
23	214

figure 4.16: Range reading values for figure 4.15

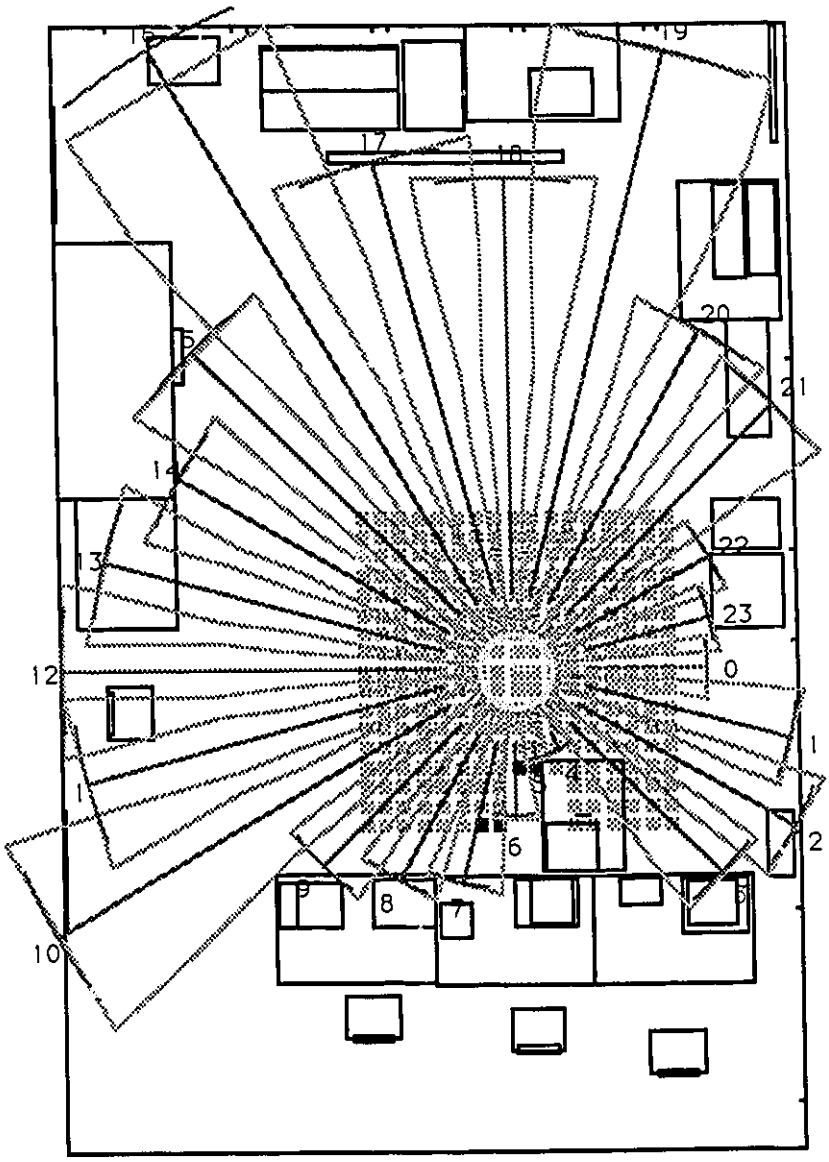


figure 4.17: Local map after the first view (15 cm grid)

Sensor #	Range (CM)
0	122
1	194
2	223
3	199
4	44
5	44
6	87
7	143
8	159
9	186
10	384
11	316
12	323
13	300
14	270
15	319
16	527
17	379
18	353
19	467
20	273
21	261
22	147
23	130

figure 4.18: Range reading values for figure 4.17

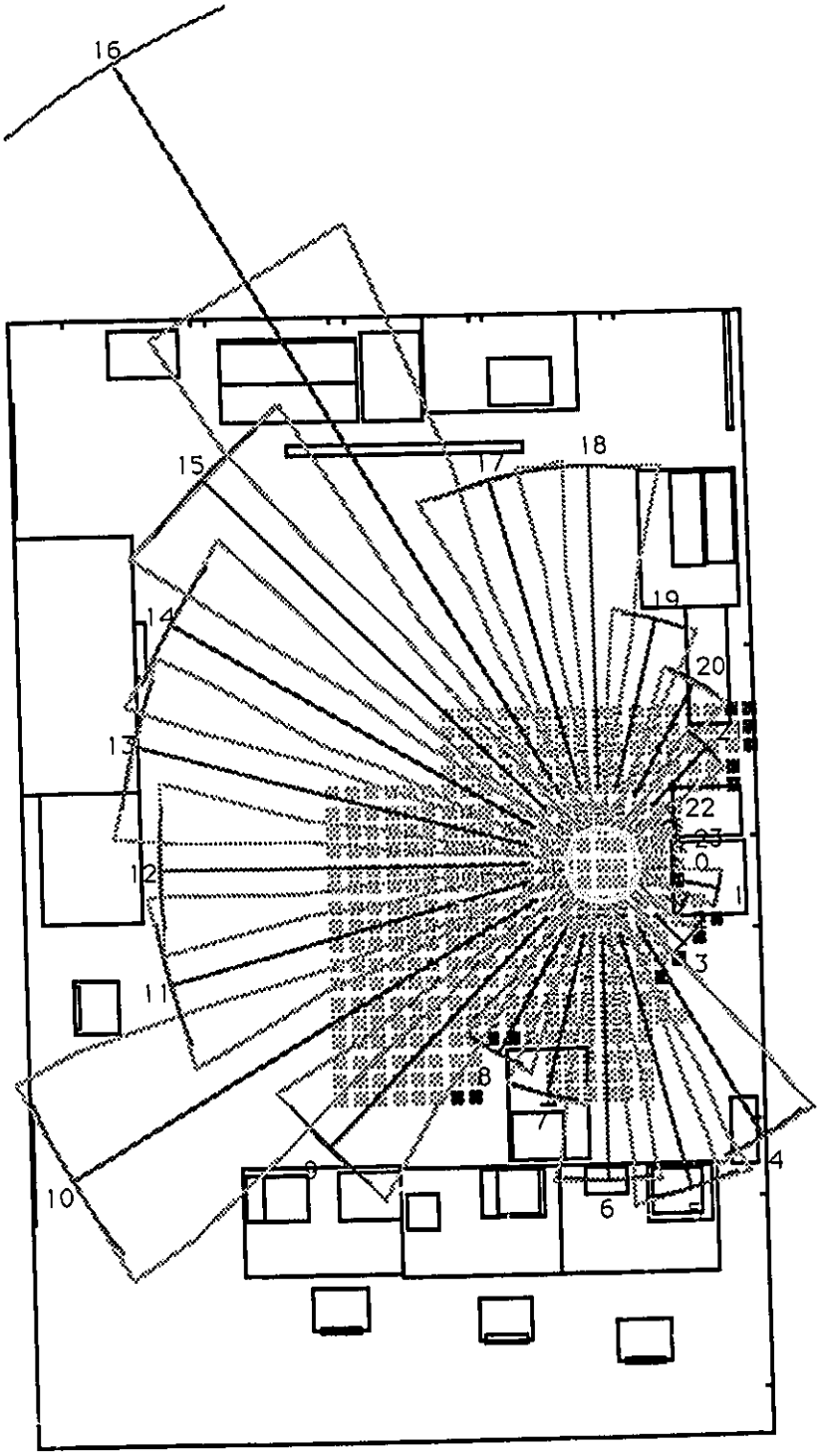


figure 4.19: Local map after the second view (15 cm grid)

Sensor #	Range (CM)
0	32
1	67
2	46
3	63
4	221
5	233
6	218
7	157
8	142
9	278
10	455
11	322
12	315
13	345
14	353
15	398
16	698
17	280
18	280
19	166
20	130
21	94
22	40
23	37

figure 4.20: Range reading values for figure 4.19

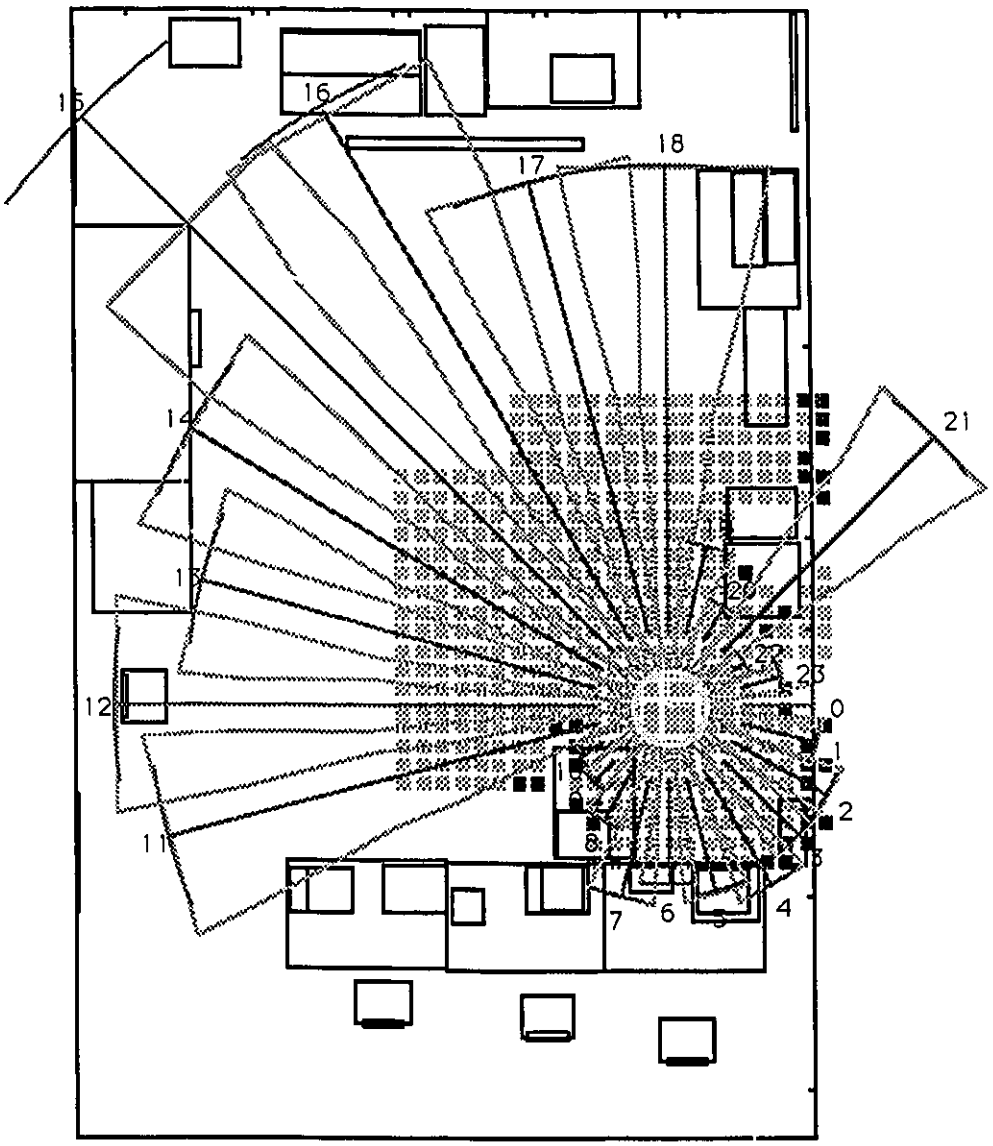


figure 4.21: Local map after the third view (15 cm grid)

Sensor #	Range (CM)
0	83
1	91
2	113
3	115
4	132
5	121
6	110
7	124
8	73
9	53
10	51
11	373
12	403
13	346
14	400
15	614
16	506
17	390
18	391
19	99
20	60
21	266
22	39
23	62

figure 4.22: Range values for figure 4.21

4.3 Performance

As discussed in section 4.1, the results obtained qualify the performance expected from the requirements definition in section 3.2. It is however difficult to quantify the performance of the mapping system in terms of execution speed or efficiency since a significant part of the code is required by the application. In a working configuration, the code would be trimmed down to satisfy the functional requirements by eliminating all the user/application interface functions.

The quality of the results that the mapping system is capable of are very encouraging for implementation of the next processing levels of a navigation system. Tasks such as path planning and object extraction seem to be directly accessible from the results.

Some of the sonar main limitations, such as the width of the beam, even when mapped by a certainty-grid based system still cause erroneous results. However, by integrating the mapping results of other types of sensor, it would be possible to be more selective with the results and therefore increasing their precision. The wide beamwidth is responsible for poor location of the detected object. On the other hand, at close range with the arc of the cone spanning a smaller area, many objects are missed by the echo, such as the table in test #3. A possible solution for resolving that problem may be to lower the ring. This may, however, cause the signal to travel below certain obstacle.

5. Conclusion

This research was part of a navigation system for an autonomous mobile robot. The processing tasks required to implement autonomous robot navigation can be broken down into seven levels which are: Robot Control, Sensor Interpretation, Sensor Integration, Real-World Modelling, Navigation, Global Planning and Control. The aim of this work was to implement the first four of these levels in order to achieve a real world mapping capability using sonars as a source of sensory data. Sonars are attractive because of their simplicity and low cost. A certainty grid based method was used to compensate for sonars inherent inaccuracies. The certainty grid method consists of overlaying an array of imaginary grids over the robot's environment and assigning occupancy probabilities to them. The probabilities are distributed over the sonar cones and are determined experimentally. In order to provide adequate resolution for nearby areas and suffucient coverage for longer range planning, different grid sizes are used implementing a map hierarchy capability.

The real-world mapping system was implemented on a Macintosh II system which interfaced with a sonar system composed of 24 sonars and three controllers. The robot used is a commercially available K2A platform controlled by the Harmony real-time multiprocessing operating system. The end product was integrated within an application environment to assists control and testing but will eventually be adapted to the Harmony system.

Software was developped using a structured method which consists of a requirements analysis phase, a design phase, a coding phase and a testing phase.

The mapping system architecture consists of four modules which are: Sonar Control, Scanner, Mapper and Integrator. The output are arrays of empty and occupied probabilities for a view and a local map which consists of views

from different vantage points integrated together.

The tests that were carried out aimed at validating the requirements defined in the requirements analysis phase. These tests did not try to quantify the results but rather to qualify the mapping system's behaviour with different grid size. From those tests the following conclusions can be drawn:

a. The main goal of the certainty grid method is to adapt to the sonar inherent limitations. The certainty grid does provide that, however certain situations cannot be compensated for as could be observed in chapter four.

b. The certainty grid method manages to compensate for the inaccuracies inherent to sonars, however as the grid size decreases, the impact of faulty readings increases.

c. The occupancy probability distributions used for the sonar cones yields good results however the on going effort to determine them analytically will more than likely improve the quality of the results.

d. Some of the sonar parameters such as the relative error and the minimum range were used as constants but in reality should be modelled as functions of other sonar parameters.

e. Integration of different types of sensor would increase the overall precision of the results.

f. Larger grid size are more resilient to sonar inaccuracies and therefore could be used without the use of other sensor types to carry out function such as global path planning.

g. Smaller grid size provide a better resolution but should be used in conjunction with other sensor types in order to be able to perform functions such as object recognition.

h. The certainty grid method, as implemented, would easily support the integration of different sensor types providing that good occupancy probability distribution models are available.

Based on these conclusions it is possible to assert that the real-world mapping system developed provides adequate resources to support the implementation of the next three levels of a navigation system. At such levels, research provides a very rich environment for the development and tests of advanced concepts in a variety of areas such as robotics , artificial intelligence, sensor interpretation and integration, modelling and planning.

References

[1] Kampmann, P.; Freyberger, F.; Karl, G.; Schmidt, G.: Real-Time Knowledge Acquisition and Control of an Experimental Autonomous Vehicle. Proceedings of International Conference on Intelligent Autonomous System, Amsterdam, The Netherlands, December 1986, pp 294-307

[2] Elfes, A.: Sonar-Based Real-World Mapping and Navigation. IEEE Journal of Robotics and Automation, Vol. RA-3, No. 3, June 1987, pp 249-265

[3] Moravec, H. P.: Sensor Fusion in Certainty Grids for Modile Robots. 1987 Annual Research Review. The Robotics Institutes, Carnegie-Mellon University, pp 33-46

[4] Kambhampati, S.; Davis, L. S.: Multiresolution Path Planning for Mobile Robots. IEEE Journal of Robotics and Automation, Vol. RA-2, No. 3, September 1986, pp 135-145

[5] Miller, D.: A Spatial Representation System for Mobile Robots. IEEE International Conference on Robotics and Automation, St. Louis, Missouri March 1985, pp 122-127

[6] Drumheller, M.: Mobile Robot Localization Using Sonar. A.1. Memo 826, Massachusetts Institute of Technology, January 1985

[7] Samet, H.: An Algorithm for Converting Rasters to Quadtrees. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-3, No. 1, January 1981, pp 93-95

[8] Crowley, J. L.: Navigation for an Intelligent Mobile Robot. IEEE Journal of Robotics and Automation, Vol RA-1, No. 1, March 1985, pp 31-41

[9] Tanimoto S.L.: The Elements of Artificial Intelligence. Computer Science Press, 1987.

[10] Kuan, D. T.; Zamiska, J. C.; Brooks, R. A.: Natural Decomposition of Free Space for Path Planning. IEEE 1985 International Conference on Robotics and Automation, St. Louis, Missouri, March 1985, pp 168-173

[11] Brooks, R. A.: Solving the Find-Path Problem by Good Representation of Free Space. IEEE Transactions on System, Man, and Cybernetics, Vol. SMC-13, No. 3, March/April 1983, pp 190-197

[12] Pressman, R.S.: Software Engineering: A Practitioner's Approach. McGraw Hill, 1987.

[13] Martin, C.F.: User-Centered Requirements Analysis. Prentice Hall, 1988

[14] Yourdon, E.: Modern Structured Analysis. Yourdon Press, 1989

[15] Samet, H.: Distance Transform for Images Represented by Quadtrees. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. PAMI-4, No. 3, May 1982, pp 298-303

[16] Elfes, A.; Moravec, H. P.: High Resolution Maps from Wide Angle Sonar. IEEE Conference on Robotics and Automation, St. Louis, Missouri, March 1985, pp 19-24

[17] Elfe, A.; Matthies, L.: Integration of Sonar and Stereo Range Data Using a Grid-Based Representation. Proceedings of the 1988 IEEE International Conference on Robotics and Automation, Vol 2., Philadelphia, April 1988, pp 727-733

[18] Green, D.; Korba, L.W.; Liscano, R.: Intelligent Platform for Health Care Application. National Research Council of Canada, Ottawa, Canada, pp 22.1-22.7

[19] Vehicle Drive Control (Level II). Presentation given by D. Green, Computing Technology Section, National Research Council of Canada, Ottawa, Canada

[20] Mobile Robot for Health Care. Presentation given by D. Green, Computing Technology Section, National Research Council of Canada, Ottawa, Canada

[21] Korba, L.W.: Command Summary for the Ultrasound Controller, National Research Council of Canada, Ottawa, Canada, June 1988

[22] Denning Mobile Robotics, Inc.: Range Transducer Control Module, Woburn, MA, September 1985

[23] Elfes, A.: A Sonar-Based Mapping and Navigation System. The Robotics Institute, Carnegie-Mellon University, pp 25-30

[24] Lang, S.Y.T.; Wong, A.K.C.: Graph Synthesis for Knowledge Acquisition in Mobile Robots, National Research Council of Canada, Ottawa, Canada

[25] Moravec, H.P.: Three Dimensional Images from Cheap Sonar. The Robotics Institute, Carnegie-Mellon University, December 1985, pp 31-35

[26] Binford, T.O.; Kriegman, D.J.; Triendl, E.: A Mobile Robot: Sensing, Planning and Locomotion. Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, March/April 1987, pp 402-408

[27] Gilbert, E.G.; Johnson, D.W.: Distance Functions and Their Application to Robot Path Planning in the Presence of Obstacles. IEEE Journal of Robotics and Automation, Vol. RA-1, No. 1, March 1985, pp 21-30

[28] Campbell, N.L.; Gex, W.T.: Local Free Space Mapping and Path Guidance. Proceedings of the 1987 IEEE International Conference on Robotics and Automation, Raleigh, North Carolina, March/April 1987, pp 424-429

[29] Davis, L.S.; Le Moigne, J.; Waxman, A.M.: Visual Navigation of Roadways. Proceedings of the International Conference on Intelligent Autonomous Systems, The Netherlands, December 1986, pp 21-30

[30] Gremban, K.D.; Marra, M.; Morgenthaler, D.G.; Turk, M.A.: VITS - A Vision System for Autonomous Land Vehicle Navigation. IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 10, No. 3, May 1988, pp 342-361

[31] Hebert, M.H.; Kanade, T.; Shafer, S.A.; Thorpe, C.: Vision and Navigation for the Carnegio-Mellon Navlab. IEEE Transactions on Pattern Analysis and Macnine Intelligence, Vol. 10, No. 3, May 1988, pp 362-373

[32] Batistoni, A.F.; Ferrari, C.; Di Manzo, M.; Ricci, F.: Using Functional Knowledge in Computer Vision. Department of Commmunication, Computer and System Sciences, University of Genoa, pp 397-404

[33] Komoriya, K.; Tachi, S.; Tanie, K.: A method of autonomous locomotion for mobile robots. Advanced Robotics, Vol 1, No. 1, 1986

[34] Petriu, E.M.: Automated Guided Vehicle with Absolute Encoded Guide-Path. University of Ottawa, Ottawa, Canada

[35] Honey, S.K.; Zavoli, W.B.: A Novel Approach to Automotive Navigation and Map Display. IEEE Transactions on Industrial Electronics, Vol. 1E-34, No. 1, February 1987, pp 40-43

[36] Missiaen, L; Lecluyse, H.; Massart, J.-P.; Bruynooghe, M.: Navigation algorithms for a mobile robot using ultrasonic sensors. Artificial Intelligence and Information Control Systems of Robots, Elsevier Science Publishers, B.V. North Holland, 1987

Appendix A

Software Development

A.0 Introduction

Software engineering refers to the use of engineering principles or techniques used by software developers to build high quality software in a productive manner and by managers to control the software development process. Many books have been written on the subject such as [12], [13] and [14] which more of less agree on a standard life cycle for software development. This life cycle is described in the next section and its main elements are presented in more detail in the following sections.

The aim of this appendix is not to describe a thorough software development method but rather to highlight some of the techniques used for the development of the software for this research.

A.1 Software Development Life Cycle

Figure A.1 shows the various phases forming the software development life cycle.

Essentially all these phases were observed for this research. The return arrow refers to the fact that any errors discovered at any specific phase may have some impact at some or all of the previous phases.

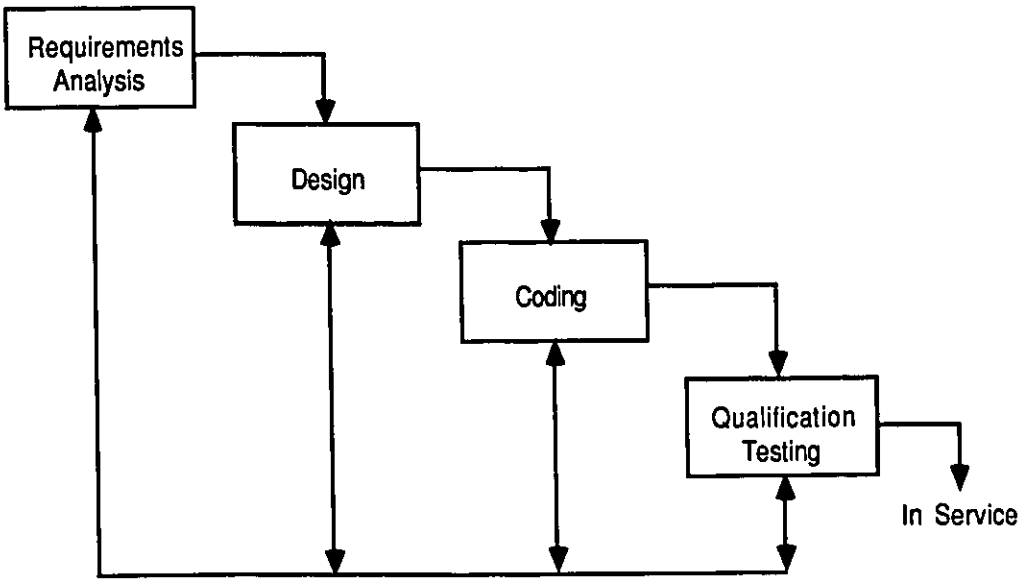


figure A.1 Software development life cycle

It is important to notice that the impact of an undiscovered error will be amplified as the project progresses. This is why it is very important to be as accurate and as thorough as possible in each of the procedures forming the phases and in particular in the early phases since the impact amplification is maximum. A discussion of each phase in detail follows.

A.2 Requirements Analysis

In this phase the requirement analysts work with the sponsors to obtain a conceptual view of the application. It is important to realize that at this stage the sponsor has a concept identification analysis and a feasibility study completed and decided to proceed with the development. Requirement analysis, as the first phase of the development process, enables the software engineer to specify

software functionality and performance, to identify the external interface requirements and define any design constraints that software must meet.

Typically, the requirements analysis phase is broken down in three sub-analyses. The first one is the objective analysis and aims at producing the information flow diagram, determining the critical success factors and defining project parameters such as purpose, objectives and features. The second sub-analysis is the functional analysis in which data flow diagrams are produced and outputs are defined. Finally, the data definition sub-analysis looks at producing data structure diagrams along with a high level data dictionary.

The product of this phase is a document called software requirement specification (SRS) which shall include the results of each of the sub-analyses. The SRS is a cognitive model as opposed to a design or implementation model. This characteristic of the SRS is essential at the qualification testing phase in order to validate the end product. Validation verifies that the right product was built correctly and deals with lower level documents such as the design documents.

Errors at this stage can have a maximum impact on the performance of the end product as explained in the previous section. For this reason, requirements reviews and walkthroughs are held with the sponsor in order to maximise error detection and to assure that both parties have the same understanding of the system.

A.3 Design

The design phase is normally broken down into two parts - the preliminary design phase and the detailed design phase. The goal of the design

phase is to produce simple, easy to maintain and reliable software. One of the key activities to take place during preliminary design consists of partitioning the requirements in the SRS into a hierarchical modular software architecture. The advantages of a good modular design are as follows:

1. improves maintainability of software
2. provides a convenient basis for testing
3. provides a convenient basis for scheduling, estimating and progress reporting
4. helps identify programming task breakdown.

During the detail design phase, the data and functional structures created during preliminary design are progressively refined and documented. The document produced during the design phase is called the design specification and includes results of both the preliminary and the detailed design phases. It describes the functional characteristics of the modules along with the interface between them and the data elements involved.

A.4 Coding

Coding is the process that transforms design into a programming language. The translation of detail design specifications into code is highly dependent on how well the design phase was carried out and documented. Indeed, experts agree that for most projects the coding effort should not account for more than 20% of the total development effort. In this phase, the programming language is chosen. It should support the functional and data structures identified in the design specification. The only documents produced during this phase are the software source listings.

A.5 Qualification Testing

Software testing is a critical element of software development as it represents the ultimate review of specification, design and coding. Testing is the process of executing various parts of the software with the intent of finding errors. This activity is carried out at every level of the hierarchical software structure and results are gathered and form parts of the final test report. The tests selected should also be structured in a way such that low level tests verify the detail design specification while high level tests verify the requirement specification thus validating the end product.

As briefly mentioned in section A.2, verification and validation (V&V) are part of what is referred to as software quality assurance and aim at making sure that the product was built correctly and that the right product was built. V&V is accomplished with a series of four steps implemented sequentially. These steps are:

1. Unit testing
2. Integration testing
3. Validation testing
4. System testing.

System testing refers to the process of testing the end product when combined with other system elements such as hardware or people. Unit testing deals with the verification of individual modules while integration testing verifies that the modules perform together as designed. The combination of unit testing and integration testing implement the verification part of V&V.

A.6 Conclusion

As mentioned before, the software development method outlined in this appendix does not constitute an ultimate tool but rather a guideline used throughout this research provide visibility regarding the development status. Furthermore, it allowed NRC to keep accurate track of progress thus enabling them to commend on development decisions made through the life cycle.

Appendix B

Sonar Control Module

```
/ *
DoCommands.c
- - - - - - - - - - - -

Send commands and data words to the controller.
        Commands:
                0 - Stop auto scan.
                1 - Start auto scan.
                2 - Perform a single scan.
                3 - Set the sensors to excite on a scan.
                4 - Set the maximum depth setting.
                5 - Set the excitation delay.
                6 - Set the mode and gain.
                7 - Set the firing pattern used on a full scan.
                8 - Find the closest object.
                9 - Resend data from last full scan.
               10 - Excite a single transducer.
               11 - Warning Beep.
               12 - Get the operating parameters.
               13 - Flush the data transmit queue.
               14 - Reset the operating parameters of the controller to the default
                    values.
               15 - Do a hard reset of controller.

Created 29 August, 1988 by Sherman Lang.
* /

#include "Sonar.i"
#include "Sonar.e"
#include "SonarSerial.e"
#include "SonarRover.e"
#include "SonarController.e"

void DoCommand(CommandNumber)
        int CommandNumber;
        {
        unsigned int
                CommandWord_1,    /* 1st command word to be sent */
                CommandWord_2;    /* 2nd command word to be sent */

        CommandWord = Command[CommandNumber];
```

```
switch (CommandNumber)
        {
        case 0:
                {
                unsigned int
                        Number, Count;        /* sensor number, sensor count */

                int
                        AckFound;     /* flag to indicate acknowledge found */

                long int
                        FinalTicks,   /* number of ticks actually delayed */
                        BytesAvail;   /* bytes available in serial buffer */

                AckFound = FALSE;
                CommandWord_1 = Command[0];
                WriteSerial(CommandWord_1);
                Delay(60L, FinalTicks);              /* delay to allow transfer */
                SerGetBuf(SerialIn,&BytesAvail);  /* see how many bytes
                                                        available */
                if ( (int)BytesAvail >= 2 )  /* must be at least 2 bytes */
                        {
                        do
                                {
        /* Read 2 bytes at a time until ACK found */
                                ReadData(&Number, &Count);
                                if ( (Number == 24) && (Count == 0) )
                                        AckFound = TRUE;
                                SerGetBuf(SerialIn, &BytesAvail);
                                }
                        while ( (int)BytesAvail != 0);
                        }
                Delay( 60L, FinalTicks);     /* delay to allow transfer of final
                                                data */
                ClearBuffer();               /* flush serial input buffers */
                if (!AckFound) DoError(6);
                }
                break;
        case 1:
        case 2:
        case 8:
        case 9:
        case 12:
        case 13:
        case 14:
```

```
case 15:
        WriteSerial(Command[CommandNumber]);
        break;
case 3:
        {
        int     i;      /* counter */

        CommandWord_1 = Command[3];
        CommandWord_2 = 0;

        /* set the bits */
        for (i=0; i<24; i++)
                {
                if (i<8)
                        { if (ExcitationMask[i] == 1) CommandWord_1 =
                          CommandWord_1 | Bit[i]; }
                else
                        { if (ExcitationMask[i] == 1) CommandWord_2 =
                          CommandWord_2 | Bit[i-8]; }
                }

        /* send the command words */
        WriteSerial(CommandWord_1);
        WriteSerial(CommandWord_2);
        }
        break;
case 4:
        WriteSerial(MaxDepth + Command[4]);
        break;
case 5:
        WriteSerial(FiringDelay + Command[5]);
        break;
case 6:
        CommandWord_1 = Command[6];
        if ((int)RangeMode) CommandWord_1 =
                CommandWord_1 | Bit[2];
        CommandWord_1 += GainProfile;
        WriteSerial(CommandWord_1);
        break;
case 7:
        {
        int     i;      /* counter */

        CommandWord_1 = Command[7];
        CommandWord_2 = 0;
        for (i=0; i<24; i++)
```

```
                        {
                        if  (i<8)
                                { if (FiringPattern[i]) CommandWord_1 =
                                        CommandWord_1 | Bit[i]; }
                        else
                                { if (FiringPattern[i]) CommandWord_2 =
                                        CommandWord_2 | Bit[i-8]; }
                        }
                    /* send the command words */
                    WriteSerial(CommandWord_1);
                    WriteSerial(CommandWord_2);
                    }
                    break;
            case 10:
                    WriteSerial(SingleSensor | Command[10]);
                    break;
            case 11:
                    {
                    unsigned int   Tone[10];
                    int                         i;

                    CommandWord_1 = Command[11] | 0x000A;
                    Tone[0] = 0x3020;
                    Tone[1] = 0x2050;
                    Tone[2] = 0x3020;
                    Tone[3] = 0x2050;
                    Tone[4] = 0x3020;
                    Tone[5] = 0x2050;
                    Tone[6] = 0x3020;
                    Tone[7] = 0x2050;
                    Tone[8] = 0x3020;
                    Tone[9] = 0x2050;
                    WriteSerial(CommandWord_1);
                    for  (i=0;i<10;i++)  WriteSerial(Tone[i]);
                    }
                    break;
            }
    }
/*
ControllerInterface.c
- - - - - - - - - - - - - - - - - - - - - -

Low and mid-level routines to interface to controller.
Created 29 August, 1988 by Sherman Lang.
*/
#include "Sonar.i"
```

```
#include "SonarSerial.e"
#include "SonarController.e"
#include "Sonar.e"

/* Clear the serial buffer. No return value. */
void ClearBuffer()
        {
        KillIO(SerialIn);
        }

/ *
Read a 16 bit word from the serial port.
If successful, returns TRUE, if not returns FALSE.
* /
int ReadSerial(DataWord)
        unsigned int
                *DataWord;

        {
        OSErr
                ReadErr,
                ReadErr1,
                ReadErr2;
        unsigned int
                DataHigh, DataLow;
        long
                Count,
                Count1,
                Count2,
                FinalTicks;
        unsigned char
                Word[3],
                Word1,
                Word2;
        int
                i;
        long int
                BytesAvail;             /* 8 bit bytes available in serial buffer */
        WindowPtr
                OldPort;
        Str255
                NumString;
        Rect
                EraseBox;

        SetRect(&EraseBox,90,90,400,200);
```

```
        Count1 = 1;
        Count2 = 1;

        ReadErr1 = FSRead(SerialIn, &Count1, &Word1);

        ReadErr2 = FSRead(SerialIn, &Count2, &Word2);

        if ( ( (ReadErr1 == noErr) && ( (int)Count1 == 1 ) )
                && ( (ReadErr2 == noErr) && ( (int)Count2 == 1 ) ) )
                {
                DataHigh = ((unsigned int)Word2) << 8;
                DataLow = (unsigned int)Word1;
                *DataWord = DataHigh + DataLow;
                return(TRUE);
                }
        else
                {
                DoError(4);
                *DataWord = 0;
                return(FALSE);
                }
        }

/ *
 Write a 16 bit word to the serial port.
Returns TRUE if successful, FALSE otherwise.
* /
int  WriteSerial(CommandWord)
        unsigned int
                CommandWord;

        {
        OSErr
                WriteErr;
        unsigned int
                CommandHigh, CommandLow;
        long
                Count;
        unsigned char
                Word[2];

        CommandHigh = (CommandWord & 65280) >> 8;
        CommandLow = (CommandWord & 255);

        Word[0] = (unsigned char)CommandLow;
        Word[1] = (unsigned char)CommandHigh;
```

```
        Count = 2;
        WriteErr = FSWrite(SerialOut, &Count, Word);

        if (WriteErr == noErr)
                {
                if (Count == 2)                 /* Command word sucessfully sent */
                        {
                        return(TRUE);
                        }
                else                            /* Unsucessful sending 2 bytes */
                        {
                        return(FALSE);
                        }
                }
        else
                {
                DoError(0);
                return(FALSE);
                }
        }

/ *
Read a 16 bit word from the serial port.
The word is split into two words : bits 11 to 15 contains the
transducer number, and the bits 0 to 10 contains the
depth count.  If data is successfully read returns TRUE,
else returns FALSE.
* /
int ReadData(Number,Count)
        unsigned int
                *Number, *Count;

        {
        unsigned int
                Data16,                         /* 16 bit data word */
                High5,                          /* high 5 bits */
                Low11;                          /* low 11 bits */

        long int
                BytesAvail;             /* 8 bit bytes available in serial buffer */

        EventRecord
                InterruptEvent;         /* interrupt event to allow us to break out of
                                           do-while loop */
```

```
        /* keep looking until at least 2 bytes available in buffer */
        do
                {
                SerGetBuf(SerialIn,&BytesAvail);
                if (GetNextEvent(keyDownMask,&InterruptEvent)) break;
                }
        while(BytesAvail < 2);

        if ( (BytesAvail > 1) && (ReadSerial(&Data16)) )
                {
                /* successful in reading */
                Low11 = Data16 & 0x7FF;          /* 2047 */
                High5 = Data16 & 0xF800; /* 63488 */
                *Number = High5 >> 11;
                *Count = Low11;
                if ((int)(*Number) == 31) DoError(7);
                return(TRUE);
                }
        else
                {
                /* unsuccessful in reading */
                *Number = 0;
                *Count = 0;
                return(FALSE);
                }
        }

/*
Receive the acknowledge of transmitted command.
Returns TRUE if ack found, FALSE otherwise. Error number is returned
in Command if FALSE returned.
*/
int ReceiveAck(Command)
        unsigned int *Command;
        {
        unsigned int
                Ack, AckData;
        long int
                BytesAvail;
        int
                i;
        WindowPtr
                OldPort;
        Str255
                NumString;
```

```
long int
        FinalTicks;

/*  Delay(20L,&FinalTicks);  */     /* wait for acknowledge to appear */

BytesAvail = 0L;

while ( (int)BytesAvail < 2 )
        {
        SerGetBuf(SerialIn,&BytesAvail);  /* See how many bytes available */
        }

if ( ((int)BytesAvail >= 2) )                   /* see if there are enough bytes */
        {
        while (!ReadData(&Ack,&AckData));       /* keep trying until
                                                   successful */

        if ((Ack == 24) && (AckData == *Command/256))  /* look for ack */
                {
                return(TRUE);
                }
        else if (Ack == 31)          /* look for error, return error number */
                {
                DoError(1);
                *Command = Ack;
                return(FALSE);
                }
        else                                    /* look for garbage, return 0 */
                {
                DoError(2);
                *Command = 32;
                ClearBuffer();
                return(FALSE);
                }
        }
 else   /* can't find data at serial port */
        {
        DoError(3);
        ClearBuffer();
        }
}

/* Read a single sensor value. Return TRUE if data read, FALSE otherwise. */
int ReadSingle()
        {
        EventRecord
```

```
        interruptEvent;

    unsigned int
        SensorNumber, SensorCount;

    if ( !GetNextEvent(mDownMask,&interruptEvent) &&
        ReadData(&SensorNumber, &SensorCount) )
        {
        NewScan[0][0] = SensorNumber;
        NewScan[0][1] = SensorCount;
        NewScan[1][0] = 24;
        NewScan[1][1] = 0;
        return(TRUE);
        }
    else
        return(FALSE);
    }

/* Reads the data received on a full scan.  No return value. */
void ReadScan()
    {
    EventRecord
        ScanEvent;    /* allow break out of while loop if mouse pressed */

    unsigned int
        SensorNumber, SensorCount;      /* sensor number and count read */
    int
        Count;                          /* count of data values read */
    long
        ElapsedTime;                    /*  time  */
    OsErr
        DataWriteErr;                   /* write error for data */
    long int
        NumBytes;                       /* number of bytes to write */
    int
        Data;

    Count = 0;

    /* if (SaveData)
        {
        NumBytes = 2L;
        FSWrite(DataRefNum, &NumBytes, &RangeMode);
        NumBytes = 2L;
        FSWrite(DataRefNum, &NumBytes, &GainProfile);
        NumBytes = 2L;
```

```
        FSWrite(DataRefNum, &NumBytes, &FiringDelay);
        NumBytes = 2L;
        FSWrite(DataRefNum, &NumBytes, &MaxDepth);
        } */

while ( !GetNextEvent(mDownMask,&ScanEvent) &&
        ReadData(&SensorNumber,&SensorCount) )
        {
        Count++;

        if (Count > 24)
                {
                if (SaveData)
                        {
                        SensorNumber = 24;
                        SensorCount = 0;
                        NumBytes = 2L;
                        FSWrite(DataRefNum, &NumBytes, &SensorNumber);
                        NumBytes = 2L;
                        FSWrite(DataRefNum, &NumBytes, &SensorCount);
                        }
                break;
                }

        NewScan[Count-1][0] = SensorNumber;
        NewScan[Count-1][1] = SensorCount;

        if (SaveData)
                {
                NumBytes = 2L;
                FSWrite(DataRefNum, &NumBytes, &SensorNumber);
                NumBytes = 2L;
                FSWrite(DataRefNum, &NumBytes, &SensorCount);
                }

        if ((int)SensorNumber == 24)
                {
                break;
                }
         else if ((int)SensorNumber > 24)
                {
                Delay(60L, ElapsedTime);
                ClearBuffer();
                break;
                }
        }
```

```
}

/* Get the status of the serial controller. */
void GetCStatus()
{
int
        i;
long int
        BytesAvail;

unsigned int
        StatusWord1, StatusWord2, StatusWord3, StatusWord4;
do
        {
        SerGetBuf(SerialIn,&BytesAvail);
        }
while( BytesAvail < 8);

ReadSerial(&StatusWord1);
ReadSerial(&StatusWord2);
ReadSerial(&StatusWord3);
ReadSerial(&StatusWord4);

/* Firing delay, gain profile, maximum depth */
FiringDelay = StatusWord1 & 255;
GainProfile = (StatusWord1 & 768) >> 8;
RangeMode = (StatusWord1 & 1024) >> 10;
MaxDepth = (StatusWord1 & 14336) >> 11;

/* Excitation mask */
for (i=0; i<16; i++)
        {
        if (Bit[i] & StatusWord2)
                ExcitationMask[i] = TRUE;
        else
                ExcitationMask[i] = FALSE;
        }
for (i=0; i<8; i++)
        {
        if (Bit[i]  & StatusWord3)
                ExcitationMask[i+16] = TRUE;
        else
                ExcitationMask[i+16] = FALSE;
        }

/* Firing pattern */
```

```
for  (i=8;  i<16;  i++)
        {
        if  (Bit[i]  &  StatusWord3)
                FiringPattern[i-8]  =  TRUE;
        else
                FiringPattern[i-8]  =  FALSE;
        }
for  (i=0;  i<16;  i++)
        {
        if  (Bit[i]  &  StatusWord4)
                FiringPattern[i+8]  =  TRUE;
        else
                FiringPattern[i+8]  =  FALSE;
        }
}
```

Appendix C

Scanner Module Code

```
/ *

Scanner.c

        This module is responsible to set up the sonar controller parameters and trigger
the scans in both far field and near field modes. In each of the two modes, three readings
are taken. The readings are then formatted and averaged. The two arrays obtained (one
for each mode) are then processed to select the most accurate reading. This module
invokes the Sonar Controller module to communicate with the sonar controller system.

* /

#include "SonarControlScanner.e"        /* Global variables definition of the
                                           variables shared by this module and the
                                           Sonar Controller module */
#include "ScannerMapper.e"              /* Global variables definition of the
                                           variables shared by this module and the
                                           Mapper module */
#include <stdio.h>                      /* Standard input/output C library */

unsigned int
        TempScan[24][2],        /* Contains the sonar position number and average
                                   range readings after every three scan in near
                                   or far mode */
        ncount[24];             /* Contains the number of not out of range readings
                                   for each sonars in near or far mode */

FILE
        *fopen(),       /* C file opening function */
        *Map;           /* File used to return the scan results */

Scan()                  /* Scanner module main routine */

        {
                float
                        Convert,        /* sonar depth count to cm conversion factor */
                        FarScan[24][2],     /* Loaded with TempScan after a scan with
                                               far mode settings */
                        NearScan[24][2];    /* Loaded with TempScan after a scan with
                                               near mode settings */

                unsigned int
                        NearCount[24],      /* Loaded with ncount after a scan with near
                                               mode settings */
```

```
        FarCount[24];       /* Loaded with ncount after a scan with near
                               mode settings */

    int
        i;                  /* Loop index variable */

    Convert = 0.7143;       /* conversion factor assignment */

    GainProfile = 1;        /* Small object rejection gain profile setting */
    RangeMode = 0;          /* Near mode setting */
    MaxDepth = 7;           /* Maximum Depth setting */
    FiringDelay = 127;      /* Firing delay setting */

/* Call up the Sonar Controller command execution routine and wait for the handshake
* /

    DoCommand(4);
    if (!ReceiveAck(&CommandWord)) ClearBuffer();
    DoCommand(5);
    if (!ReceiveAck(&CommandWord)) ClearBuffer();
    DoCommand(6);
    if (!ReceiveAck(&CommandWord)) ClearBuffer();

/* Trigger a single scan  of the ring with current settings */

    TrigScan();

/* Transfer TempScan and ncount into NearScan and Nearcount */

    for (i = 0; i < 24; i++)
        {
        NearScan[i][0] = TempScan[i][0] * 15;
        NearScan[i][1] = TempScan[i][1] * Convert;
        NearCount[i] = ncount[i];
        }

    RangeMode = 1;              /* Change mode setting to far mode */

/* Call up the Sonar Controller command execution routine and wait for the handshake*/

    DoCommand(6);
    if (!ReceiveAck(&CommandWord)) ClearBuffer();

/* Trigger a single scan  of the ring with current settings */

    TrigScan();

/* Transfer TempScan and ncount into NearScan and Nearcount */

    for (i = 0; i < 24; i++)
```

```
                {
                FarScan[i][0] = TempScan[i][0] * 15;
                FarScan[i][1] = TempScan[i][1] * Convert;
                FarCount[i] = ncount[i];
                }

/* Selection of the best reading based on the number of not out of range readings or the
range */

        for (i = 0; i < 24; i++)
                {
                MapScan[i][0] = NearScan[i][0];
                if (NearCount[i] < FarCount[i])
                        MapScan[i][1] = FarScan[i][1];
                else if (NearCount[i] > FarCount[i])
                        MapScan[i][1] = NearScan[i][1];
                else if (NearScan[i][1] < 1)
                        MapScan[i][1] = 702;
                else if (NearScan[i][1] > 400)
                        MapScan[i][1] = FarScan[i][1];
                else
                        MapScan[i][1] = NearScan[i][1];
                }

/* Dump the final scan result into a file */

        Map = fopen("ScanResult","w");
        fprintf(Map, "Sonar #    Range (cm)\n\n");
        for (i = 0; i < 24; i++)
                {
                fprintf(Map, "%4.0f        %7.2f \n", MapScan[i][0],
                        MapScan[i][1]);
                NewScan[23 - i][0] = (unsigned int)(MapScan[i][0]/15);
                NewScan[23 - i][1] = (unsigned int)(MapScan[i][1]/0.7143);
                }
        fclose("ScanResult");

    UpdateGraph();
    UpdateText();
    }


TrigScan()          /* Scanner module scan trigger routine */

    {
        int
            j,          /* Loop variable index */
            k;          /* Loop variable index */

/* Initilize TempScan and ncount for a new set of readings */
```

```
            for (k = 0; k < 24; k++)
                  {
                  TempScan[k][1] = 0;
                  ncount[k] = 0;
                  }

/* Loop controlling the firing of the scans */

            for (j = 0; j < 3; j++)
                  {

/* Call up the Sonar Controller command execution routine and wait for the handshake*/

                  DoCommand(2);
                  if (ReceiveAck(&CommandWord))
                        {
                        ReadScan();
                        UpdateGraph();
                        UpdateText();
                        }
                  else
                        {
                        ClearBuffer();
                        }

/* Transfer the data from the Sonar Controller module into TempScan and increment
ncount if range reading not out of range */

                  for (k = 0; k < 24; k++)
                        {
                        TempScan[k][0] = NewScan[23 - k][0];
                        if (NewScan[23 - k][1] != 1)
                              {
                              TempScan[k][1] += NewScan[23 - k][1];
                              ncount[k] += 1;
                              }
                        }
                  }

/* Compute the average reading */

            for (k = 0; k < 24; k++)
                  if (ncount[k] == 0)
                        TempScan[k][1] = 1;
                  else
                        TempScan[k][1] /= ncount[k];
      }
```

Appendix D

Mapper Module Code

```
/ *

Mapper.c

        This module is responsible to generate a certainty grid-based view of the robot's
environment for a particular robot location.
        For a grid size passed down from the Integrator module, two arrays are produced:
a probably empty array and a somwhere occupied array. These arrays are initialized to
the unknown state and are updated based on where each grid center lies in relation to the
robot volume or the various sonar cones as reported by the Scanner module.
        The final view is computed from the two arrays after all the grid centers in each
array have been processed for the given view.

* /

#include "integratorMapper.e"      /* Global variables definition of the variables
                                        shared by this module and the integrator module */
#include  "Sonar.i"              /* Global variables definition of variables shared by this
                                    module and the lower level modules */

#define sqr(z) z * z            /* Square root function definition */

float
        FinView[17][17];    /* Array containing the final grid certainty value */

float
        TetaGrid,           /* Angle formed by the center of the grid being processed,
                                the center of the view and the X axis */
        RRadius = 45,       /* Robot radius */
        RingRadius = 30,    /* Sonar ring Radius */
        RMin = 27.4,        /* Minimum sonar range reading */
        err = 0.1,          /* Sonar range reading error */
        BeamSpan = 30;      /* Sonar beam span */

int
        BeamSelR,           /* Contains the sonar number of the sonar whose beam is
                                on the right of the grid being processed */
        BeamSelL;           /* Contains the sonar number of the sonar whose beam is
                                on the left of  the grid being processed */

float
        Occ,            /* Probability for the grid being processed of being occupied */
        Emp;            /* Probability for the grid being processed of being empty */

float
```

```
    MapScan[24][2];          /* Array that contains the final scan results as
                                processed by the Scanner module */

Mapper()          /* Mapper module main routine */

    {
        int
            i,           /* X coordinates of the grid being processed*/
            j;           /* Y coordinates of the grid being processed*/

        FILE
            *fopen(),    /* C file opening function */
            *obstacle,   /* File used to dump the content of SomwOcc */
            *clear,      /* File used to dump the content of ProbEmp */
            *view;       /* File used to dump a character coded
                            representation of the content of FinView */


/* Opening of the data files */

            obstacle = fopen("OccResult","w");
            clear = fopen("EmpResult","w");
            view = fopen("FinResult","w");

/* Initialization of the arrays */

            for (j = -NumGrid ; j <= NumGrid ; j++)
                for (i = -NumGrid ; i <= NumGrid ; i++)
                {
                ProbEmp[i + NumGrid][j + NumGrid] = 0;
                SomwOcc[i + NumGrid][j + NumGrid] = 0;
                FinView[i + NumGrid][j + NumGrid] = 0;
                }

/* Execution of a scan invoking the Scanner module main routine */

             Scan();

/* Loop controlling the processing of each grid in the view starting from the lower left
corner and proceeding by rows first */

for (j = NumGrid ; j >= -NumGrid ; j--)
                    {
                        for (i = -NumGrid ; i <= NumGrid ; i++)
                            {

/* Call of the subroutine that assigns the probabilities to the grid at coordinates (i,j)*/

                                AssignProb(i,j);
```

```
/* Tresholding of the final grid probability and loading of FinView */

                                    if (SomwOcc[i + NumGrid][j + NumGrid] >
                                        ProbEmp[i + NumGrid][j + NumGrid])
                                          FinView[i + NumGrid][j + NumGrid] =
                                          SomwOcc[i + NumGrid][j + NumGrid];
                                    else if (ProbEmp[i + NumGrid][j + NumGrid] >
                                        SomwOcc[i + NumGrid][j +NumGrid])
                                          FinView[i + NumGrid][j + NumGrid] =
                                          -ProbEmp[i + NumGrid][j + NumGrid];

/* Generation of the result files in the same order as the grid inspection */

                                    fprintf(obstacle, "%6.2f",
                                            SomwOcc[i + NumGrid][j + NumGrid]);
                                    fprintf(clear, "%6.2f",
                                            ProbEmp[i + NumGrid][j + NumGrid]);
                                    if (FinView[i + NumGrid][j + NumGrid] > 0)
                                           fprintf(view, "+ ");
                                    if (FinView[i + NumGrid][j + NumGrid] < 0)
                                           fprintf(view, "- ");
                                    if (FinView[i + NumGrid][j + NumGrid] == 0)
                                           fprintf(view, ". ");
                                    }

/* Print a line feed at the end of every rows */

                                fprintf(obstacle, "\n");
                                fprintf(clear, "\n");
                                fprintf(view, "\n");
                        }

/* Print the grid size used to generate the views at the end of the result files */

                fprintf(view, "\n\nGrid Size = %4.2f cm", GridSize);
                fprintf(obstacle, "\n\nGrid Size = %4.2f cm", GridSize);
                fprintf(clear, "\n\nGrid Size = %4.2f cm", GridSize);

/* Close the result files */

                fclose("OccResult");
                fclose("EmpResult");
                fclose("FinResult");

        }

/* Routine that assigns a probability to the grid located at coordinates (i,j) */

AssignProb(i,j)
```

```
int
        i,              /* X coordinates of the grid being processed*/
        j;              /* Y coordinates of the grid being processed*/

{
        float
                x,      /* X coordinates of the grid being processedin cm */
                y;      /* Y coordinates of the grid being processedin cm */

/* Default value for the left beam, in the case where TetaGrid correspond to a sonar
angle, processing would be done with only the right beam */

                BeamSelL = -1;

/* Assignement of x and y */

                x = i * GridSize;
                y = j * GridSize;

/* Condition that checks if the grid lies within the area currently occupied by the robot.
If so the grid probability of being occupied is 0 and the probability of being empty is 1
* /

        if (-RRadius <= x && x <= RRadius
                        && -RRadius <= y && y <= RRadius)
                        {
                        SomwOcc[i + NumGrid][j + NumGrid] = 0;
                        ProbEmp[i + NumGrid][j + NumGrid] = 1;
                        }

/* If the grid doen't lie within the area occupied by the robot,  the sonar beams on each
sides of the grid are found */

        else
                        {

/* Computation of TetaGrid */

                        if (x == 0 && y > 0)
                                TetaGrid = 90;
                        else if (x == 0 && y < 0)
                                TetaGrid = 270;
                        else
                                TetaGrid = atan(y / x) * (180 / pi);

                        if (x < 0)
                                TetaGrid += 180;
                        else if (y < 0 && x > 0)
                                TetaGrid += 360;
```

```
/* Sonar beam selection */

                    if(fmod(TetaGrid,15)  !=  0)
                            {
                            BeamSelR = floor(TetaGrid / 15);
                            BeamSelL = BeamSelR + 1;
                            }

                    else
                            BeamSelR = TetaGrid/15;

/* Call of the routine that compute the probabilities based on the sonar beam selected */

                    ComputeProb(i,j,x,y);
                    }

        }

/*  Routine that compute the probabilities for the grid at coordinates (i,j) based on the
sonar beam selected */

ComputeProb(i,j,x,y)

        int
                i,              /* X coordinates of the grid being processedin */
                j;              /* Y coordinates of the grid being processedin */

        float
                x,              /* X coordinates of the grid being processedin cm */
                y;              /* Y coordinates of the grid being processedin cm */

        {
/* if the left beam was selected then the probabilities in relation with both beams is
computed separately and then probabilistically merged  together */

                if (BeamSelL >= 0)
                        {
/* Compute probabilities with the right beam...*/

                        BeamProb(BeamSelR,x,y);

/*  ...temporarily  store  them  */
                        SomwOcc[i + NumGrid][j + NumGrid] = Occ;
                        ProbEmp[i + NumGrid][j + NumGrid] = Emp;

/*  ...then  with  left  beam*/

                        BeamProb(BeamSelL,x,y);

/*  ...and  Probabilistically  merge  them  */
```

```
                    ProbEmp[i + NumGrid][j + NumGrid] =
                            (ProbEmp[i + NumGrid][j + NumGrid] + Emp) -
                            (ProbEmp[i + NumGrid][j + NumGrid] * Emp);
                    Occ *= (1 - ProbEmp[i + NumGrid][j + NumGrid]);
                    SomwOcc[i + NumGrid][j + NumGrid] =
                            (SomwOcc[i + NumGrid][j + NumGrid] + Occ) -
                            (SomwOcc[i + NumGrid][j + NumGrid] * Occ);
                    }

/* If only the right beam was selected the no probabilistic merge is required */

                else
                    {
                    BeamProb(BeamSelR,x,y);
                    SomwOcc[i + NumGrid][j + NumGrid] = Occ;
                    ProbEmp[i + NumGrid][j + NumGrid] = Emp;
                    }

        }

/* Routine that compute the probabilities for a grid at coordinates (x,y) for a sigle
sonar beam */

BeamProb(SonarNumber,x,y)

        int
                SonarNumber;            /* Number of the sonar selected */

        float
                x,          /* X coordinates of the grid being processedin cm */
                y;          /* Y coordinates of the grid being processedin cm */

        {
                float
                        xSonar,     /* X coordinate of the sonar position */
                        ySonar,     /* Y coordinate of the sonar position */
                        TetaSonar,  /* Angle formed by the X axis, the sonar position
                                       and the grid at coordinates (x,y) */
                        TetaMain,   /* Angle formed by the sonar beam main axis, the
                                       sonar position and the grid at coordinate (x,y) */
                        RadiusSonar, /* Distance in cm between the grid coordinates and
                                        the sonar position */
                        R,          /* Range measurement extrcted from MapScan for
                                       the selected sonar */
                        RelErr;     /* Relative range measurement error */

/* Computation of the sonar position and its distance to the grid at coordinates (x,y) */

                xSonar = RingRadius * cos(SonarNumber * 15 * pi / 180);
```

```
            ySonar = RingRadius * sin(SonarNumber * 15 * pi / 180);
            RadiusSonar = sqrt(sqr((y - ySonar)) + sqr((x - xSonar)));

/* Computation of TetaSonar */

            if ((x - xSonar) == 0)
                  TetaSonar = 90;
            else
                  TetaSonar = atan((y - ySonar) / (x - xSonar)) * (180 / pi);

            if ((x - xSonar) <= 0)
                  TetaSonar += 180;
            else if ((y - ySonar) < 0 && (x - xSonar) > 0)
                  TetaSonar += 360;

/* Computation of the sonar beam main axis angle */

            TetaMain = abs((SonarNumber * 15) - TetaSonar);

/* Read the Range reading value from MapScan and compute the relative error on it */

            R = MapScan[SonarNumber][1];
            RelErr = err * R;

/* Computation of Emp and Occ based on the grid position within the sonar beam */

            if (RMin < RadiusSonar && RadiusSonar < (R - RelErr))
                  Emp = (1 - sqr((RadiusSonar-RMin) / (R - RelErr - RMin)))
                        * (1 - sqr((2 * TetaMain) / BeamSpan));
            else
                  Emp = 0;

            if (Emp < 0)
                  Emp = 0;

            if ((R - RelErr) < RadiusSonar && (R + RelErr) > RadiusSonar)
                  Occ = (1 - sqr((RadiusSonar - R) / RelErr)) *
                        (1 - sqr((2 * TetaMain) / BeamSpan));
            else
                  Occ = 0;

            if (Occ < 0)
                  Occ = 0;
      }
```

Appendix E

Integrator Module Code

```
/ *

Integrator.c

        This module is responsible for the control of the overall mapping system. It
allows the user to merge views together in order to form a local map.
        When merging views, the Integrator commands the execution of the mapper
module with pre-selected settings for the number of grids in the view. The user is
requested to input the grid size he desires. After each execution of the mapper module,
the occupancy probabilities are integrated with their respective map arrays.
        When an occupancy probability is integrated, if the map array already has a
valuefor that location, the two values are merged probabilistically. On the other hand, if
the new view array values are not matched with already existing map array values, they
are considered as the new map array values. Finally, if an existing map array value is
not updated, its value is decreased.
        The robot remains at the center of the local maps and therefore the grid array
values that "fall off" the map boundaries are dropped.

* /

#include "Sonar.i"    /* Global variables definition of the variable shared by this
                         module and the other mapping system modules */
#include "Sonar.e"    /* Declaration of the variables defined in Sonar.i */

/* Global variables definition of the variable shared by this module and the user
application modules */

#include "SonarRover.e"
#include "SonarGraphics.e"
#include "CGrid.e"


void Integrator()     /* Integrator module main routine */
      {
            int
                  i,      /* X grid coordinate index */
                  j,      /* Y grid coordinate index */
                  x,      /* Shifted X grid coordinate index */
                  y;      /* Shifted Y grid coordinate index */

/* Local map empty, occupied and final arrays */
```

```
        float
                Map1Emp[GRIDSIZE][GRIDSIZE],
                Map1Occ[GRIDSIZE][GRIDSIZE],
                Map1Fin[GRIDSIZE][GRIDSIZE];

/* External files for storage of the local map arrays */

        FILE
                *fopen(),
                *Mapobs,
                *Mapclr,
                *Mapview;

/* User interface application related variables */

        DialogPtr
                GridDial,
                CoordDial,
                ContinueDial;

        int
                theItem;

        int
                TheItem;
        int
                TheItemType;
        Handle
                TheItemHandle;
        Rect
                Box;
        Str255
                GridString,
                XString,
                YString,
                OrString;

        long int
                LongTemp;


/* Dialog box to ask the user for a grid size */

                GridDial = GetNewDialog(267, 0L, (WindowPtr)-1);
                NumToString( (long)GridSize, GridString);
```

```
                GetDItem(GridDial, 1, &TheItemType,
                        &TheItemHandle, &Box);
                SetIText(TheItemHandle, GridString);
                SelIText(GridDial,1,0,2);
                ModalDialog(0L, &theItem);
                GetDItem(GridDial, 1, &TheItemType,
                        &TheItemHandle, &Box);
                GetIText(TheItemHandle, GridString);
                StringToNum( GridString, &LongTemp);
                DisposDialog(GridDial);
                GridSize = LongTemp;


/* Initialization of the local map arrays */

                for (y = -NumGridMap ; y <= NumGridMap ; y++)
                        for (x = -NumGridMap ; x <= NumGridMap ; x++)
                        {
                        i = x + NumGridMap;
                        j = y + NumGridMap;
                        Map1Occ[i][j] = 0;
                        Map1Emp[i][j] = 0;
                        Map1Fin[i][j] = 0;
                        }

/* Initialization of the origin position */

                LastPos[0] = 0;
                LastPos[1] = 0;
                LastOri = 0;

/* Main loop, acquires views until the user decides to stop */

        do
                {

/* Dialog box to ask the user for the robot position and orientation */

                CoordDial = GetNewDialog(268,0L,(WindowPtr)-1);

                NumToString( (long)RobotPos[0], XString);
                GetDItem(CoordDial, 1, &TheItemType,
                        &TheItemHandle, &Box);
                SetIText(TheItemHandle, XString);
                SelIText(CoordDial,1,0,4);
```

```
NumToString( (long)RobotPos[1], YString);
GetDItem(CoordDial, 2, &TheItemType,
        &TheItemHandle, &Box);
SetIText(TheItemHandle, YString);

NumToString( (long)RobotOri, OrString);
GetDItem(CoordDial, 3, &TheItemType,
        &TheItemHandle, &Box);
SetIText(TheItemHandle, OrString);

ModalDialog(0L, &theItem);

GetDItem(CoordDial, 1, &TheItemType,
        &TheItemHandle, &Box);
GetIText(TheItemHandle, XString);
StringToNum( XString, &LongTemp);
RobotPos[0] = (float)LongTemp;

GetDItem(CoordDial, 2, &TheItemType,
        &TheItemHandle, &Box);
GetIText(TheItemHandle, YString);
StringToNum( YString, &LongTemp);
RobotPos[1] = (float)LongTemp;

GetDItem(CoordDial, 3, &TheItemType,
        &TheItemHandle, &Box);
GetIText(TheItemHandle, OrString);
StringToNum( OrString, &LongTemp);
RobotOri = (float)LongTemp;
```

/* Conversion of robot position and orientation from robot counts into centimetres and degrees */

```
RobotPos[0] *= 0.3048;
RobotPos[1] *= 0.3048;
RobotOri *= 0.3516;
```

/* Register the robot position with the room model in the user interface application */

```
XPosition = RobotPos[0] + 322 ;
YPosition = RobotPos[1] + 468 ;
ScaledX += (int)( ScalingFactor * (float)XPosition);
ScaledY += (int)( ScalingFactor * (float)YPosition);

DisposDialog(CoordDial);
```

```
/* Call up the main routine of the Mapper module */

                    Mapper();

/* Call up the routine to merge the new view into the local map */

                    MapGridProb(Map1Emp,Map1Occ);

/* Transfer temporary arrays into local map arrays */

                    for (y = NumGridMap ; y >= -NumGridMap ; y--)
                        for (x = -NumGridMap ; x <= NumGridMap ; x++)
                            {
                            i = x + NumGridMap;
                            j = y + NumGridMap;
                            Map1Occ[i][j] = TempOcc[i][j];
                            Map1Emp[i][j] = TempEmp[i][j];
                            if (Map1Occ[i][j] > Map1Emp[i][j])
                                Map1Fin[i][j] = Map1Occ[i][j];
                            else if (Map1Emp[i][j] > Map1Occ[i][j])
                                Map1Fin[i][j] = -Map1Emp[i][j];
                            else
                                Map1Fin[i][j] = 0;
                            }

/* Update last position variables */

                    LastPos[0] = RobotPos[0];
                    LastPos[1] = RobotPos[1];
                    LastOri = RobotOri;

/* compute final local map */

                    for (y = NumGridMap ; y >= -NumGridMap ; y--)
                        {
                        for (x = -NumGridMap ; x <= NumGridMap ; x++)
                            {
                            i = x + NumGridMap;
                            j = y + NumGridMap;
                            if (Map1Fin[i][j] > 0)
                                Grid[i][j] = 1;
                            if (Map1Fin[i][j] < 0)
                                Grid[i][j] = -1;
                            if (Map1Fin[i][j] == 0)
                                Grid[i][j] = 0;
                            }
```

```
                        if (!CGridOn) CGridOn = TRUE;
                        }

/* update user interface application screen display */

                    UpdateWindow(graphWindow);
                    SpoolPicture();

/* Query the user to find out if he wants to add another view */

                    ContinueDial = GetNewDialog(269, 0L,
                                (WindowPtr)-1);
                    ModalDialog(0L, &theItem);
                    DisposDialog(ContinueDial);
                    }
                while (theItem == 1);

/* Dump the results into the external files */

                Mapobs = fopen("MapOccResult","w");
                Mapclr = fopen("MapEmpResult","w");
                Mapview = fopen("MapFinResult","w");

                for (y = NumGridMap ; y >= -NumGridMap ; y--)
                        {
                        for (x = -NumGridMap ; x <= NumGridMap ; x++)
                                {
                                i = x + NumGridMap;
                                j = y + NumGridMap;
                                fprintf(Mapobs, "%6.2f", Map1Occ[i][j]);
                                fprintf(Mapclr, "%6.2f", Map1Emp[i][j]);
                                if (Map1Fin[i][j] > 0)
                                        fprintf(Mapview, "+ ");
                                if (Map1Fin[i][j] < 0)
                                        fprintf(Mapview, "- ");
                                if (Map1Fin[i][j] == 0)
                                        fprintf(Mapview, ". ");
                                }
                        fprintf(Mapobs, "\n");
                        fprintf(Mapclr, "\n");
                        fprintf(Mapview, "\n");
                        }

                fprintf(Mapview, "\n\nGrid Size = %4.2f cm", GridSize);
                fprintf(Mapobs, "\n\nGrid Size = %4.2f cm", GridSize);
                fprintf(Mapclr, "\n\nGrid Size = %4.2f cm", GridSize);
```

```
            fclose("MapOccResult");
            fclose("MapEmpResult");
            fclose("MapFinResult");

        }

/* MapGridProb subroutine */

MapGridProb(Map1Emp,Map1Occ)

/* Definition of the local map empty and occupied arrays passed from Integrator */

        float
                Map1Emp[][GRIDSIZE],
                Map1Occ[][GRIDSIZE];

        {
                float

/* Translated and rotated X and Y origin shift between the old robot position coordinates
system and the new robot position coordinate system */

                        XP,
                        YP,

/* Rotated X and Y origin shift between the old robot position coordinates system and the
new robot position coordinate system */

                        XShift,
                        YShift,

/* Translated X and Y origin shift between the old robot position and the new robot
position */

                        DeltaPos[2],
                        DeltaOri;

                int

/* X and Y grid coordinates in relation to the new origin */

                        XPP,
                        YPP,

                        i,      /* X grid coordinate index */
```

```
        j,      /* Y grid coordinate index */
        x,      /* X shifted grid coordinate index */
        y;      /* Y shifted grid coordinate index */

/*Compute the translation and rotation deltas */

        DeltaPos[0] = LastPos[0] - RobotPos[0];
        DeltaPos[1] = LastPos[1] - RobotPos[1];
        DeltaOri = LastOri - RobotOri;

/* Compute the local map grids empty and occupied probabilities  adding the results of
the new view */

        for (y = -NumGridMap ; y <= NumGridMap ; y++)
                for (x = -NumGridMap ; x <= NumGridMap ; x++)
                {

/* Compute local map grid coordinates in relation to the new origin */

                i = x + NumGridMap;
                j = y + NumGridMap;
                XShift = (DeltaPos[0] * cos(RobotOri * pi / 180)) +
                        (DeltaPos[1] * sin(RobotOri * pi / 180));
                YShift = (-DeltaPos[0] * sin(RobotOri * pi / 180)) +
                        (DeltaPos[1] * cos(RobotOri * pi / 180));
                XP = (x * GridSize) - XShift;
                YP = (y * GridSize) - YShift;
                XPP = round(((XP * cos(DeltaOri * pi / 180)) +
                       (YP * sin(DeltaOri * pi / 180))) / GridSize) +
                NumGridMap;
                YPP = round(((-XP * sin(DeltaOri * pi / 180)) +
                       (YP * cos(DeltaOri * pi / 180))) / GridSize) +
                NumGridMap;

/* If the grid is in the new view */

                if ( abs(x) <= NumGrid && abs(y) <= NumGrid)

/* If the local map probabilities are unknown */

                        if (XPP > (2 * NumGridMap) || YPP >
                        (2 * NumGridMap) || XPP < 0 || YPP < 0)
                        {

/* The view probabilities are assigned to the local map */
```

```
                        TempEmp[i][j] = ProbEmp[x + NumGrid][y + NumGrid];
                        TempOcc[i][j] = SomwOcc[x + NumGrid][y + NumGrid];
                        }

/* If the local map probabilities are not unknown */

                        else
                        {

/* The view probabilities are probalistically merged with the local map probabilities
* /

                        TempEmp[i][j] = (ProbEmp[x + NumGrid][y + NumGrid]
                                        + Map1Emp[XPP][YPP]) -
                                        (ProbEmp[x + NumGrid][y + NumGrid]
                                        * Map1Emp[XPP][YPP]);

                        SomwOcc[x + NumGrid][y + NumGrid] *=
                                                (1 - TempEmp[i][j]);

                        TempOcc[i][j] = (SomwOcc[x + NumGrid][y + NumGrid]
                                        + Map1Occ[XPP][YPP]) -
                                        (SomwOcc[x + NumGrid][y + NumGrid]
                                        * Map1Occ[XPP][YPP]);
                        }

/* If the grid is not in the new view */

                    else

/* If The local map probabilities are unknown */

                        if (XPP > (2 * NumGridMap) || YPP >
                        (2 * NumGridMap) || XPP < 0 || YPP < 0)
                        {

/* Probabilities remain unkhown */

                        TempEmp[i][j] = 0;
                        TempOcc[i][j] = 0;
                        }

/* If The local map probabilities are not unknown */

                        else
                        {
```

```
/* the local map probabilities are decreased by 10% */

                        TempEmp[i][j] = Map1Emp[XPP][YPP] * 0.9;
                        TempOcc[i][j] = Map1Occ[XPP][YPP] * 0.9;
                        }

                }

        }



/* Real number rounding function */

int  round(x)

        float
                x;

        {
                float
                        temp;

                temp = fmod(x,1);
                if (abs(temp) > 0.5)
                        x = ceil(x);
                else
                        x = floor(x);
                return(x);
        }
```

Appendix F

Global Variable Files

```
/ *

IntegratorMapper.e

* /

extern float
       ProbEmp[][17],
       SomwOcc[][17];

extern int
       NumGrid;

extern float
       GridSize,
       pi;


/ *

ScannerMapper.e

External declaration of the Scanner/Mapper interface variables

* /

extern float
       MapScan[][2];


/* SonarControlScanner.e

External declaration for the Sonar Control/Scanner interface variable

* /

extern unsigned int
       MaxDepth,
       GainProfile,
       RangeMode,
```

```
        FiringDelay;

extern unsigned int
        NewScan[][2];

extern unsigned int
        CommandWord;



/ *
CGrid.h
- - - - - - -

Header file for certainty grid routines.  Global data declared.
* /

int
        NumGrid = 8,
        NumGridMap = NUMGRIDMAP;
float
        ProbEmp[17][17],
        SomwOcc[17][17];
float
        LastPos[2] = {0, 0},
        LastOri = 0,
        RobotPos[2] = {0, 0},
        RobotOri = 0,
        GridSize = 25;
float
        pi = 3.141592654;
float
        *TempEmp[GRIDSIZE],
        *TempOcc[GRIDSIZE];
int
        *Grid[GRIDSIZE];



/ *
Sonar.h
- - - - - - - -

Header file for Sonar.c

Created 3 August, 1988 by Sherman Lang.
* /
```

```
int
        Done;                   /* flag to tell when to quit */
unsigned int
        Bit[16] = {   1,   2,   4,   8,          /* Bit mask values */
                     16,  32,  64, 128,
                    256, 512, 1024, 2048,
                   4096, 8192, 16384, 32768 },

        Command[16] = {0,  256,  512,  768,     /* 0 to 15 shifted left 8 bits */
                    1024, 1280, 1536, 1792,
                    2048, 2304, 2560, 2816,
                    3072, 3328, 3584, 3840 };
float
        World[500][4];
int
        Walls;
int
        DataRefNum; /* reference number of file returned by FSOpen opened to save
                    data */
int
        SaveData = FALSE;    /* flag to tell when to save data */
FILE
        *DataFile;           /* scan data file */
int
        CGridOn = FALSE;     /* flag to tell when to display certainty grig */



/ *
SonarController.h
- - - - - - - - - - - - - - - - -

Header file of ultrasound controller state variables for Sonar.c

Created 3 August, 1988 by Sherman Lang.
* /

unsigned int
        LastScan[24][2],                /* Last scan displayed */
        NewScan[24][2] = {24, 0},       /* New scan acquired */
        SingleSensor = 0;               /* Single sensor to be excited */
int
        ExcitationMask[24],         /* Excitation Mask: 1-enabled, 0-disabled */
        FiringPattern[24];          /* Firing Pattern: 1-enabled, 0-disabled */
unsigned int
        GainProfile,                    /* Gain Profile: 0, 1, 2, 3   * /
```

```
        RangeMode,                  /* Range Mode: 0-near, 1-far */
        FiringDelay,                /* Firing Delay: 0-255 */
        MaxDepth;                   /* Maximum depth read */
unsigned int
        CommandWord;                /* Command to be acknowledged, command
                                    error returned */


/*
SonarGraphics.h
---------------

Header file of graphics variables for Sonar.c

Created 3 August 1988 by Sherman Lang.
*/

MenuHandle
        SonarMenu[7];               /* menu handle for application */
WindowPtr
        graphWindow,                /* pointer to graphics window */
        textWindow,                 /* pointer to text window */
        WhichWindow;                /* pointer to determine which window mouse
                                    pressed in */
WindowRecord
        graphWRecord,               /* window record for graphics window */
        textWRecord;                /* window record for text window */
int
        graphWindSize,              /* size of square graphics window */
        textWindXSize = 130,        /* width of text window */
        textWindYSize = 320;        /* height of text window */
Rect
        graphRect,                  /* rectangle for graphics window */
        textRect,                   /* rectangle for text window */
        DragBoundary;               /* boundary on dragging */
float
        RobotPoints[24][2],         /* 24 points describing the robot polygon */
        SensorOffsets[24][2],       /* 24 points describing the offsets from center of
                                    the sensors */
        ScalingFactor = 1.0;        /* scaling factor for display */
int
        GridOn = FALSE;             /* flag to tell if the grid is on */
int
        ScaledX, ScaledY;       /* X and Y coordinates of robot scaled to window size */
int
        MapRead = FALSE,            /* flag to tell if map read */
```

```
        DisplayMap = TRUE;          /* flag to tell if map should be diaplayed */
THPrint
        SonarPrRcrd;                /* print record */
PicHandle
        WorldPic;           /* picture for recording world model for display */
PicHandle
        GridPic;                    /* picture for recording grid for display */


/*
SonarRover.h
- - - - - - - - - - - - -

Header file of rover state variables for Sonar.c

Created 3 August, 1988 by Sherman Lang.
*/
int
        XPosition,                  /* X coordinate of robot center */
        YPosition;                  /* Y coordinate of robot center */
double
        Theta = 0;                  /* Orientation of Robot */
float
        CMperCount = 0.7143,        /* number of centimeters per count  */
        InPerCount = 0.2812,        /* number of inchs per count */
        RobotRadius = 30,           /* radius of robot in CM */
        CosTheta,                   /* cosine of Theta */
        SinTheta;                   /* sine of Theta */


/*
SonarSerial.h
- - - - - - - - - - - - -

Header file of serial port and driver variables for 'Sonar Map'.

Created 3 August, 1988 by Sherman Lang.
*/

int
        SerialIn,                   /* Input serial driver reference number */
        SerialOut;                  /* Output serial driver reference number */

SPortSel
        SerialPort;                 /* Serial port selected */
```

```
unsigned char
      SerBuf[10000];          /* Buffer for serial input channel */


/ *
Sonar.i
- - - - - - - -

Include files for Sonar.c and subroutines.

Created 16 June, 1988 by Sherman Lang.
Modified 28 June, 1988 by SYTL.
* /

#define _MC68881_             /* for 68881 code generation */

/ *
#include <QuickDraw.h>
#include <MacTypes.h>
#include <FontMgr.h>
#include <WindowMgr.h>
#include <MenuMgr.h>
#include <TextEdit.h>
#include <EventMgr.h>
#include <DialogMgr.h>
#include <DeskMgr.h>
#include <ControlMgr.h>
#include <ResourceMgr.h>
#include <FileMgr.h>
#include <ToolboxUtil.h>
#include <pascal.h>
#include <StdFilePkg.h>
#include <OSUtil.h>
* /

#include <SerialDvr.h>
#include <math.h>
#include <stdio.h>
#include <strings.h>
#include <PrintMgr.h>

/* defines for TRUE and FALSE
#define TRUE 1
#define FALSE 0

/* define resource IDs of menus */
```

```
#define appleID 128
#define fileID 129
#define editID 130
#define doID 131
#define setID 132
#define resetID 133
#define lpnID 134

/* define menu names */
#define appleM 0
#define fileM 1
#define editM 2
#define goM 3
#define setM 4
#define resetM 5
#define lpnM 6

/* define menu item numbers */
#define fmNew 1
#define fmOpen 2
#define fmClose 4
#define fmSave 5
#define fmSaveAs 6
#define fmRevert 7
#define fmPageSetup 9
#define fmPrint 10
#define fmReadMap 12
#define fmSaveData 13
#define fmQuit 15

#define emUndo 1
#define emCut 3
#define emCopy 4
#define emPaste 5
#define emClear 6
#define emSelectAll 7
#define emShowClipboard 8

#define dmSingleScan 1
#define dmAutoScan 2
#define dmScanOverSettings 3
#define dmSingleSensor 5
#define dmClosestSensor 6
#define dmReSendLast 8
#define dmGetOpParm 9
#define dmWarningBeep 11
#define dmTest 13
```

```
#define smMask 1
#define smPattern 2
#define smRangeGain 4
#define smDelay 5
#define smMaxDepth 6
#define smPosition 8
#define smOrientation 9
#define smScaling 11
#define smGrid 12

#define rmFlush 1
#define rmResetOp 3
#define rmResetCon 4

#define lpmIntegrator 1
#define lpm2 2
#define lpm3 3

#define NUMGRIDMAP 16
#define GRIDSIZE 2*NUMGRIDMAP+1
```