BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-91-M5

An Implementation of Certainty Grids for Mobile Robot Exempt from the Higher Order Echo Reflection

by
Tu-Hsin Tsai

An Implementation of Certainty Grids for Mobile Robot Exempt from the Higher Order Echo Reflection

by
Tu-Hsin Tsai

Department of Computer Science
Brown University

Thesis

Submitted in partial fulfillment of the requirments
for the degree of Master of Science in the
Department of Computer Science at Brown University

December, 1990

This thesis by Tu-Hsin Tsai is accepted in its present form by the Department of Computer Science as satisfying the thesis requirement for the degree of Master of Science.

Data 12/20/90

Thomas L. Dean
Advisor

Contents

1 Introduction

A numerical representation of uncertainty and incomplete sensor knowledge called Certainty Grids, suggested by Elfes [1], has been used in several mobile robot control programs and has proven itself to be a powerful and efficient solution for world modeling [2].

Since the use of sonar sensors suffers from high order reflections inherently, the world map may be contaminated by these high order reflective noise, and become improper if our robot goes into a narrow hallway. In other words, a certainty grid is easy to build up and update but hard to maintain in real world.

Durrant-Whyte has introduced a new feature called RCD (Regions of Constant Depth) [3] to filter sonar reading data. It provides an effective way to extract RCD's from each scan and then classify their order[1]. This method therefore is helpful for us to prevent our map from contaminated by higher order reflections.

The project I have done is to build a software framework running on a SUN 4/60 workstation connected to our mobile robot that will maintain a probabilistic, geometric map of the robot's surroundings exempt from the higher order noise as it moves.. This certainty grid representation allows the map to be incrementally updated from sonar sensors. To avoid higher order reflection noises as the robot moves at a narrow hallway, each sensor holds a specific sensory data filter to extract lower order RCD's from each scan. They filter out improper readings and only pick up the lower order data for map update. This map is then used for higher task (navigator), as will be described in section 5, to avoid obstacle and select a local clear paths. Moreover, its certainty grid representation has been extended in the time dimension and used to classify static and dynamic obstacles. With this classification, our robot is able to choose the proper action to avoid obstacle ahead, either keep waiting until the moving objects pass away or make a detour directly.

There are more discusses in the following sections. Section 2 describes the physical sonar sensor system and the map output of our robot. Since our robot uses certainty grid as its space and object representation. The grid is

[1] Durrant-Whyte defines the order of a range measurement as the number of surfaces the sound has reflected off before returning to the transducer.

updated by incoming sensory data recurrently. A discuss on these is included in section 3. Section 4 explains the approach to extract lower order RCD's from each scan. Finally, the architecture of the whole system is presented in section 5.

2 The Robot

2.1 The Sensor System

Our mobile robot hires eight sonar sensors as its input devices. Each sensor was fixed at different orientation, as shown in Fig 1, so that the robot can observe its environment through these devices.

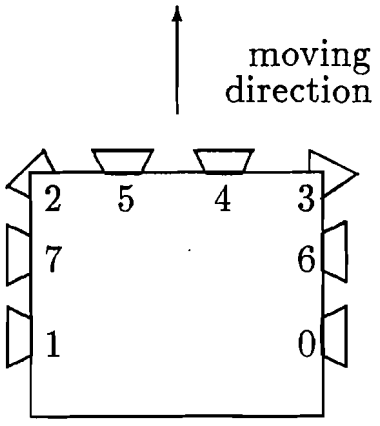


Figure 1: The orientation and place for each sonar sensor. Each sensor has its own id number.

In order to avoid the interference by adjacent sonar sensors, the sonar firing sequence is set as 3,1,4,0,2,6,7,5 at a interval of 30 milliseconds. The sensors then have at least 90 milliseconds to prevent the delayed reflection of their adjacent sonar sensors at the same side.

2.2 Real-time Map Display

The robot's real-time map, a 80 $\times$ 80 grid map, is shown through X-window on Sun workstation. The probabilistic value in each cell is repre-

sented with 32 gray intensity levels, with 0 as pure bright (white) and 1 as pure dark (black).

3 The World Representation

3.1 The Certainty Grid Representation

Our robot used the certainty grid as the space and object representation. Space is represented as a grid of cells, each mapping an area 20 centimeters on a side. and containing a probabilistic estimate of its state. The state variable $S(C)$ associated to a cell C is defined as a discrete random variable with two states, occupied and empty, denote OCC and EMP, and where

$$P[S(C) = OCC] + P[S(C) = EMP] = 1$$

Each cell has, therefore, an associated probability mass function. Consequently, the uncertainty grid corresponds to a discrete-state (binary) random field [4]. A realization of the uncertainty grid is obtained by estimating the state of each cell using sonar sensor data.

3.2 Bayesian Estimation of the Uncertainty Grid

Since a robot can only observe its environment indirectly through the use of sensors, the recovery of a spatial world model from sensor data can be modeled using a estimation theory approach. Fig 2 shows the process for the estimation of uncertainty grid from range sensor data.

Initially, a sensor range reading r is interpreted using a stochastic sensor model, defined by the probability density function $p(r|z)$, where z is the actually distance to the object. Subsequently, a Bayesian estimation procedure is used to update the Occupancy Grid cell state probabilities. Finally, a deterministic world model can be obtained, using the maximum a posteriori(MAP) decision rule to assign discrete states to the cells, labeling them as OCCUPIED or EMPTY. Then our tasks can operate directly on this Occupancy Grid model.

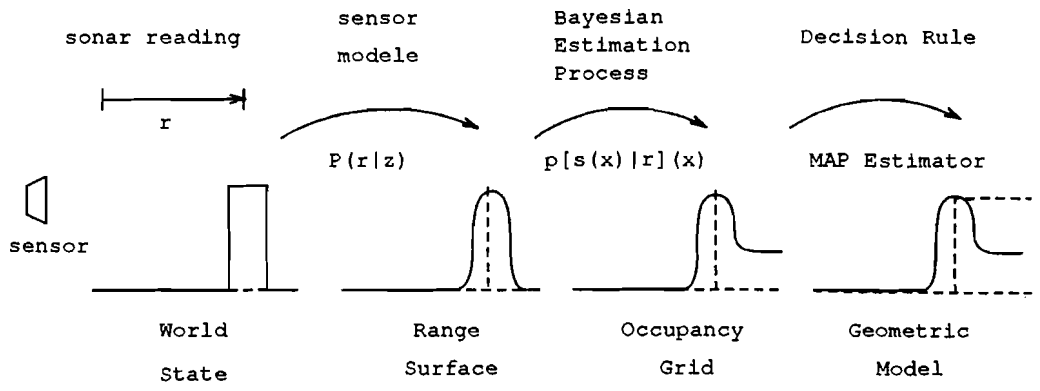


Figure 2:

3.3 Sequential Updating of the Occupancy Grid

In our system, the map is updated as the robot moves. Information from multiple sensor readings taken from different viewpoints is composed to improve the map. To allow the incremental composition of sensory information, we use a sequential updating formulation of Bayes theorem, derived by Elfes [1, chapter 4]. Given the current probabilistic estimate of the state of a cell $s(C_i)$, $P[s(C_i) = OCC|\{r\}_t]$, based on observations $\{r\}_t = \{r_1, \ldots, r_t\}$, and a new observation r_{t+1}, we have:

$$P[s(C_i) = OCC|\{r\}_{t+1}] = \frac{P[r_{t+1}|s(C_i) = OCC]P[s(C_i) = OCC|\{r\}_t]}{\sum_{s(C_i)} P[r_{t+1}|s(C_i) = OCC]P[s(C_i) = OCC|\{r\}_t]} \quad (1)$$

In this recursive procedure, the previous estimate of the cell state, $P[s(C_i = OCC|\{r\}_t]$, serves as the prior and is obtained directly from the Occupancy Grid. After computing the update using the sensor-derived terms, the new cell state estimate $P[s(C_i) = OCC|\{r\}_{t+1}]$ is subsequently stored again in the map (Fig 3).

3.4 Conflict Update

Since the use of sonar sensors has a inherent shortage, suffering from higher order echo reflections, especially at a narrow hallway, it occurs fre-

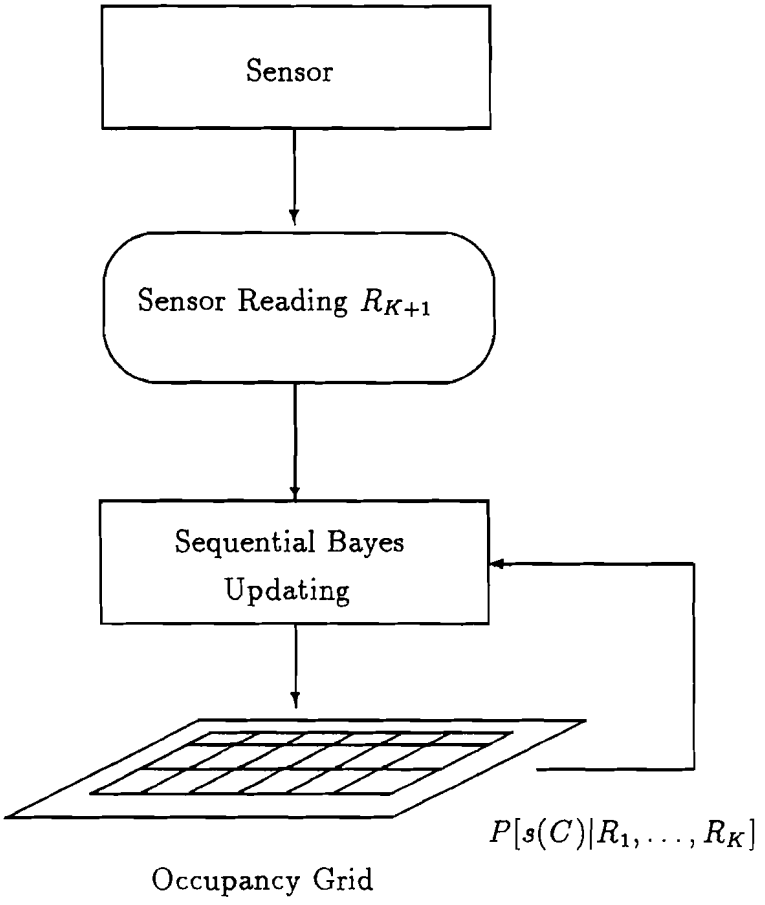


Figure 3: Sequential Occupancy Grid Updating.

quently that the new observation conflicts the previous estimate. Therefore the recursive procedure may be modified to prevent conflict update.

In our system, the state of cell C, $s(C)$, is defined as a discrete random variable with two states, *occupied* and *empty*. We may use a decision rule [2] to classified all the cells of grid. We can regard to update the Occupancy Grid as a Bayes Decision Procedure. The probability of occupancy for each cell $P[s(C_i) = OCC|\{r\}_t]$ serves as the state of nature as well as the probability density function itself. For each new observation r_{t+1}, if the state of nature $P[s(C_i) = OCC|r_{t+1}]$ on the new observation conflicts with the state of nature $P[s(C_i) = OCC|\{r\}_t]$ base on the previous observation, we have,

$$P(error|r_{t+1}) = \begin{cases} P[s(C_i) = OCC|\{r\}_t] & \text{when } s(C_i) \text{ is OCC.} \\ 1 - P[s(C_i) = OCC|\{r\}_t] & \text{when } s(C_i) \text{ is EMP.} \end{cases}$$

We then can define the loss function as:

$$\lambda(r_{t+1}) = \begin{cases} 1 & \text{when conflict occurs} \\ 0 & \text{otherwise} \end{cases}$$

Therefore the Bayes Risk can be derived as follows,

$$R[s(C_i)|r_{t+1}] = \sum_{s(C_i)} \lambda(r_{t+1}) P(error|r_{t+1})$$

Bayes Risk serves as a filter in our updating procedure. We compute the Bayes Risk for each cell before taking the new observation into account. If a high risk occurs [3], we reduce the effect of the new observation to our map.

In practice this procedure can prevent the region of the map where the robot has traveled from being contaminated by subsequent high order reflections. Unfortunately, this procedure also deters our robot from recovering the part of map that has been disturbed by high order reflections before it gets into that region. Hence the following section will describe an alternative to recover our map from contamination.

[2] For the sake of simplicity, we use the value 0.5 as our decision boundary, i.e., if $P(s(C_i) = OCC) > 0.5$, $s(C_i) = OCC$. Otherwise $s(C_i) = EMP$.

[3] In our program the value 0.85 has been suggested as a threshold of high risk.

4 Abstract Feature Extraction from Sensor Data

It is not appropriate to use raw range data from sonar sensors directly. The main reason is that raw data contain noises or themselves are noises. We then use the approach provided by Durrant-Whyte to group range readings from each sensor into RCD's (Regions of Constant Depth) [3], and extract lower order RCD's for updating our map.

4.1 RCD's Grouping and Extraction

In order to obtain a high sampling speed, the sonar sensors were fixed at eight different orientation. In such a case, each range measurement has no local supports. To overcome this disadvantage, our system employs three s statistic factors for each sonar sensor to group range reading into RCD's and classify their order. There are mean (μ), variance (σ^2) and the number of range reading samples within the current constant interval (δ).

We use the following formulations to group range readings into RCD's,

$$0.84 \times E[\{r\}_\delta] \leq r_{\delta+1} \leq 1.16 \times E[\{r\}_\delta] \tag{2}$$

$$Var[\{r\}_\delta] \leq 1000 \tag{3}$$

For each scan, if the new range reading $r_{\delta+1}$ satisfies equation(2) and (3), $r_{\delta+1}$ is taken into the group $\{r\}_\delta$. Otherwise, the current constant interval ends up and the new constant interval begins at $r_{\delta+1}$.

The next step is to classify the current RCD's order. We can determine a RCD's to be lower order (either first or second order) as long as the current constant interval contains morn than a fixed number of constant reading samples [4]. Therefore those range readings which are grouped into a wide RCD's are picked up to update our map.

4.2 Map Recovering

Since the navigator keeps our robot moving parallel to the surfaces of obstacles in the environment, this procedure makes the side sensors gather

[4] In our system, five samples width is suggested. More detail will be discussed in section 5.

RCD's as widely as possible. Readings from side sensors in a wide RCD's are then validated to be first order and adapted to update the regardless of the map conflictness. So the probabilistic estimate procedure in section 3.3 can be rewritten as:

$$P[s(C_i) = OCC|\{r\}_{\delta+1}] = P[r_{\delta+1}|s(C_i) = OCC] \quad (4)$$

With this replace update formula, our robot is able to quickly recover the part of the map that might be contaminated by the front sensors' estimate in the previous time.

5 The Framework Architecture

The framework of our mobile robot includes a number of application-specific processes (or module). They are:

- Initialization.
- Sensory Control.
- Map Updating.
- Navigator.
- Map Display.

These processes are integrated as in Fig 4.

5.1 Initialization

The first process in our system is to set up global variables initial values:

1. The probabilistic estimate of occupancy for each cell c_i is set to be the unknown state ($P[s(C_i) = OCC] = 0.5$) initially.
2. Three statistic factors for each sonar sensor S_i are set to 0 ($\mu_{S_i} = 0, \sigma^2_{S_i} = 0, \delta_{S_i} = 0$).

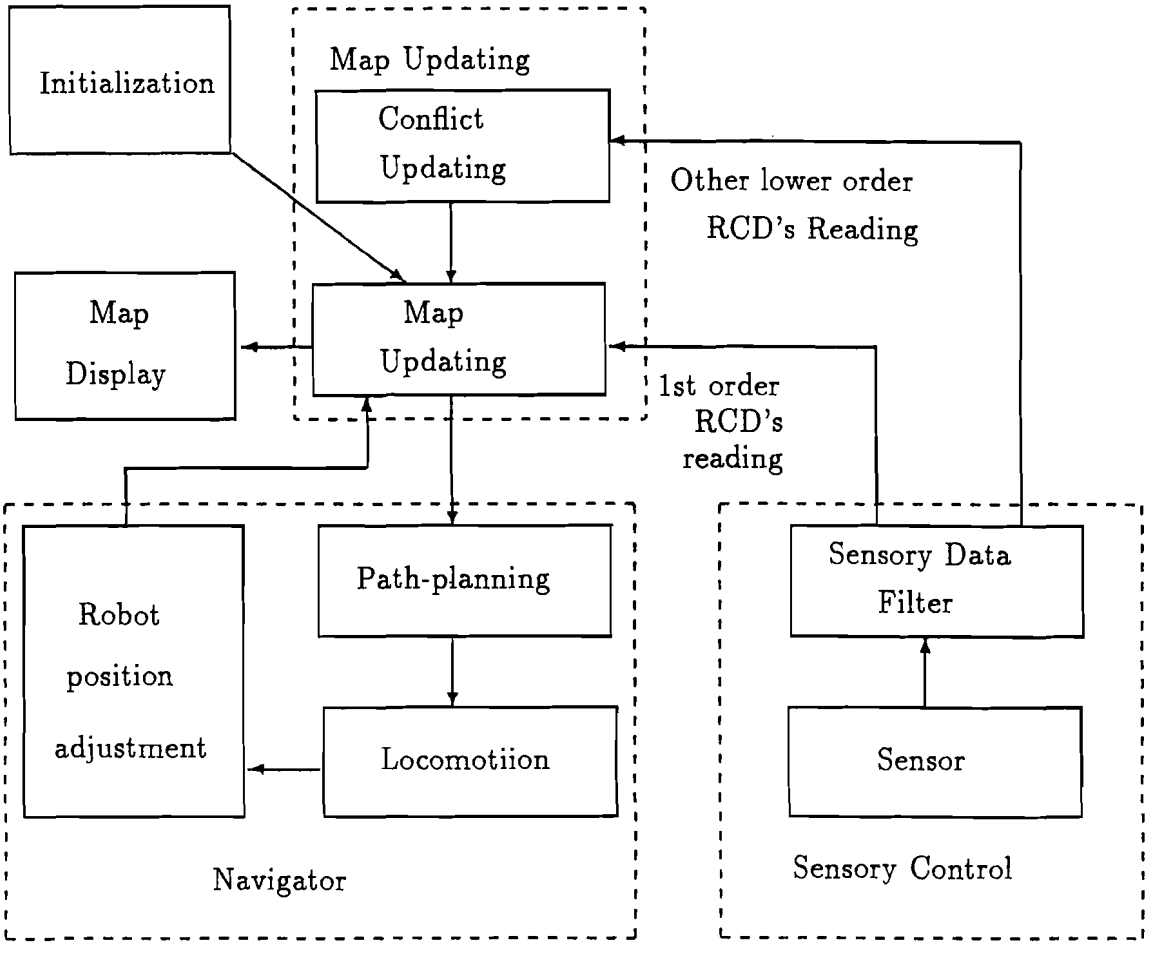


Figure 4: Architecture for framework.

5.2 Sensory Control

The main part of Sensory Control process is Sensory Data Filter. Each sonar sensor holds three statistic factors ($\mu, \sigma^2, and\ \delta$) to group new coming range readings. A new range reading from sensor S_i is put in the current RCD's group of S_i as long as equation (2) and (3) are satisfied. Each time a new range reading is associated in the current RCD's group, the three factors will be updated immediately:

$$\begin{cases} \mu_{\{r\}_{\delta+1}} &= \frac{\delta \times \mu_{\{r\}_\delta} + r_{\delta+1}}{\delta+1} \\ \sigma^2_{\{r\}_{\delta+1}} &= \frac{\sigma^2_{\{r\}_\delta} \times \delta}{\delta+1} + \frac{(\mu_{\{r\}_\delta} - r_\delta)^2}{\delta} \\ \delta &= \delta + 1 \end{cases}$$

Since these three recurrences take only constant time, the factors' updating is then efficient and irrelevant to the size of samples.

In the other hand, if a new coming reading r_{new} is not satisfied equation (2) or (3) and is then not belong to the current RCD's group, the current RCD's group then terminate immediately and a new RCD's group begins at r_{new}. The sensory factors are then initialized :

$$\begin{cases} \mu_{\{r\}} &= r_{new} \\ \sigma^2_{\{r\}} &= 0 \\ \delta &= 1 \end{cases}$$

In general, lower order target (like wall, corner, etc...) have wide RCD's because they are good reflectors of acoustic energy [3]. We then define a threshold β to delimit the reflection returning from higher order targets and lower order ones. Hence, if a RCD's is greater than β, the readings in this RCD's group are validated to be lower order, and pick up to update our robot's map. In our practice experiences, β is defined to be 5.

Since the navigator keeps the robot moving parallel to the surface of objects, the range readings from the side sensors are then confirmed to be first as long as they are in a wide RCD's group.

In summary, the Sensory Control Process extracts the lower range reading from sensors' input, splits them into two (first order RCD's and other lower order RCD's), and then feeds them into Map Updating Process for map updating.

5.3 Map Updating

A range reading passing through the sensory filter is then fed into this process for map update. Each reading is then adds fifteen degrees swath of emptiness and fifteen degrees arc of occupancy to update the certainty grid. Rather than using a single unified sensor model to update the global map, we have two sensor models (Fig 5) and two updating formulas (1) and (4), to deal with three different situations. They are described as follows:

5.3.1 No Conflicts ; Normal Updating

In the case of no conflicts between the new coming range reading and the current map, the first order sensory model in Fig 5 (a) serves as the conditional distribution $P[r|s(C_i)]$. And the map is updated by the formula (1) in section 3.3.

5.3.2 Conflicts ; Replace Updating

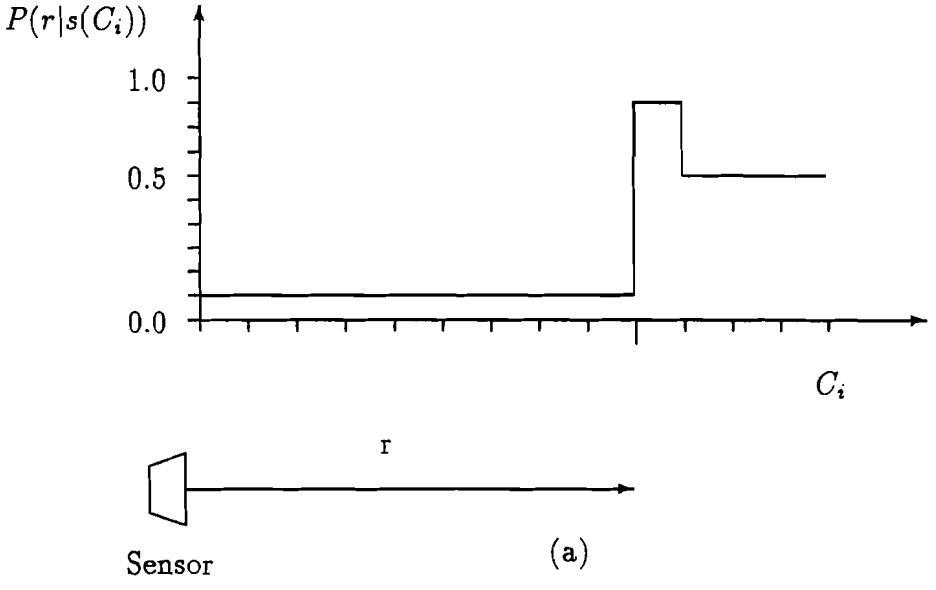
In the case of conflicts but the input range reading being validated as first order, the sensory model Fig 5 (a) is again used to update the map. But the updating formula used at this time is the replace updating formula (4) because the reading is validated to be first order. This case usually occurs when the map has be contaminated at the previous time. With this updating, our robot can quickly recover its map accurately.

5.3.3 Conflicts; Weak Updating

The last case is that there are conflicts between the incoming sensory data and the current map, but the input range reading being not first order. The weak sensory model Fig 5 (b) is therefore used to update the map by updating formula (1). This updating may affect the map slightly.

5.4 Navigator

The Navigator is composed of two subprocesses, obstacle avoidance and local path planning. As the robot is in motion, the navigator check whether there are obstacles blocking ahead. In case that there are no obstacles within



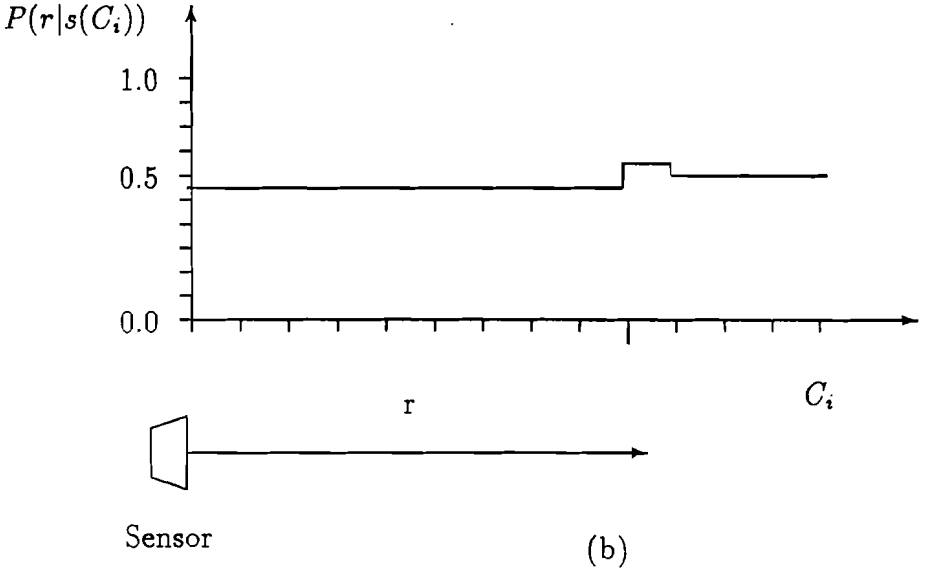


Figure 5: (a) First order sensory model for first order range reading. (b) Weak sensory model for other lower order range reading.

a distance threshold ahead, the robot will trace a closer surface of objects within a threshold range at either sides. The similar triangles formula is used to get the amending degrees to keep the robot parallel to the surface. Fig 6 shows how to compute the amending degrees by the use of two side sensors.

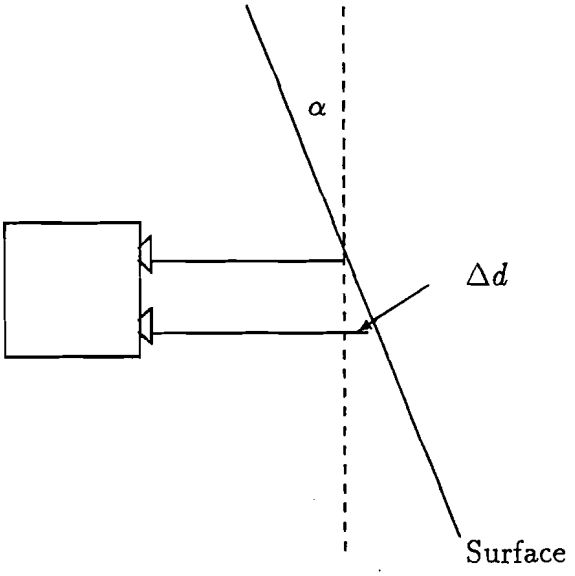


Figure 6: Given a robot width w and the difference of the range reading of the two side sensors d, we can figure out that the amending degrees $\alpha = atan(\frac{\Delta d}{w})$. , where w is the width of our robot.

If there is some obstacle detected ahead by any of the front four sensors, it then classified its surrounding according to the list of obstacle classification in Fig 7. With this recognition, our robot takes the proper action:

(a) In case of obstacles blocking in the front and the corresponding cells in map is occupied, the robot then checks both sides to find a clear path and turns to the path that show more room. If there are no clear path at both sides, the robot turns back.

(b) In case of obstacles blocking in the front but the corresponding cells in map is not occupied (either unknown or empty), the robot will regard this obstacle as a dynamic object. It then stops and keeps waiting until the object has passed away.

(c) In case that the obstacles is detected in the front-left direction, the robot checks the right side sensors whether there are obstacles in the right. If it is not, the navigator keeps the robot rotating the base clockwise until the sensor #2 does not "see" the obstacle. If obstacles is also detected in the right side (Fig 7 case (e)), the robot turns -135 degree to avoid the obstacles.

(d) In case that the obstacles is detected in the front-right direction, the robot checks the left side sensors whether there are obstacles in the left. If it is not, the navigator keeps the robot rotating the base counter-clockwise until the sensor #3 does not "see" the obstacle. If obstacles is also detected in the left side (Fig 7 case (f)), the robot turns 135 degree to avoid the obstacles.

(g) If sensor #2 and #3 detect obstacles concurrently, the robot will turn back even though there are no obstacles detected by sensor #4 and #5.

5.5 Map Display

This process creates a 80×80 grid window under X-window environment to show the up-to-date map. Each cell of grid consists of 5×5 pixels. 32 grave intensity levels are allocated to represent the probabilistic values in the domain $0.0 \ldots 1.0$. The robot mark (red triangle) moves in the window as the real robot runs in the real world.

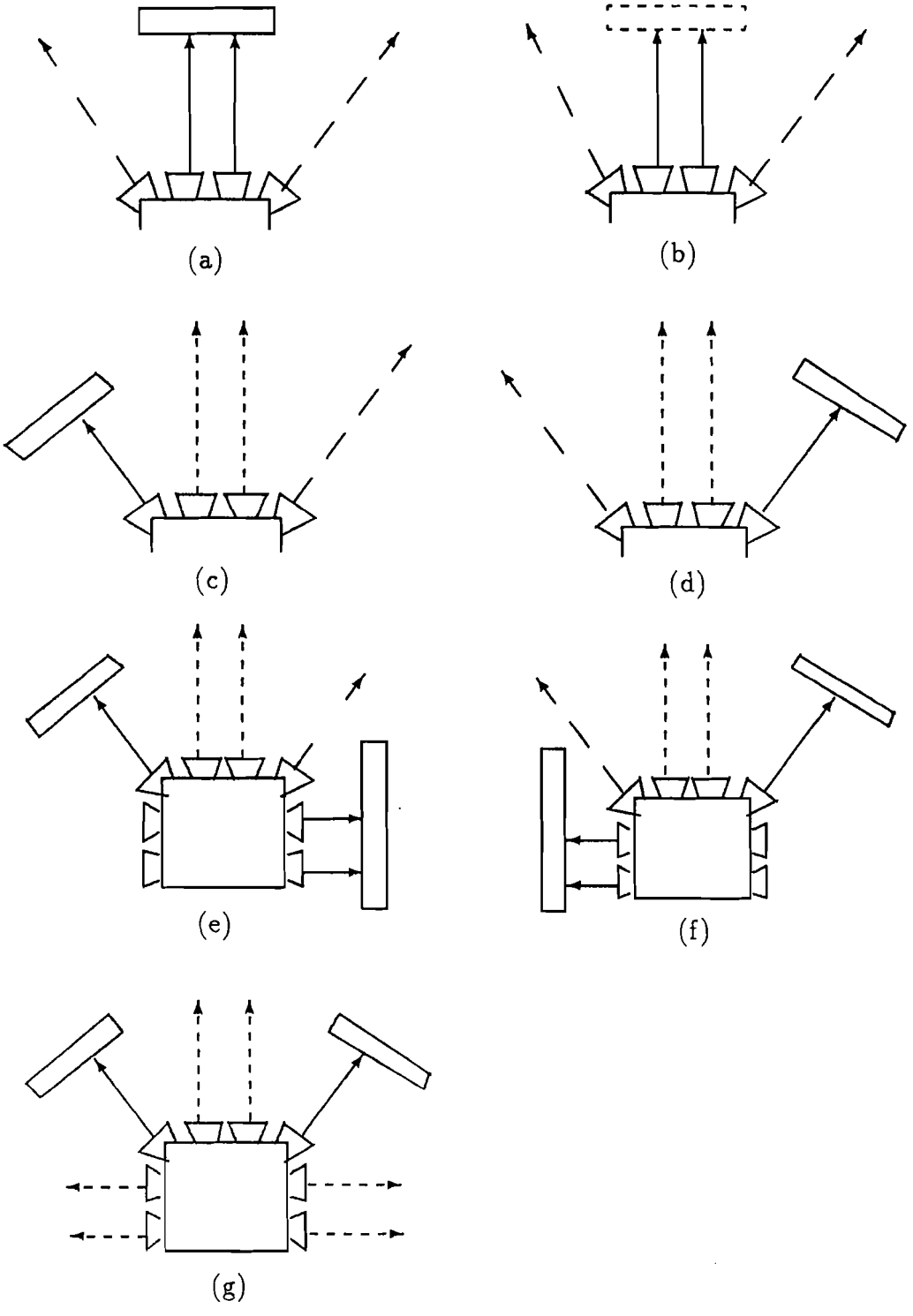


Figure 7: The list of surrounding obstacles classifaction

6 Discussion

The implementation in this paper utilizes Elfes's certainty grid for world representation [1] and Durant-Whyte's approach [3] to extract lower order sensory data for map update. This map is maintained by the sequential updating recurrences with these lower order sensory data. This up-to-date map is then used for high level tasks (obstacle avoidance and path planning).

In our practice experiences, our mobile robot is able to move at high speed (12 cm/sec) in a narrow hallway without being contaminated by high order echo reflections. It can moves forward continuously and select a clear local path to avoid static obstacles. The map has also extended in time dimension to classified dynamic and static obstacles. Our robot can choose proper action with this classification.

But due to the interior uncertainty of sonar sensors and robot's movement, position error will be accumulated incrementally. This position error may blur the map gradually. The Approximate Transformation framework [5] provides a general method to solve this position error. This framework is suggested for further work.

7 Acknowledgment

First of all, I would like to thank Professor Tom Dean for his advice through this this research. Many thanks go to Ken Basye and Jak Kirman for offering their technics and experience in robotics.

References

[1] A. Elfes, *Ph.D. Thesis*, 1989.

[2] Hans P. Moravec, *Certainty Grid for Mobile Robots*, Robotics Institute, Carnegie-Mellow University, 1989.

[3] H. Durrant-Whyte, *A Unified Approach to Mobile-Robot Navigation*, Dept. of Engineering Since, University of Oxford, 1989.

[4] E. Vanmarcke. *Random Fields: Analysis and Synthesis*, MIT press, Cambridge, MA, 1983.

[5] R.C. Smith, M. Self, and P. Cheeseman, *On the representation and Estimation of Spatial Uncertainty*, The international Journal of Robotics Research, 5(4), winter 1986.

A List of Source Codes

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include <math.h>

#include "/pro/ai/robot/software/control/src/robot.h"
#include "/pro/ai/robot/software/control/src/control.h"
#include "/pro/ai/robot/software/control/src/sensor.h"
#include "/pro/ai/robot/include/sonar.h"

typedef enum {FALSE, TRUE} boolean;

/* Estimate model */
#define         VACCANCY_PR       0.1               /* 1st order estimate  */
#define         OCC_PR            0.9
#define         WEAK_OCC_PR       1.0 - 0.5*OCC_PR  /* Weak model estimate */
#define         WEAK_VACCANCY_PR  0.5*(1.0 - VACCANCY_PR)
#define         PR_MIN            1E-5    /* Limitation for empty state    */
#define         PR_MAX            0.99999 /* Limitation for occupancy state*/
#define         MAX_RISK          0.99    /* Limitation for a risk         */

/* Map information */
#define         MAPSIZE           80      /* Map is a 80 x 80 grid       */
#define         GRIDSIZE          200     /* Cell is 200 x 200 cm square */
#define         HALFGRID          (GRIDSIZE >> 1)
#define         CENTER            HALFGRID /* Center of each cell */
#define         MAPBOUND          MAPSIZE * GRIDSIZE /* The bound for a map */
#define         THRESHOLD         360     /* The distance threshold to      */
                                          /* monitor the closest object     */

#define         M_PI_3_4th      (M_PI_2 + M_PI_4) /* Value for three forth of Pi */
#define         RADIAN          (180.0 / M_PI) /* Value for a radian             */

#define         INFINITY        5999     /* Value for infinite distance    */

/* Physical Information of the Robot */
#define         SONAR_NUM       8         /* The number of sonar sensors   */
#define         Speed           12        /* The regular speed of robot    */
#define         RotateSpeed     20        /* The rotation speed for the base */
#define         BEAM_ANGLE      0.26
#define         RRadis          140.0
#define         ROBOTWIDE       240.0     /* Robot's width                 */
```

```
/*******************************************************/
/*          Define some useful macros for use          */
/*******************************************************/
#define ROUND(A)            ( (int) floor((A) + 0.5 ) )
#define SQUARE(A)           ( (A)*(A) )
#define OVER(A)             ( (A) <= MAPBOUND    ? (A) : MAPBOUND )
#define UNDER(A)            ( (A) >= 0           ? (A) : 0 )
#define MAX(A,B,C)          ( (A) >= (B) && (A) >= (C) ? OVER(A) : MAX2(B,C) )
#define MAX2(A,B)           ( (A) >= (B) ? OVER(A) : OVER(B) )
#define MIN(A,B,C)          ( (A) <= (B) && (A) <= (C) ? UNDER(A) : MIN2(B,C) )
#define MIN2(A,B)           ( (A) <= (B) ? UNDER(A) : UNDER(B) )
#define IndexMapping(A)     ( floor((A) / GRIDSIZE ) )

#define RAD(A)              ( ((double) A) / RADIAN )
#define sec(A)              ( 1.0 / cos(RAD(A)) )

#define dominate(V,M,New)  ((V) < 1000 && (New) < ROUND(1.16*(M)) && (New) > ROUND(0.84*(M)
) ? TRUE : FALSE)

#define LIMIT(A)            ((A) > 0.5 ? PR_MAX : PR_MIN)
#define BaysianPr(P1,P2)    ((((P1) >= 0.9) && ((P2) >= PR_MAX)) || (((P1) <= 0.1) && ((P2)
<= PR_MIN)) ? LIMIT(P2) :  (P1)*(P2) / (P1*P2 + (1 - P1)*(1-P2)) )
#define BaysianRisk(P1,P2,D) ( (((P2) - (P1)) > 0.8) && (D) <= 10 ? TRUE : FALSE)
#define PrOnRisk(P)         ( (P) > 0.5 ? (WEAK_OCC_PR) : (WEAK_VACCANCY_PR) )
#define Mean(M,C,D)         (ROUND( ((M)*( (C) - 1 ) + (D)) / (C)))
#define VAR_COM(M,V,X,N,NN) ( ROUND((V) * (N) / (NN) + SQUARE((M) - (X)) / (N)))

#define ObstacleForce(X) ((X) < 0 ? -INFINITY : THRESHOLD - (X) )

/*******************************************************/
/*      Sonar position and orientation information     */
/*******************************************************/

static double Pos_orient[SONAR_NUM] = {
        -M_PI_3_4th,    M_PI_3_4th,     M_PI_4,         -M_PI_4,
        -M_PI_4,        M_PI_4,         -M_PI_4,        M_PI_4};
static double Emit_orient[SONAR_NUM] = {
      -M_PI_2,          M_PI_2,         M_PI_4,         -M_PI_4,
        0.0,            0.0,            -M_PI_2,        M_PI_2};
static int    Sonar_id[SONAR_NUM] = {
        2, 3, 4, 5, 7, 1, 0, 6, };
```

```
/*********************************************************/
/*                    Prototypes                         */
/*********************************************************/
void    Init_Map();              /* Initialization Module */
void    Init_Robot();
void    Draw_initmap();

void    Swap();                  /* Control Module        */
void    DecreaseSpeed();
void    TransferSensorFactor();
int     Stop();
int     Turn_Back();
int     BlockAhead();
int     Find_Alternative();
int     Path_Amend();
int     Slant();
int     Turn_Compute();
void    CompleteTurn();
void    Make_Turn();
int     Walk();

void    Update();                /* Update Module         */
void    Update_Map();
void    Update_Grid();
void    Listen();
boolean PassFilter();

void    Refresh();               /* Map-display Module    */
void    allocateColors();
void    Set_RobotColor();
void    Create_Window();
void    Draw_MapImage();
void    Draw_GridImage();
void    Draw_Robot();
```

```
/*************************************************************************/
/*              Global  Variable for Main Program                        */
/*=======================================================================*/
/*  MAP: <usage> for maintaining a certainty grid                        */
/*       <range> 1.0 - 0.0                                               */
/*  SONARFACTOR:                                                         */
/*       <usage> to store statistic factors for each sonar,              */
/*               they are [mean,smaple total number, variance]           */
/*************************************************************************/

double  MAP[MAPSIZE][MAPSIZE];
double  SONAR_VAR = 50.0;
int     SONARFACTOR[SONAR_NUM][3] = {
                        {0,0,0}, {0,0,0}, {0,0,0}, {0,0,0},
                        {0,0,0}, {0,0,0}, {0,0,0}, {0,0,0} };
```

```
/*************************************************************************/
/* This program implement a robot navigator. A certainty grid is used to */
/* represent the space and objects. There are four modules in this       */
/* program:                                                              */
/*    (1) Control module:                                                */
/*        This module involves an obstacle avoidance and clear-path      */
/*        choosing.                                                      */
/*    (2) Display module:                                                */
/*        This module shows the up-to-date map via X-window, and also    */
/*        locates the current robot.                                     */
/*    (3) Initial module:                                                */
/*        This module initializes the map and all filter factors.        */
/*    (4) Update module:                                                 */
/*        This module updates the map at each scan.                      */
/*                                                  Author : Tu-Hsin Tsai */
/*************************************************************************/
#include <stdio.h>
#include <certainty_grid.h>
#include <mainvar.h>

static jmp_buf r_env;

/*************************************************************************/
/* robot_int                                                             */
/*                                                                       */
/* This function abort the program when users type in crl-C. The map     */
/* will stay on the screen until users type in a character.              */
/*************************************************************************/
robot_int(sig,code,scp,addr)
int sig,code;
struct sigcontex *scp;
char *addr;
{int c;

 set_velocity(0);              /* Stop the robot                         */
 set_slew(0);                  /* Stop the robot's base                  */
 flush_serial_line(BASE_LINE);
 c = getchar();                /* Get a character from standart input    */
 exit(1);                      /* Quit the program                       */
}
```

```
main()
{
 int i,j;
 double cx = 7900.0,              /* Initial position for the robot        */
        cy = 9900.0;
 int orient;
 int pre_distance,
     distance;

/* Create a window for display current certainty grid */
 Create_Window();
 allocateColors();
 Set_RobotColor();

/* Initialization */
 signal(SIGINT,robot_int);
 Init_Robot();
 Init_Map();
 orient = 90;
 pre_distance = query_distance();
 Draw_initmap(cx,cy);

 set_velocity(Speed);

 while(TRUE)
 {    /* Make a scan and update the map       */
      Listen(orient,cx,cy);

      /* Moniter and control the moving robot */
      orient -= Walk(&cx,&cy,&pre_distance,orient);

      /* To trim the degree into the domain (0,360) */
      if (orient > 360)
         orient -= 360;
      else if (orient < 0)
               orient += 360;
 }
}
```

```
#include <certainty_grid.h>

extern  double  MAP[MAPSIZE][MAPSIZE];
extern  int     SONARFACTOR[SONAR_NUM][3];


/*************************************************************/
/* Init_Map                                                  */
/*                                                           */
/* This routine sets all cells of the map to be unknow state. */
/* **********************************************************/
void
Init_Map()
{
 int i,j;

 for (i=0; i < MAPSIZE ; i++)
     for (j=0; j < MAPSIZE ; j++)
        MAP[i][j] = 0.5;
}


/*************************************************************/
/* Init_Robot                                                */
/*                                                           */
/* This routine initializes the sensors and robot's base.    */
/* **********************************************************/
void
Init_Robot()
{
 base_init();
 start_sonar();
}


/*************************************************************/
/* Draw_initmap                                              */
/*                                                           */
/* This routine draws the initial map.                       */
/* **********************************************************/
void
Draw_initmap(x,y)
double x,y;
{
 Draw_MapImage(MAP);
 Draw_Robot(x,y,M_PI_2);
}
```

```
/*************************************************************************/
/*  These procedures monitors the robot in motion and control it to     */
/*  avoid the obstacle along its path.                                   */
/*************************************************************************/

#include <certainty_grid.h>

typedef enum {NO_BLOCK,DYNAMIC_OBSTACLE,STATIC_OBSTACLE,FRONT_RIGHT_BLOCK,
              FRONT_LEFT_BLOCK} blockingType;

#define LEFTSIDE         1
#define RIGHTSIDE        -1
#define FRONT_LEFT_DEG   -45
#define FRONT_RIGHT_DEG  45
#define DANGERTHRESHOLD  300
#define VARIATION        100
#define BACK_DEG         179
#define SLIPPERYFACTOR   10
#define SlewRate         1

#define DEG(X)           ( ROUND(RADIAN * (X)) )
#define Block(P)         ((P) > 0.9 ? TRUE : FALSE)
#define Degree_Amend(X,S) ((X) > 64 ? (S) * 15 :  (S) * DEG(atan((X) / ROBOTWIDE) ))
#define ApproachForce(F,B) ( (F) < 0 && (B) < 0 ? INFINITY : (F + B) >> 1 )
#define BlockForce(F1,F2) ( (F1 <= 0) || (F2 <= 0) ? 0 : ((F1) + (F2)) )
#define OutSideBoundary(X,Y,A) ( ((X) > MAPSIZE && cos(A) > 0 ) || ((Y) > MAPSIZE && sin(A
) > 0 ) || ((X) < 0  && cos(A) < 0) || ((Y) < 0 && sin(A) < 0) ? TRUE :FALSE )
#define AvoidTooClose(F,S) ((F) > DANGERTHRESHOLD ? (S) * ((F) >> 6) : 0)
#define SetDivision(X) ((int) (X/M_PI_4 + 0.5))

extern  double  MAP[MAPSIZE][MAPSIZE];
extern  int     SONARFACTOR[SONAR_NUM][3];

static int      Sonar_Data[SONAR_NUM];
int             Leftforce,
                Rightforce;

/* The distance Threshold for the four front sensors */
int             BlockThreshold[4] = {-70, -70, -10, -10};
```

```
/*************************************************************************/
/* Swap                                                                  */
/*                                                                       */
/* This routine swaps the given two parameters                           */
/*************************************************************************/
void
Swap(arg1, arg2)
int * arg1, * arg2;
{  int tmp;

   tmp = * arg1;
   * arg1 = * arg2;
   *arg2 = tmp;
}


/*************************************************************************/
/* DecreaseSpeed                                                         */
/*                                                                       */
/* This routine decelerates the robot by 3 cm/sec at each time so that   */
/* the robot's speed drop down smoothly.                                 */
/*************************************************************************/
void
DecreaseSpeed(speed,decrease)
int * speed;
int decrease;
{
   if (* speed > decrease)
      * speed -= decrease;
   else * speed = 0;
   set_velocity(* speed);
}
```

```
/*************************************************************************/
/* TransferSensorFactor                                                  */
/*                                                                       */
/* This routine inherits the sensory factors to approper sensors         */
/* when the robot make a turn.                                           */
/*************************************************************************/
void
TransferSensorFactor(orientation)
int orientation;
{  int i;

   switch (orientation) {
   case -2:
   case 2:  for (i=0;i<3;i++)
            {    Swap(&SONARFACTOR[6][i],&SONARFACTOR[1][i]);
                 Swap(&SONARFACTOR[7][i],&SONARFACTOR[0][i]);
            }
            break;
   case 1:  for (i=0;i<3;i++)
            {    SONARFACTOR[1][i] = SONARFACTOR[5][i];
                 SONARFACTOR[7][i] = SONARFACTOR[4][i];
                 SONARFACTOR[4][i] = SONARFACTOR[0][i];
                 SONARFACTOR[5][i] = SONARFACTOR[6][i];
            }
            break;
   case -1: for (i=0;i<3;i++)
            {    SONARFACTOR[6][i] = SONARFACTOR[5][i];
                 SONARFACTOR[0][i] = SONARFACTOR[4][i];
                 SONARFACTOR[4][i] = SONARFACTOR[1][i];
                 SONARFACTOR[5][i] = SONARFACTOR[7][i];
            }
            break;
   default:
            break;
   }
}
```

```
/*************************************************************************/
/* Stop                                                                  */
/*                                                                       */
/* This routine stops the robot for at most 8 seconds until the block    */
/* ahead is removed, and find a alternative clear path if the waiting    */
/* time is over 60 sec.                                                  */
/*************************************************************************/
int
Stop()
{
 int waitime;
 int timethreshold = 8;
 int sonar_data[SONAR_NUM];

 set_velocity(0);
 for (waitime = 0; waitime <= timethreshold; waitime++)
 {      sleep(1);
        read_sonar( sonar_data );
        if ( BlockForce(ObstacleForce(sonar_data[4]),
                        ObstacleForce(sonar_data[5]) ) <= 0)
                break;
 }
 if (waitime >= timethreshold) /* Obstacle ahead is clear now */
     return( Find_Alternative() );

 /* To check whether there is anything still blocking in front left of
    front right. If there is, make a turn to avoid it in advance       */
 if (ObstacleForce(sonar_data[3] > 0))
     return(FRONT_LEFT_DEG);
 else if (ObstacleForce(sonar_data[2] > 0))
          return(FRONT_RIGHT_DEG);
      else return(0);
}


/*************************************************************************/
/* Turn_Back                                                             */
/*                                                                       */
/* To make the robot turn back. Check both sides before turning in       */
/* order to avoid it hit the object beside during it turns               */
/*************************************************************************/
int
Turn_Back()
{
 int back_degree;

 back_degree = (Leftforce > Rightforce) ? BACK_DEG : -BACK_DEG;
 Make_Turn(back_degree);
 return(back_degree);
}
```

```
/*************************************************************************/
/* BlockAhead                                                            */
/*                                                                       */
/* To check whether there is any obstacle blocking ahead. If there is,   */
/* it determines the obstacles's blocking direction and type, and        */
/* it.                                                                   */
/*************************************************************************/
int
BlockAhead(x,y,direction)
int x,y;
double direction;
{
 int mx[4],my[4];
 short divisions[4];
 blockingType type = NO_BLOCK;
 short i,j;

 mx[0] = mx[1] = mx[2] = mx[3] = x;
 my[0] = my[1] = my[2] = my[3] = y;
 divisions[0] = divisions[1] = divisions[2] = divisions[3]
              = SetDivision(direction);
 divisions[1] = (divisions[2] < 8) ? divisions[2]+1 : 8;
 divisions[0] = (divisions[3] > 0) ? divisions[3]-1 : 8;

 read_sonar(Sonar_Data);

 /* to check the four front sensors whether there is any obstacle
    blocking ahead.                                                  */
 for (i=3;i>=0 && !type;i--)
 {    if (ObstacleForce(Sonar_Data[Sonar_id[i]]) > BlockThreshold[i])
      {   type = DYNAMIC_OBSTACLE;
          if (i<2)
          {   i--;
              break;
          }

          /* To check the corresponding cells whether they are occpupied. */
          /* If they are, the ostacles is static. Otherwise, it is dynamic*/
          switch (divisions[i]) {
          case 0:
          case 8: mx[0] += 2;  mx[2] = mx[0];
                  mx[1] += 3;  mx[3] = mx[1];
                  my[2]--;     my[3]--;
                  break;
          case 1: mx[0]++;  mx[2] = mx[0];
                  mx[1] += 2;  mx[3] = mx[1];
                  my[0] ++;    my[2] = my[1];
                  my[1] += 2;  my[3] = my[0];
                  break;
          case 2: my[0] += 2;  my[2] = my[0];
                  my[1] += 3;  my[3] = my[1];
                  mx[2]++;     mx[3]++;
                  break;
          case 3: mx[0]--;     mx[2] = mx[0];
                  mx[1] -= 2;  mx[3] = mx[1];
                  my[0]++;     my[2] = my[1];
                  my[1] += 2;  my[3] = my[0];
                  break;
          case 4: mx[0] -= 2;  mx[2] = mx[0];
                  mx[1] -= 3;  mx[3] = mx[1];
                  my[2]++;     my[3]++;
                  break;
          case 5: mx[0]--;     mx[2] = mx[0];
```

```
                mx[1] -= 2;  mx[3] = mx[1];
                my[0]--;     my[2] = my[1];
                my[1] -= 2;  my[3] = my[0];
                break;
        case 6: my[0] -= 2;  my[2] = my[0];
                my[1] -= 3;  my[3] = my[1];
                mx[2]++;     mx[3]++;
                break;
        case 7: mx[0]++;     mx[2] = mx[0];
                mx[1] += 2;  mx[3] = mx[1];
                my[0]--;     my[2] = my[1];
                my[1] -= 2;  my[3] = my[0];
                break;
        default: break;
        }
        type = Block(MAP[mx[0]][my[0]]) || Block(MAP[mx[1]][my[1]]) ||
               Block(MAP[mx[2]][my[2]]) || Block(MAP[mx[3]][my[3]])
               ? STATIC_OBSTACLE
               : type;
    }
 }
 if ((i<1) && type) /* Blocking ahead on either 45 or 135 degree direction */
     type = 3 - i;
 return(type);
}
```

```
/*************************************************************************/
/* Find_Alternative                                                      */
/*                                                                       */
/* This routine check the robot's sides to find a clear path at left     */
/* or right side. In case that there is no path at left or right sides,  */
/* the robot chose to turn back.                                         */
/*************************************************************************/
int
Find_Alternative()
{
 int degree = 180;
 int leftapproachforce,
     rightapproachforce;

 leftapproachforce = ApproachForce(Sonar_Data[7],Sonar_Data[1]);
 rightapproachforce = ApproachForce(Sonar_Data[6],Sonar_Data[0]);
 if (leftapproachforce || rightapproachforce)
        degree = (leftapproachforce > rightapproachforce) ? -90 : 90;
 return(degree);
}


/*************************************************************************/
/* Path_Amend                                                            */
/*                                                                       */
/* This routine computes the degree amendment along the path via the     */
/* side sensors in a monitoring range.                                   */
/*************************************************************************/
int
Path_Amend()
{
 int amend_degree = 0;

 /* To check the left side where there is a object within the monitoring  */
 /* threshold                                                             */
 Leftforce = BlockForce(ObstacleForce(Sonar_Data[7]),
                        ObstacleForce(Sonar_Data[1]) );

 /* To check the right side where there is a object within the monitoring */
 /* threshold                                                             */
 Rightforce = BlockForce(ObstacleForce(Sonar_Data[6]),
                         ObstacleForce(Sonar_Data[0]) );

 if (Leftforce || Rightforce) /* There is a object near either sides */
 {   amend_degree = (Leftforce > Rightforce)
                  ? Degree_Amend(Sonar_Data[1] - Sonar_Data[7],LEFTSIDE)
                  : Degree_Amend(Sonar_Data[0] - Sonar_Data[6],RIGHTSIDE);

     /* To avoid to be close to the object beside */
     amend_degree += (Leftforce > Rightforce)
                   ? AvoidTooClose(Leftforce,LEFTSIDE)
                   : AvoidTooClose(Rightforce,RIGHTSIDE);
   }
 return(amend_degree);
}
```

```
/*************************************************************************/
/* Slant                                                                 */
/*                                                                       */
/* This routine makes the robot avoid the obstacle in front left or      */
/* front right directions.                                               */
/*************************************************************************/
int
Slant(slantDegree,sonar_id)
int slantDegree;
int sonar_id;
{
 int currentOrientation,turn_degree = 0;
 int speed = Speed;
 int decrease = 4;

 /* To rotate the robot until it cannot see the obstacle via the given  */
 /* sensor.                                                             */
 while (ObstacleForce(Sonar_Data[sonar_id]) > -50)
 {   execute_turn(slantDegree,RotateSpeed);
     DecreaseSpeed(&speed,decrease);
     decrease++;
     turn_degree += slantDegree;
     read_sonar(Sonar_Data);
  }
 set_velocity(Speed);
 return(turn_degree);
}
```

```
/*************************************************************************/
/* Turn_Compute                                                          */
/*                                                                       */
/* This routine computes the degrees that the robot should turn based    */
/* on the blocking type returned by the routine BlockAhead.              */
/*************************************************************************/
int
Turn_Compute(x,y,direction)
int x,y;
double direction;
{
 int choice;
 int turn_degree = 0;
 int dir;

 choice = BlockAhead(x,y,direction);
 switch (choice) {
 case NO_BLOCK:       /* No block ahead                                */
        turn_degree = Path_Amend();
        if (turn_degree != 0)
            execute_turn(turn_degree,Speed);
        break;
 case DYNAMIC_OBSTACLE: /* Dynamic object blocks ahead, keep waiting */
        turn_degree = Stop();
        if (turn_degree != 0)
            Make_Turn(turn_degree);
        else set_velocity(Speed);
        break;
 case STATIC_OBSTACLE:  /* Static object blocks ahead, turn back     */
        turn_degree = Find_Alternative();
        Make_Turn(turn_degree);
        break;
 case FRONT_RIGHT_BLOCK: /* Objects block on 45 degree direction  */
        if (ObstacleForce(Sonar_Data[1]) > 0 &&
            ObstacleForce(Sonar_Data[7]) > 0)
        {   turn_degree = 135;
            Make_Turn(turn_degree);
            turn_degree += Slant(12,2);
        }
        else if (ObstacleForce(Sonar_Data[2]) > 0)
                 turn_degree = Turn_Back();
             else turn_degree = Slant(-12,3);
        break;
 case FRONT_LEFT_BLOCK: /* Objects block on 135 degree direction  */
        if (ObstacleForce(Sonar_Data[0]) > 0 &&
            ObstacleForce(Sonar_Data[6]) > 0)
        {   turn_degree = -135;
            Make_Turn(turn_degree);
            turn_degree += Slant(-12,3);
        }
        else if (ObstacleForce(Sonar_Data[3]) > 0)
                 turn_degree =  Turn_Back();
             else turn_degree = Slant(12,2);
        break;
 }
 return(turn_degree);
}
```

```
/*************************************************************************/
/* CompleteTurn                                                          */
/*                                                                       */
/* This routine makes sure the robot has completed the turning           */
/*************************************************************************/
void
CompleteTurn(angle,time_unit,orientation)
int angle;
int time_unit;
int orientation;
{
   int current_angle;

   TransferSensorFactor(orientation);
   sleep(time_unit);
   while ((current_angle = query_angle()) != angle)
   { angle = current_angle;
     sleep(time_unit);
    }
}

/*************************************************************************/
/* MakeTurn                                                              */
/*                                                                       */
/* This routine halts the robot and executes a turn with the given       */
/* degree.                                                               */
/*************************************************************************/
void
Make_Turn(degree)
int degree;
{
    int time;

    set_velocity(0);
    set_turn(degree);
    CompleteTurn(query_angle(),SlewRate,ROUND(degree/90));
    set_velocity(Speed);
/*    if (abs(degree) >= BACK_DEG)
    {  time = 3;
       sleep(time);
     } */
}
```

```
/*************************************************************************/
/* Walk                                                                  */
/*                                                                       */
/* This routine is the main routine to monitor the environment feedback */
/* and find a clear path to avoid obstacles                              */
/*************************************************************************/
int
Walk(cx,cy,old_distance,direction)
double *cx, *cy;
int *old_distance;
int direction;
{
 int x,y;
 int i;
 int distance;
 double dir;
 int Theta;

 dir = RAD(direction);
 distance = (int) SLIPPERYFACTOR * (query_distance() - *old_distance);
 x = (int) IndexMapping( *cx + distance * cos( dir ) );
 y = (int) IndexMapping( *cy + distance * sin( dir ) );

 /* If the robot walks outside the bondering, make it back.     */
 /* Otherwise, amend its path if necessary.                     */
 Theta = ( OutSideBoundary(x,y,dir) )
               ? Turn_Back()
               : Turn_Compute(x,y,dir);
 distance = query_distance() - *old_distance;

 /* To correct the position of robot */
 *old_distance += distance;
 distance = (int) SLIPPERYFACTOR * distance;
 *cx +=  distance * cos(dir);
 *cy +=  distance * sin(dir);

 /* To adjust the sensory factor by the current motion */
 SONARFACTOR[4][0] -= distance;
 SONARFACTOR[5][0] -= distance;
 SONARFACTOR[2][0] -= ROUND(M_SQRT2*distance);
 SONARFACTOR[3][0] -= ROUND(M_SQRT2*distance);

 for (i=4; i<=7 ; i++)
      PassFilter(Sonar_id[i],Sonar_Data[Sonar_id[i]]);

 return(Theta);
}
```

```
/*************************************************************/
/* Display Module:                                           */
/*    This module displays the up-to-date map via X-window   */
/*************************************************************/

/* Xlib include file */
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <X11/Xos.h>
#include <math.h>

#include <stdio.h>

#include <certainty_grid.h>

#define maxCoord        80
#define DEFAULT_WIDTH   400
#define DEFAULT_HEIGHT  400
#define borderw         4
#define NUM_COLORS      32
#define pixelsPerVertex 5
#define BASE            2048

/*************************************************************/
/*              Define  Macro                                */
/*************************************************************/
#define SelectColor(A)   (floor((A) * (NUM_COLORS - 1)))

/*************************************************************/
/*              Local   Variable                             */
/*************************************************************/
char    * display_name = NULL;
Display *display;
Window  win;
int     screen;
char    * window_name = "Certainty Grid";
char    * icon_name   = "basicwin";
XSizeHints size_hints;
Pixmap  icon_pixmap;
GC      gc;
XGCValues gcv;
XEvent report;

Colormap xcolormap;
unsigned long colorArray[ NUM_COLORS ];
XColor robotcolor;

XPoint pt[3];

extern  double  MAP[MAPSIZE][MAPSIZE];
```

```
/*******************************************************/
/* allocateColors                                      */
/*                                                     */
/*  This routine allocates a color table for use       */
/*******************************************************/
void allocateColors()
{
    Status status;
    unsigned long plane_masks;
    XColor colorTemplate;
    unsigned int index,pvalue;


    status = XAllocColorCells (display, xcolormap, 0,
                                &plane_masks, 0, colorArray, NUM_COLORS);

    colorTemplate.flags = DoRed | DoGreen | DoBlue;

    for ( index = 0; index < NUM_COLORS; index++ )
    {   pvalue = (NUM_COLORS - index) * BASE - 1;
        colorTemplate.pixel = colorArray[ index ];
        colorTemplate.red = pvalue;
        colorTemplate.green = pvalue;
        colorTemplate.blue = pvalue;

        XStoreColor (display, xcolormap, colorTemplate);
            }
}


/*******************************************************/
/* Set_RobotColor                                      */
/*                                                     */
/*  This routine set the robot's color to be red.      */
/*******************************************************/
void
Set_RobotColor()
{
 robotcolor.red = 45000;
 robotcolor.green = 65536;
 robotcolor.blue = 65536;

 if ( !XAllocColor( display, xcolormap, &robotcolor) )
    exit(-1);
}


/*********************************************************************/
/* Refresh                                                           */
/*                                                                   */
/* This routine refresh teh screen to show the current image         */
/*********************************************************************/
void
Refresh()
{
 XFlush(display,screen);
}
```

```
/*******************************************************/
/* Create_Window                                       */
/*                                                     */
/*  This routine create a 400 x 400 pixels window to   */
/*  show up-to-date map.                               */
/*******************************************************/
void
Create_Window()
{
 unsigned int width, height;
 int    x = 0, y = 0;
 int    ac = 1;
 char   * av[4];

 if ((display=XOpenDisplay(display_name)) == NULL)
 {      printf("basicwin: cannot connect to X server \n");
        exit( -1 );
 }

 screen = DefaultScreen(display);
 width = DEFAULT_WIDTH;
 height = DEFAULT_HEIGHT;
 win = XCreateSimpleWindow(display, RootWindow(display,screen),
                           x, y, width, height, borderw,
                           BlackPixel(display,screen),
                           WhitePixel(display,screen));

 size_hints.flags = PPosition | PSize;
 size_hints.x = x;
 size_hints.y = y;
 size_hints.width = width;
 size_hints.height = height;
 av[0] = "y";
 av[1] = 0;
 XSetStandardProperties(display, win,window_name,icon_name,
                        icon_pixmap,av,ac,&size_hints);
 XSelectInput(display, win, ExposureMask | StructureNotifyMask );
 gc = XCreateGC( display, win, 0, &gcv);
 XSetForeground(display, gc, BlackPixel(display,screen));
 XMapWindow(display,win);
 XFlush(display,screen);

 XNextEvent(display,&report);
 xcolormap = DefaultColormap( display, DefaultScreen( display ));
}
```

```
/*******************************************************/
/* Draw_MapImage                                       */
/*                                                     */
/*  This routine draws the whole grid map              */
/*******************************************************/
void
Draw_MapImage(grid)
double  grid[maxCoord][maxCoord];
{
 register int x,y;
 int color;

 for (x=0; x < maxCoord; x++)
   for ( y = 0; y < maxCoord; y++ )
   {   color = SelectColor(grid[x][y]);
       XSetForeground( display, gc, colorArray[color] );
       XFillRectangle( display, win, gc,
                       x*pixelsPerVertex, y*pixelsPerVertex,
                       pixelsPerVertex, pixelsPerVertex);
     }
 XFlush(display,screen);
}


/*****************************************************************/
/* Draw_GridImage                                                */
/*                                                               */
/* This routine draws a corresponding gray color to the specified cell  */
/*****************************************************************/
void
Draw_GridImage(x,y,pr)
int x,y;
double pr;
{
 int color;

  color = SelectColor(pr);
  y = 79 - y;
  XSetForeground( display, gc, colorArray[color] );
  XFillRectangle( display, win, gc,
                  x*pixelsPerVertex, y*pixelsPerVertex,
                  pixelsPerVertex, pixelsPerVertex);
}
```

```
/*************************************************************************/
/* Draw_Robot                                                            */
/*                                                                       */
/* This routine draws the robot in the current position                  */
/*************************************************************************/
void
Draw_Robot(dx,dy,orient)
double dx,dy;
double orient;
{
  int l = 6;
  double dis_angle = 2.617993878;
  int x,y;
  static int ix,iy;

  /* Clear the robot shape */
  for (x=ix-1;x<=ix+1;x++)
      for (y=iy-1;y<=iy+1;y++)
          Draw_GridImage(x,y,MAP[x][y]);

  ix = ROUND( IndexMapping(dx) );
  iy = ROUND( IndexMapping(dy) );

  /* redraw the robot in the new position */
  x = 3 + ix * pixelsPerVertex;
  y = 3 + iy * pixelsPerVertex;
  pt[0].x = x + l*cos(orient);
  pt[0].y = DEFAULT_HEIGHT - y - (int) l*sin(orient);
  pt[1].x = x + (int) l*cos(orient + dis_angle);
  pt[1].y = DEFAULT_HEIGHT - y - (int) l*sin(orient + dis_angle);
  pt[2].x = x + (int) l*cos(orient - dis_angle);
  pt[2].y = DEFAULT_HEIGHT - y - (int) l*sin(orient - dis_angle);
  XSetForeground( display, gc, robotcolor.pixel );
  XFillPolygon( display, win, gc, pt, 3, Convex, CoordModeOrigin);
  XFlush(display,screen);
}
```

```
/*************************************************************************/
/* Update Module:                                                        */
/*                                                                       */
/*  To make a scan and update the global grid.                           */
/*************************************************************************/

#include <certainty_grid.h>

#define DELTA_DETERMINE(D) ( abs(cos(D)) > abs(sin(D)) ? abs(1.0/cos(D)) : abs(1.0/sin(D))
 )
#define CompareDistance(D,DS,DL) ( (D) >= (DS) && (D) <= (DL) ? TRUE : FALSE)
#define definetype(R,B) ( (R << 1) + B )

extern  double  SONAR_VAR;
extern  double  MAP[MAPSIZE][MAPSIZE];
extern  int     SONARFACTOR[SONAR_NUM][3];



/*************************************************************************/
/* PassFilter                                                            */
/*                                                                       */
/* This routine filters out the noises that are in narrow RCD or         */
/* the reading less than 0.                                              */
/*************************************************************************/
boolean
PassFilter(id,reading)
int id;
int reading;
{
 if (reading < 0)
    return(FALSE);
 if (SONARFACTOR[id][1] != 0 && dominate(SONARFACTOR[id][2],
                                        SONARFACTOR[id][0],reading))
 {  SONARFACTOR[id][1]++;
    SONARFACTOR[id][0] = Mean(SONARFACTOR[id][0],SONARFACTOR[id][1],reading);
    SONARFACTOR[id][2] = VAR_COM(SONARFACTOR[id][0],
                                 SONARFACTOR[id][2],
                                 reading,
                                 SONARFACTOR[id][1] - 1,
                                 SONARFACTOR[id][1]);
    if (SONARFACTOR[id][1] >= 6)
        return(1);
 }
 else
 {  SONARFACTOR[id][0] = reading;
    SONARFACTOR[id][1] = 1;
    SONARFACTOR[id][2] = 0;
 }
 return(FALSE);
}
```

```
/*************************************************************************/
/* Listen                                                                */
/*                                                                       */
/*  This routine makes a scan and feeds in the sonsery data to routine   */
/*  Update. In order to avoid coner, this routine checks it and          */
/*  interrupts the update module if there is a coner beside.             */
/*************************************************************************/
void
Listen(orient,cur_cx,cur_cy)
int orient;
double cur_cx,cur_cy;
{
 double  cur_orient;
 int     sonar_data[SONAR_NUM];
 int     j,id1,id2;

 cur_orient = RAD(orient);
 read_sonar(sonar_data);

 /* To check the coner beside */
 if (ObstacleForce(sonar_data[2]) >= -40 ||
     ObstacleForce(sonar_data[3]) >= -40)
    return; /* if there is a coner, interrupt immediately */

 Update(2,2,sonar_data[2],sonar_data[2],cur_cx,cur_cy,cur_orient);
 Update(3,3,sonar_data[3],sonar_data[3],cur_cx,cur_cy,cur_orient);

/* update with the rest sonar readings */
 for (j=2; j<=7; j++)
 {   id1 = Sonar_id[j];
     j++;
     id2 = Sonar_id[j];
     Update(id1,id2,sonar_data[id1],sonar_data[id2],
            cur_cx,cur_cy,cur_orient);
     Update(id2,id1,sonar_data[id2],sonar_data[id1],
            cur_cx,cur_cy,cur_orient);
  }
  Draw_Robot(cur_cx,cur_cy,cur_orient);
}
```

```
/*************************************************************************/
/*  Update                                                               */
/*                                                                       */
/*  This routine updates the map via the data given by sensor id1        */
/*************************************************************************/
void
Update(id1,id2,sonar_data1,sonar_data2,x,y,orientation)
int id1,id2;
int sonar_data1,sonar_data2;
double x,y;
double orientation;
{
 double theta;
 double  cur_px,cur_py;

 if ( PassFilter(id1,sonar_data1) )    /* To filter out improper data */
 {       theta = Pos_orient[id1] + orientation;
         cur_px = x + RRadis * cos(theta);
         cur_py = y + RRadis * sin(theta);
         Update_Map(id1,cur_px,cur_py,Emit_orient[id1]+orientation,
                                      sonar_data1);
 }
}
```

```
/*************************************************************************/
/* Update_Map                                                            */
/*                                                                       */
/*  This routine updates the map. To make the updating efficiently,      */
/*  this routine uses a specificied algorithm called midpoint line       */
/*  scan-conversion algorithm. With less multiple and square root        */
/*  computation, this routine can update the map efficiently.            */
/*************************************************************************/
void
Update_Map(sonar_index,cur_x,cur_y,orient_angle,range_reading)
int sonar_index;
double cur_x,cur_y;
double orient_angle;
int range_reading;
{
 int x,y;
 short i,j,xunit,yunit;
 short end,grid_size,range;
 double px,py;
 double m,abs_m;
 double d,incrD,incrAD;
 double shortDistanceSquare,
        longDistanceSquare;

 m = tan(orient_angle);
 x = (int) IndexMapping( cur_x );
 y = (int) IndexMapping( cur_y );

 if ((abs_m = fabs(m)) < 1)
 {   d = 2*abs_m;
     incrD = 2*abs_m;
     incrAD = 2*(abs_m - 1);
     shortDistanceSquare
       = SQUARE(range_reading - HALFGRID / fabs(cos(orient_angle)) );
     longDistanceSquare
       = SQUARE(range_reading + HALFGRID / fabs(cos(orient_angle)) );
     end = (int) IndexMapping(range_reading * cos(orient_angle));
     if (sin(orient_angle) > 0)
     {  yunit = 1;
        grid_size = GRIDSIZE;
     }
     else
     {  yunit = -1;
        grid_size = -GRIDSIZE;
     }
     Update_Grid(sonar_index,x,y,range_reading,FALSE);
     if (end > 0)
     {  for (i=1,x++;i < (end - 1);i++,x++)
        {   if (d <= 0)
                d += incrD;
            else
            {   d += incrAD;
                y += yunit;
             }
            range = i >> 4;
            for (j = -range; j <= range; j++)
                 Update_Grid(sonar_index,x,y+j,range_reading,FALSE);
         }
        px = x*GRIDSIZE - CENTER;
        py = y*GRIDSIZE + CENTER;
        for (;i <= (end+1); i++, x++)
        {   if (d <= 0)
            {   d += incrD;
```

```
                px += GRIDSIZE;
             }
            else
            {   d += incrAD;
                y += yunit;
                px += GRIDSIZE;
                py += grid_size;
             }
            range = i >> 4;
            for (j = -range; j <= range; j++)
                Update_Grid(sonar_index,x,y+j,range_reading,
                            CompareDistance(SQUARE(px-cur_x) + SQUARE(py-cur_y),
                                            shortDistanceSquare,
                                            longDistanceSquare) );
         }
      }
     else
     {  for (i=-1,x--;i > (end + 1);i--,x--)
        {   if (d >= 0)
                d -= incrD;
            else
            {   d -= incrAD;
                y += yunit;
             }
            range = abs(i) >> 4;
            for (j = -range; j <= range; j++)
                 Update_Grid(sonar_index,x,y+j,range_reading,FALSE);
         }
        px = (x+1)*GRIDSIZE + CENTER;
        py = y*GRIDSIZE + CENTER;
        for (;i >= (end-1); i--, x--)
        {   if (d <= 0)
            {   d -= incrD;
                px -= GRIDSIZE;
             }
            else
            {   d -= incrAD;
                y += yunit;
                px -= GRIDSIZE;
                py += grid_size;
             }
            range = abs(i) >> 4;
            for (j = -range; j <= range; j++)
                Update_Grid(sonar_index,x,y+j,range_reading,
                            CompareDistance(SQUARE(px-cur_x) + SQUARE(py-cur_y),
                                            shortDistanceSquare,
                                            longDistanceSquare) );
         }
      }
  }
 else
 {   d = 1 - 2/abs_m;
     incrD = -2/abs_m;
     incrAD = 2*(1 - 1/abs_m);
     shortDistanceSquare
        = SQUARE(range_reading - HALFGRID / fabs(sin(orient_angle)) );
     longDistanceSquare
        = SQUARE(range_reading + HALFGRID / fabs(sin(orient_angle)) );
     end = (int) IndexMapping(range_reading * sin(orient_angle));
     if (cos(orient_angle) > 0)
     {  xunit = 1;
        grid_size = GRIDSIZE;
     }
     else
```

```
{  xunit = -1;
   grid_size = -GRIDSIZE;
}
Update_Grid(sonar_index,x,y,range_reading,FALSE);
if (end > 0)
{  for (i=1,y++;i < (end - 1);i++,y++)
   {   if (d >= 0)
           d += incrD;
       else
       {   d += incrAD;
           x += xunit;
        }
       range = i >> 4;
       for (j = -range; j <= range; j++)
            Update_Grid(sonar_index,x+j,y,range_reading,FALSE);
    }
   px = x*GRIDSIZE + CENTER;
   py = y*GRIDSIZE - CENTER;
   for (;i <= (end+1); i++, y++)
   {   if (d >= 0)
       {   d += incrD;
           py += GRIDSIZE;
        }
       else
       {   d += incrAD;
           x += xunit;
           px += grid_size;
           py += GRIDSIZE;
        }
       range = i >> 4;
       for (j = -range; j <= range; j++)
           Update_Grid(sonar_index,x+j,y,range_reading,
                       CompareDistance(SQUARE(px-cur_x) + SQUARE(py-cur_y),
                                       shortDistanceSquare,
                                       longDistanceSquare) );
    }
 }
else
{  for (i=-1,y--;i > (end + 1);i--,y--)
   {   if (d <= 0)
           d -= incrD;
       else
       {   d -= incrAD;
           x += xunit;
        }
       range = abs(i) >> 4;
       for (j = -range; j <= range; j++)
            Update_Grid(sonar_index,x+j,y,range_reading,FALSE);
    }
   px = x*GRIDSIZE + CENTER;
   py = (y+1)*GRIDSIZE + CENTER;
   for (;i >= (end-1); i--, y--)
   {   if (d <= 0)
       {   d -= incrD;
           py -= GRIDSIZE;
        }
       else
       {   d -= incrAD;
           x += xunit;
           px += grid_size;
           py -= GRIDSIZE;
        }
       range = abs(i) >> 4;
       for (j = -range; j <= range; j++)
```

```
                Update_Grid(sonar_index,x+j,y,range_reading,
                            CompareDistance(SQUARE(px-cur_x) + SQUARE(py-cur_y),
                                            shortDistanceSquare,
                                            longDistanceSquare) );
            }
        }
    }
}
```

```
/*************************************************************************/
/* Update_Grid                                                           */
/*                                                                       */
/* This routine updates a single grid given by the caller.               */
/* In this routine, Baysian Risk and  sonar's variance are taken into    */
/* account for determining the certainty value.                          */
/*************************************************************************/
void
Update_Grid(id,x,y,range,occupied)
int id;
int x,y;
int range;
int occupied;
{
 double new_pr;
 int risk = 0;
 int type;

 new_pr = (occupied) ? OCC_PR : VACCANCY_PR;
 risk = BaysianRisk(new_pr,MAP[x][y],SONARFACTOR[id][2] / SONARFACTOR[id][1]);
 type = definetype(risk,range > (THRESHOLD << 1));
 switch(type) {
 case 0:
 case 1:
        MAP[x][y] = BaysianPr(new_pr,MAP[x][y]);
        break;
 case 2: MAP[x][y] = new_pr;
        break;
 case 3: MAP[x][y] = BaysianPr(PrOnRisk(new_pr),MAP[x][y]);
        break;
 default: break;
 }
 Draw_GridImage(x,y,MAP[x][y]);
}
```