# APPENDICES

# APPENDIX A. DERIVATION OF OPTICAL EQUATIONS

This appendix shows the detailed derivation for basic equations describing the Rocky Rover obstacle sensors. All equations make perfect thin lens and "pinhole" camera assumptions. The actual curvature of the wide angle lens used on the Utah State University test bed vehicle complicates the matter. The actual parameters for pixel deviation need to be determined heuristically; however, for simulation and first cut approximations the following equations are useful.

Given that the angles from the lens to an object can be determined by the location of the image of the object in the focal plane of the camera the following equations provide two of the three equations necessary to solve for the object's exact location:

$$\tan(\phi_c - \phi) = \frac{h_c - z_o}{y_o} \tag{1}$$

and

$$\tan\theta = \frac{x_o}{\sqrt{y_o^2 + (h_c - z_o)^2}} \tag{2}$$

where $h_c$ is the height of the camera, $\phi_c$ is the angle of the camera declination from horizontal, $(\theta, \phi)$ are the coordinate pair for the angle of the object from the centerline extending normal to the lens, and $(x_o, y_o, z_o)$ is the triplet of the objects location with the origin of the coordinate frame at ground level directly below the camera lens with positive $x$-axis to the right, the positive $y$-axis forward, and (of course) the positive $z$-axis is toward the sky vertically.

The third equation necessary to solve for the unknowns $(x_o, y_o, z_o)$ is given by the equation of the plane of light created by the striping lasers. All planes are vertical, so the expression for the normal vector is

$$\vec{N_n} = \cos(\theta_{l_n})\vec{i} - \sin(\theta_{l_n})\vec{j}$$

where $\theta_{l_n}$ is the angle from the $y - axis$ direction the $n^{th}$ laser protrudes and $l_n$ reefers to the $n^{th}$ laser plane itself. Since all laser planes intersect at a vertical line directly in front of the rover,

$y_o = d_f$ and $x_o = \pm d_o$, the planar equations can be written

$$\cos\theta_{l_n}(x_o \mp d_o) - \sin\theta_{l_n}(y_o - d_f) = 0$$

or rearranged to form

$$y_o \sin\theta_{l_n} - x_o \cos\theta_{l_n} = D_n \tag{3}$$

where

$$D_n = d_f \sin\theta_{l_n} \mp d_o cos\theta_{l_n}.$$

The constant $d_f$ being the distance in front of the rover the lasers cross and $\pm d_o$ being the camera offset distance from the center of the rover (top sign for the left camera and bottom for the right camera).

At this point it is convenient to define geometric constants and write Equation (3) as

$$a_n y_o - b_n x_o = D_n \tag{4}$$

with

$$a_n = \sin\theta_{l_n}$$

$$b_n = \cos\theta_{l_n}$$

$$D_n = d_f a_n \mp d_o b_n$$

.

Now, by combining Equations (1) (2) and (4), an expression for $y_o$ dependent only on $\theta$, $\phi$, and $n$ is found:

$$y_o(\theta, \phi, n) = \frac{D_n}{a_n - \tan\theta\, b_n \sqrt{1 + \tan^2(\phi_c - \phi)}}. \tag{5}$$

Now $x_o$ and $z_o$ can be found in terms of $y_o$ in Equations (6) and (7).

$$z_o(y_o, \phi) = h_c - y_o \tan(\phi_c - \phi) \tag{6}$$

$$x_o(y_o, n) = \frac{a_n y_o - D_n}{b_n} \tag{7}$$

Equations (5) (6) and (7) would be more useful if they where referenced to the front center of the rover. This translation only effects $x_o$, assuming the cameras are directly above the front of the rover. In the rover reference frame:

$$(x_r, y_r, z_r) \rightarrow (x_o \pm d_o, y_o, z_o)$$

So Equation (7) becomes

$$x_r(y_r, n) = \frac{a_n y_r - D_n}{b_n} \pm d_o \tag{8}$$

and Equations (5) and (6) do not change except for the subscript $r$.

With Equations (5)(6) and (8), the exact location of a lasers intersection with an object can be calculated if the angular positions and the laser associated with the object are known.

To relate the ray described by the angles $\theta$ and $\phi$ to the location on focal plane the geometries of the camera must be specified. These equations introduce the error of neglecting the curvature caused by the lens of the system itself. For small angles these equations are correct and will serve as approximations elsewhere. Note that all equations are derived as if the focal plane was in front of the camera (i.e. consider that the equations here have been mirrored so the image is not inverted).

We can specify a relative height and width of the focal plane knowing the full angle field of view, $\Phi_{FOV}$, and the distance the focal plane, $\eta$, is behind the lens as

$$h_{fp} = 2\eta \, \tan \frac{\Phi_{FOV_y}}{2} \tag{9}$$

and

$$w_{fp} = 2\eta \, \tan \frac{\Phi_{FOV_x}}{2} \tag{10}$$

with the x and y subscripts on the field of view allowing for different geometries on the vertical and horizontal axis of the lens or focal plane.

The distance between the center of pixels on the focal plane are written in relative terms of $w_{fp}$ and $h_{fp}$ as

$$\Delta_{y_p} = \frac{h_{fp}}{N_y} \tag{11}$$

and

$$\Delta_{x_p} = \frac{w_{fp}}{N_x} \tag{12}$$

where $N_y$ and $N_x$ are the number of pixels in there respective dimensions.

The center pixel will be ordered $(0, 0)$ and all other pixels denoted by a pair $(x_p, y_p)$ pixels to the right and up increasing by integers. So a pixel in the upper left quadrant would have the signs $(-, +)$. The pixel location is related to the angles $\phi$ and $\theta$ by

$$\tan\theta = \frac{x_p \, \Delta_{x_p}}{\eta} = \frac{x_p w_{fp}}{\eta N_x} \tag{13}$$

and

$$\tan\phi = \frac{y_p \, \Delta_{y_p}}{\eta} = \frac{y_p h_{fp}}{\eta N_y} \tag{14}$$

Another pair of geometric constants are defined to simplify the equations:

$$g_{xp} = \frac{2\tan\frac{\Phi_{FOV_x}}{2}}{N_x} \tag{15}$$

and

$$g_{yp} = \frac{2\tan\frac{\Phi_{FOV_y}}{2}}{N_y}. \tag{16}$$

Combining Equations (9)(10)(11)(12)(13) and (14) and making judicious substitution of (15) and (16), simple forms for the angular displacement to pixel location can be written as

$$\tan\theta = x_p g_{xp} \tag{17}$$

and

$$\tan\phi = y_p g_{yp} \tag{18}$$

.

The means of solving for the angles given the pixel location are now defined and ready for combination with the previous equations.

Here equations are combined to write the obstacle location in terms of the pixel location and the laser associated with the obstacle. In other words, by detecting the pixel location, the location of the obstacle can be determined with the additional information of the location of the laser plane causing the image in the first place. Also, scan lines are introduced so that equations are dealing with only preset rows of pixels.

Equations (5) and (6) combine with (17) and (18) to eliminate $\theta$ and $\phi$ and form:

$$y_r(x_p, y_p, n) = \frac{D_n}{a_n - x_p g_{yp} b_n \sqrt{1 + \tan^2\left[\phi_c - \tan^{-1}(y_p g_{yp})\right]}} \tag{19}$$

and

$$z_r(y_r, y_p) = h_c - y_r \tan\left[\phi_c - \tan^{-1}(y_p g_{yp})\right]. \tag{20}$$

Equation (8) is unchanged since it is dependent on $y_r$ only.

As a point of interest the shape of the laser stripes as projected on the focal plane are defined by the following function of $y_p$ in terms of $x_p$ for the $n^{th}$ laser.

$$y_p = \frac{1}{g_{yp}} tan\left[\phi_c + \tan^{-1}\left(\frac{h_c a_n}{D_n + x_p h_c g_{xp}}\right)\right] \tag{21}$$

The algorithms used for the Rocky Rover break the processing down into scan lines. Each scan line is a row of pixels on the focal plane that represents a horizontal line along the ground at a particular distance from the vehicle. The algorithms will process $M$ scan lines so parameters defined by scan lines are denoted with the subscript $m$. Each scan line has an angle of depression with respect to the camera, $\phi_m$, associated with it. Using Equation (1) and setting $z_o$ equal to 0, ground level, $\phi_m$ is found to be:

$$\phi_m = \phi_c - \tan^{-1}\left(\frac{h_c}{y_{s_m}}\right) \tag{22}$$

where $y_{s_m}$ is the distance from the front of the vehicle in which the scan line projects. Equation (22) can substitute into (18) resulting in the scan line pixel location $y_{p_m}$.

$$y_{p_m} = \frac{\tan\left[\phi_c - \tan^{-1}\left(\frac{h_c}{y_{s_m}}\right)\right]}{g_{yp}} \tag{23}$$

Two scan line geometry parameters can be defined to simplify the end equations.

$$c_m = \tan(\phi_c - \phi_m) \tag{24}$$

$$e_m = \sqrt{1 + c_m^2} \tag{25}$$

Equation (23) written in terms of (24) becomes

$$y_{p_m} = \frac{c_m}{g_{yp}}. \tag{26}$$

Finally the coordinate transfer equations are written with the scan lines included as:

$$y_r(x_{p_{m,n}}, m, n) = \frac{D_n}{a_n - x_{p_{m,n}} g_{xp} b_n e_m} \tag{27}$$

$$z_r(y_r, m) = h_c - y_r c_m \tag{28}$$

and

$$x_r(y_r, n) = \frac{y_r a_n - D_n}{b_n} \pm d_o \tag{29}$$

$x_{p_{m,n}}$ denoting the pixel location for the $n^{th}$ laser for the $m^{th}$ scan line.

Once the geometries of the sensor system are defined, Equations (27)(28) and (29) provide calculation of object location given pixel location of laser stripe.

# APPENDIX B. SOURCE CODE

Header files for inclusion in source code.

```
// fuzz_alg.h
// written by: Tim McJunkin
//
//      Copyright 1994 Utah State University
//
//   January 19, 1994
//
//   General header file for fuzzy map routines
//
//   Last Modified: July 18, 1994
//
#ifndef _fuzz_alg_h
#define _fuzz_alg_h

#include <stdlib.h>
#include <stdio.h>
#include <fstream.h>
#include <string.h>
#include <strstrea.h>
#include <alloc.h>
#include <fuzz_def.h>                              // fuzzy define values

typedef struct info{
  int direction;              // in

  float *memvalues;           // out
  float confidence[5];   // 0= help,1 = forward, 2 = left, 3 back, 4
right
  unsigned char far *image[2];                        // 0 = left, 1=
right
  float x,y,theta;                                   // north=0.0
clockwise
  float dx, dy, dtheta;                                   //
target/desired
} RoverData;


//header information for routines defined
//in fuzz_alg.cpp

int get_fuzzy_values(RoverData *Values);
//call to have rover collect sensor information and pass
//it back in Values. Normally called by user interface each
//move
```

```
int UpdateMap(RoverData *Values , int Camera);
//get sensor information for given camera or both

int UpdateSimMap(RoverData *, int Camera);
//get map information from previously saved run.

void subtract(unsigned char far *first, unsigned char far *second);
//difference to images and return in location of first
//image

void Calibrate(RoverData *);
//obsolete code that was meant to auto calibrate system
//on startup.  note: not practical for most situations

int PixelDev(int *pixel, int scanline, int Camera);
//return pixel deviation back to pixel given pixel
//locations in pixel the scanline and camera involved
//attempts to correct for false laser locations.

int AddToMap(float *memvalue, int *pixel_dev , int scanline, int
camera);
//takes pixel devation and translates that into a
//membership values which are added into the map at
//appropriate locations per scanline and camera

int LookUpFuzzy(int *delta_pix ,float *memvalue, int row, int Camera);
//maps pixel deviation into membership value
//using appropriate membership function as per row and Camera

int TranslateMap(float *memvalue , int direction);
//given a movement the values in the map are translated
//according to direction passed.

int WriteCalibration(char *);
int ReadCalibration(char *, RoverData *);

void init_fuzzy(RoverData *);
void cleanup_fuzzy(RoverData *);
//initialize and shutdown map.  allocate. initialize. etc.

int WriteImage(unsigned char far *, char *);
int ReadImage(unsigned char far *, char *);
int NameImage(char *, int, char * ="" );

int LoadSeries(char *);
int WriteMap(RoverData *);
int ReadMap(RoverData *, FILE *);

int ScoreImage(unsigned char far *, unsigned char far *);
```

```
//header information for routines in filter.cpp
int FindPixels(int*, int, int,
                      unsigned char far *, unsigned char far *);
int FindPixelsNew(int*, int, int,
                          unsigned char far *, unsigned char far*);


float pix_filter(unsigned char far *);
float pix_filt_dif(unsigned char far *, unsigned char far *);

//header information for calls to routines
//that handle image/frame grabber operations
//and ADAC commands
//file: rov_adac.c for source
int  Init_ADAC();
void Lasers(int);
void init_odx();
int  TakePicture(int, unsigned char far * );
int  MoveRover(int );

//header information for routines that are
//most likely obsolete
int fget_fuzzy_values(float*, int);
int fUpdateMap(float*, int , ifstream&);
int fFindPixels(float* ,int, int, ifstream&);
int MapDump(float* );


//Dave Madsen's fuzzy decision maker calles
//in fuzycont.c
int fuzzcont(RoverData *);

//dead reckoning header for dead.c
int  DeadReckon(RoverData *);
void FindHeading(RoverData *);

//calls to display text to user interface
void disp_text(char *string);
void WriteTextMessage(char * , ...);
//same formatting as printf(char *, ...);
#endif
```

Definitions for symbols used throughout source code.

```
// fuzz_def.h
//   Tim McJunkin
//   April 20, 1994
//
//   Header file for defines used in association
//   with calls to fuzz_alg.cpp
//   Included in fuzz_alg.h
//
//   Last Modified April 22, 1994 TMc
#ifndef _fuzz_def_h
#define _fuzz_def_h

#define C_BOTH 0
#define C_LEFT -1
#define C_RIGHT 1

#define I_LEFT 1
#define I_RIGHT 0

#define ERROR 0
#define D_HELP 0
#define D_FORWARD 1
#define D_LEFT 2
#define D_RIGHT 3
#define D_BACKWARD 4

#define LASER_ON  1
#define LASER_OFF 0

//current image dimensions
#define ImageW 320L
#define ImageH 200L

//frame grabber port number
#define FRAME_PORT 0x330

//no of lasers acknowledged by each camera
#define NoLasCam 3
#define NoLasers 5

//default calibration file
#define DEFAULT_CALIB "c:\\rover\\bin\\calib.dat"
#define DEFAULT_SERIES "c:\\rover\\bin\\series.dat"
#define DEFAULT_READ   "c:\\rover\\bin\\readsers.dat"
```

```
#define IMAGE_PATH     "c:\\rover\\image\\"
#define DATA_PATH      "c:\\rover\\data\\"

//dead reckoning constants in centimeters
#define FORWARD_DIST 11
//distance front center is displaced
#define TURN_SIDE_DIST 5
#define TURN_FORW_DIST 10
#define DELTA_THETA    7.5
#define BACKWARD_DIST  11
#define FORWARD_DIST   11


#define FL 0
#define FML 1
#define FC 2
#define FMR 3
#define FR 4

#define L 5
#define ML 6
#define C 7
#define MR 8
#define R 9

#define NL 10
#define NML 11
#define NC 12
#define NMR 13
#define NR 14

#define PL 15
#define PR 16

#define FSL 17
#define FSR 18
#define FaSL 23
#define FaSR 24

#define SL  21
#define SR  22

#define BSL 19
#define BSR 20

#define BL 25
#define B  26
#define BR 27
```

```
#endif
```

Header file for ADAC board routines.

```
//
// file: adac.h
#ifndef _adac_h
#define _adac_h

/*
 * Name of hardware config file.
 */

#define CONFIG_FILE "C:\\rover\\bin\\adac.dat"

/*
 * Channel for each type of movement.
 */

#define TURN_CHANNEL 0
#define MOVE_CHANNEL 1
#define DIGI_CHANNEL 0

/*
 * Which bit means what.
 */

#define LASER_BIT 1
#define CAMERA_BIT 2

/*
 * Values that will cause the steering to go to the appropriate
condition.
 */

#define TURN_LEFT 140
#define TURN_NEUTRAL 500
#define TURN_RIGHT 920

/*
 * Values that will cause the motion to enter the appropriate state.
 */

#define MOVE_FORWARD 860
#define MOVE_NEUTRAL 580
#define MOVE_BACKWARD 240

/*
 * Hold times for each of the outputs.
```

```
 */

#define TURN_LEAD 1
#define TURN_HOLD 6
#define MOVE_HOLD 4  // was 100

#endif /* _adac_h */
```

Main fuzzy mapping routines are coded in file fuzz_alg.cpp.

```
// file: fuzz_alg.cpp
// written by: Tim McJunkin
//
// Copyright 1994 Utah State University
//
// this file provides the functions that
// provide for driving the rover from
// information provided in the call of
// get_fuzzy_values(float *, int)
// it provides calls to functions that
// will control the adac board or in the
// case of simulation move the virtual
// rover.  It will provide calls to
// grab images from the frame grabber card
// and processing functions or simulation
// files for the same data.
// The meat of this file is the fuzzy
// map algorithm for updating the map
// with new sensor data and translating
// the map with rover movement instructions.

// primarilly called from windows user interface
// software

// for: Center for Self Organizing and Intelligent
//      Systems - Utah State University
// funding from: Rocky Mountain NASA Space Grant
//                         Consortium
// date: January 20, 1994
//
// this source file
// last modified: July 18, 1994
//
// modified Febuary 6, 1994:
//    added place to put camera switching routine
//    for ADAC bored and GetImage() call to call
//    Frame Grabber code in GSTIM.C
//
// Modification Febuary 8, 1994:
//      Made Frame grabber code work must faralloc(LONG)
//      with the empasis on LONG ints don't work for
//      64000.
//      Took out test file code.
//      Working on Image algorithm.
```

```
//
// Modification June 14, 1994
// Added structure for passing parameters
//
// Modification July 18, 1994
// Added comments and deleted unused commented
// obsolete code in preparation for delivery to
// INEL
//
// Other header files included from
// fuzz_alg.h
// fuzz_alg.h includes all other necesary header files.
//
//
#define image_yes

#include <fuzz_alg.h>
#define max(a,b)    (((a) > (b)) ? (a) : (b))
#define min(a,b)    (((a) < (b)) ? (a) : (b))

#define NumberOfRegions  28

//Expected Pixel location of Pixels for flat
// terrain for the Right Camera.
// left to right - center, right, far right
// rows starting at top 3   Vehicle Lengths
//                      1.5 Vehicle Lengths
//                      1   Vehicle Lengths
//
// Data from Excel Worksheet for ideal lens.
// Calibration of actual sensors will determine
// them heuristically.
//
// Left Camera must reverse the order of the
// Rows and Change the sign to have appropriate
// value.  This is taken care of in function
// for getting pixel values.

   int   *RExpectedPixels;  //global pointer allocated in
                            //init_fuzzy();
   int   *LExpectedPixels;  //global pointer allocated in
                            //init_fuzzy();

// FuzzPoint defines key points of fuzzy membership functions
// versus pixel deviation.  These values are grouped as
// follows: Far Scanline, Medium Scanline, Near Scanline
// Each group is ordered rows: Center Laser, Onside Laser(RIGHT)
// Far Onside(RIGHT) Laser.
// Each row defines key points a,b,c,d,e,f for the shape of
```

```
// the piecewise linear curve:
//       a - any deviation larger than a is a critical hole m =1
//       b - m=0 point interpolate between a and b
//       c - m=0 point values between b and c are 0
//       d - m=1 point interpolate between c and d
//           a value (c+d)/2 is the pixel deviation of the
//           defined critical height(distance) of obstacle
//           to give m = 0.5
//       e - m=1 point values between d and e are m=1
//       f - m=0 point interpolate between e and f
//           a value greater than f is m =0 because detection
//           is in nearer region and blocking detection of
//           anything in that region
//
   int *RFuzzPoint; //global pointer allocated in init_fuzzy();
   int *LFuzzPoint; //global pointer allocated in init_fuzzy();

// scan_pixel holds placement for pixel row to use for each
// scan line row 0 being the furthest out scanline
   int no_scan_lines  = 3;
// initialized to default value
// read in from config.dat must exist
   int *Rscan_pixel; //global pointer allocated in init_fuzzy();
   int *Lscan_pixel; //global pointer allocated in init_fuzzy();

//       #include <adliborl.h>
//       #include <adac.h>
   #include <math.h>

// global variables and flags
   int Series; //tracks the series increments each run
   int move_no=0; //increments each move
   int save_image_flag=0; //saves all images if set in config.dat
   int save_data_flag=0;  //saves all map data if set
   int sub_image_flag= 0; //display subtracted images if set
   int simulation_flag=0; //read map and images from saved files
                          //no actual run if set

   FILE *sim_fp;          //simulation file pointer

   unsigned char far *image = NULL; //pointer to image used

                                    // for laser off images

   int get_fuzzy_values(RoverData *Values){
//
// Function: get_fuzzy_values
// is called by the user interface to update
// the region membership values and to pass
```

```
// the users control direction to the process
//
// structure RoverData includes direction
// passed by user interface
//
// returned through structure:
//      pointer to updated map values
//      pointer to new images
//      dead reckonning informations
//      suggestions from  automated system
//
// Direction:
//   D_FORWARD  = 1
//   D_LEFT     = 2
//   D_RIGHT    = 3
//   D_BACKWARD = 4
//   BOTH       = 0  Take picture with both cameras
//                   no movement of rover
//                   Used for initial sensor information
//                   or possibly after map is reset for
//                   some reason.
//
//

//
// Written by: Tim McJunkin
// Date: January 20, 1994
// Last Modified: Febuary 21, 1994
//

// selected appropriate program route
// depending on simulation flag
if (simulation_flag){
   UpdateSimMap(Values, C_BOTH);
}else
{
   switch (Values->direction){
      case D_BACKWARD :
      case D_RIGHT:
      case D_LEFT:
      case D_FORWARD :
         UpdateMap(Values, C_BOTH);
         break;
      case 0:
         UpdateMap(Values, C_BOTH);
         break;
      default:
         exit(0);
   }
```

```
}
move_no++;

return(1);
}

int UpdateMap( RoverData *Values, int Camera){
   int Pixel[6];  // hold laser pixel locations on a scanline
   int newpix[6];
   char buf[80];
   int glitch_flag;
   int repeat;
   int row;
   char Image_file[50];


   if (save_data_flag)
      WriteMap(Values);
   if (Values->direction){

// if DEBUG is defined durring compile do not move rover
#ifndef DEBUG
      MoveRover(Values->direction);
#endif
      DeadReckon(Values);
      FindHeading(Values);
      TranslateMap(Values->memvalues, Values->direction);
      }

   if (Camera >= 0){  //If right camera or both cameras
      glitch_flag = 0; //glitch detection not used
      repeat = 0;
      while (glitch_flag && repeat++ < 5){
         glitch_flag=0;
         Lasers(LASER_OFF);
         TakePicture(C_RIGHT, image);
         Lasers(LASER_ON);
         TakePicture(C_RIGHT, Values->image[I_RIGHT]);
         Lasers(LASER_OFF);
         for (row=0; row < 3; row++){
            Pixel[0] = 0;
            Pixel[1] = 0;
            Pixel[2] = 0;
            newpix[0] = 0;
            newpix[1] = 0;
            newpix[2] = 0;
            FindPixels(Pixel, Rscan_pixel[row], C_RIGHT,
            Values->image[I_RIGHT], image);
#ifdef CALIBRATE
```

```
            WriteTextMessage("%d   %d    %d   %d   %d   %d",
                             Pixel[0], Pixel[1], Pixel[2],
                             Pixel[3], Pixel[4], Pixel[5]);
#endif
//
//glitch detection goes here when developed
//  set glitch flag if inconsistancy in images occur

            if (!glitch_flag || repeat==5){
               PixelDev(Pixel, row, C_RIGHT);
               AddToMap(Values->memvalues, Pixel, row, C_RIGHT);
            }
         }
    }
    Lasers(LASER_OFF);
    ScoreImage(Values->image[I_RIGHT], image);
    if (save_image_flag){
       NameImage(Image_file,C_RIGHT);
       WriteImage(Values->image[I_RIGHT], Image_file);
       if (sub_image_flag)
          subtract(Values->image[I_RIGHT], image);
       NameImage(Image_file,C_RIGHT,"s");
       WriteImage(Values->image[I_RIGHT], Image_file);
    }
  }

   if (Camera <= 0){
      glitch_flag = 0;
      repeat = 0;
      while (glitch_flag && repeat++ < 5){
         glitch_flag=0;
         Lasers(LASER_OFF);
         TakePicture(C_LEFT, image);
         Lasers(LASER_ON);
         TakePicture(C_LEFT, Values->image[I_LEFT]);
         Lasers(LASER_OFF);
         for (row=0; row < 3; row++){
            Pixel[0] = 0;
            Pixel[1] = 0;
            Pixel[2] = 0;
            FindPixels(Pixel, Lscan_pixel[row], C_LEFT,
                       Values->image[I_LEFT], image);
#ifdef CALIBRATE
            WriteTextMessage("%d   %d    %d   %d   %d   %d",
                             Pixel[0],Pixel[1], Pixel[2],
                             Pixel[3],Pixel[4], Pixel[5]);
#endif
```

```
            if (!glitch_flag || repeat==5){
               PixelDev(Pixel, row, C_LEFT);
               AddToMap(Values->memvalues, Pixel, row, C_LEFT);
            }
         }
   }
   ScoreImage(Values->image[I_LEFT], image);
   if (save_image_flag){
      NameImage(Image_file,C_LEFT);
      WriteImage(Values->image[I_LEFT], Image_file);
      if (sub_image_flag)
         subtract(Values->image[I_LEFT], image);
      NameImage(Image_file,C_LEFT,"s");
      WriteImage(Values->image[I_RIGHT], Image_file);
   }
 }

Lasers(LASER_OFF);
Values->direction = fuzzcont(Values);
return(1);
}

int UpdateSimMap (RoverData *Values, int Camera){
   int foo=1;
   static char Image_file[40];

   if (ReadMap(Values, sim_fp)){
      if (Camera >= 0){ //right camera
         NameImage(Image_file,C_RIGHT);
         ReadImage(Values->image[I_RIGHT], Image_file);
         ScoreImage(Values->image[I_RIGHT], image);
      }
      if(Camera <=0){ //left camera
         NameImage(Image_file,C_LEFT);
         ReadImage(Values->image[I_LEFT], Image_file);
         ScoreImage(Values->image[I_LEFT], image);
      }
   }
return foo;
}




void subtract(unsigned char far *image1, unsigned char far *image2){
   int temp;
```

```
   for (long i=0; i<(320L*200L); i++){
      temp = abs((int)(*image1 - *image2));
      *image1 = temp;
      image1++;
      image2++;
   }
}


int PixelDev(int *Pixel, int scanline, int Camera){
   int DeltaPix[6]={0,0,0,0,0,0};
   int Pix;
   int i,j;
   int Start = scanline * 3;
      for (i=0;i<3;i++){
                  // Pix will hold current pixel deviation
         Pix = -11; // initialize Pix to enter while loop
                  // help take care of reflections by not
                  // allowing large negative deviations
                  // if large negative deviation then
                  // shift Pixel[i+1] to Pixel[i]
                  // etc. unitil bogus detects are eliminated
                  // unfortunately there will always be some
                  // ambiguity in specifying what laser caused
                  // what spot when all lasers are turned on.
                  // this section of code tries to eliminated
                  // inadvertant mistakes are interpret them such
                  // that mistakes are non critical.  Thus
                  // close scanlines will be given large pixel
                  // deviation
                  // if a mistake is possible and far scanlines are
                  // given
                  // 0 pixel deviations since they are less critical
         while (Pix <-10){
            switch (Camera){
               case C_RIGHT:
                  Pix = Pixel[i]-RExpectedPixels[Start+i];
                  break;
               case C_LEFT:
                  Pix = Pixel[i]-LExpectedPixels[Start+i];
                  break;
               default:
                  exit(0);
            }
            if (Pixel[i] <= 0)
               if (scanline == 2)
                  Pix = -69; // indicated large cliff if
                             // no laser is detected on
                             // near scanline
```

```
                        // note gives same result as
                        // large obstacle blocking camera
                        // view in actual map
            else
               Pix = 0;   // indicated no obstacle since
                          // at far scanlines the missing
                          // laser could be caused by low
                          // signal to noise ratio dependant
                          // on light conditions
         if (Pix < -10)
            for (j=i;j<5;j++)
               Pixel[j] = Pixel[j+1]; // shift laser locations
      }
// if pixel location is inside the expected location
// of the next laser expected location shift all
// of them down and set DeltaPix to 0


      if (i<2) // don't do this on 3rd laser expected pixel
               // will be wrong. since i+1 expected pixel location
               // does not exist (i.e. i=3 is not valid)
         if (((Pixel[i]+10)>RExpectedPixels[Start+i+1]
              &&Camera==C_RIGHT)||
              ((Pixel[i]+10)>LExpectedPixels[Start+i+1]
              &&Camera==C_LEFT))
         {

               // if a laser isn't found on the near scanline
               // then make the devation such that the region
               // is impassable because this means either
               // there is a cliff and the light isn't found
               // or there is somthing blocking the line of
               // sight of the camera.  This is extremely
               // critical on the near scanline.  On further
               // scanlines the lighting conditions may
               // be such that the laser can not be found.

            if (scanline==2)
               Pix = -69;
            else
               Pix = 0;

            Pixel[i+2] = Pixel[i+1];  //shift value towards center
            Pixel[i+1] = Pixel[i];    //lasers since the value
                                      //of Pixel[i] is closer to
                                      //the next region
          }
          DeltaPix[i] = Pix;  //put value into array of pixel
                              //deviations
```

```
     }

     for (i=0;i<3;i++)
        Pixel[i] = DeltaPix[i]; //return pixel deviations in Pixel
return(1);
}


int AddToMap(float *RegMem, int *Pixel, int row,
             int Camera){

   float fuz_val[NoLasCam];

   // NoLasers is total number of lasers
   //    also number of regions associated with
   //    each scanline
   //    First regions are the active regions numbered
   //    0 through NoLasers * # of scanlines
   // NoLasCam is number of lasers used by each camera
   int Start = row * NoLasers + 2;     //Start from center laser

   // pass pixel deviation in Pixels and return
   // membership values for row in fuz_val
   LookUpFuzzy(Pixel,fuz_val, row, Camera);

   for (int i=0; i<NoLasCam; i++){
      if ((Camera == C_LEFT))
         RegMem[Start-i] = min(1,RegMem[Start-i] + fuz_val[2-i]);
      else    //right camera
         RegMem[Start+i] = min(1,RegMem[Start+i] + fuz_val[2-i]);

   }
return (1);
}

int LookUpFuzzy(int *Pixel,float *fuz_val, int row, int Camera){
// Determine Membership value according to Pixel
// Deviation replace Pixel values with Membership
// value as defined in calib.dat file.

   int *C_Fuzz;
   float val;
   int Start = 6*NoLasCam*row; //six points per laser
                               //three lasers per row

   switch (Camera){
      case C_RIGHT:
         C_Fuzz = RFuzzPoint + Start;
         break;
```

```
      case C_LEFT:
         C_Fuzz = LFuzzPoint + Start;
         break;
      default:
         exit(0);
   }

   for (int i=0;i<NoLasCam;i++){ // through three lasers on the scanline

      //Large Dip
      if ((val = -Pixel[i]) > C_Fuzz[0])
         fuz_val[i] = 1.0;

      else if (val > C_Fuzz[1] )        //Small Dip
              fuz_val[i] = (val-C_Fuzz[1])/
                           (C_Fuzz[0]-C_Fuzz[1]);

      else if (val > C_Fuzz[2] )        //Level surface
              fuz_val[i] = 0;

      else if (val >= C_Fuzz[3] )        //Small Protrusion
              fuz_val[i] = (C_Fuzz[2]-val)/
                           (C_Fuzz[2]-C_Fuzz[3]);

      else if (val >= C_Fuzz[4] )        //Full Protrusion
              fuz_val[i] = 1;

      else if (val >= C_Fuzz[5] )        //Nearing Protrusion
              fuz_val[i] = (val-C_Fuzz[5])/
                           (C_Fuzz[4]-C_Fuzz[5]);

      else                               //Protrusion in near
             fuz_val[i] = 0;           //Region
      if (i==2) fuz_val[i] *= 0.6; //for center laser

//factor value down
   C_Fuzz += 6;   // jump to next lasers membership function
   }
return(1);
}//LookUpFuzzy()//




int TranslateMap(float *RegMem, int direction){
// Reduce Each region membership by appropriate amount
// Then adjust upward the regions in which values would
// Travel
```

```
int i;

if (direction == D_FORWARD){

RegMem[FaSL] = 0.9  *RegMem[FaSL]   + 0.11 * RegMem[FL];
RegMem[FaSR] = 0.9  *RegMem[FaSR]   + 0.11 * RegMem[FR];
RegMem[B]    = 0.7  *RegMem[B];
RegMem[BL]   = 0.65 *RegMem[BL]   + 0.2  * RegMem[SL]
              +0.25  *RegMem[BSL];
RegMem[BR]   = 0.65 *RegMem[BR]   + 0.2  * RegMem[SR]
              +0.25  *RegMem[BSR];
RegMem[SL]   = 0.7  *RegMem[SL]   + 0.25 * RegMem[PL];
RegMem[SR]   = 0.7  *RegMem[SR]   + 0.25 * RegMem[PR];
RegMem[BSL]  = 0.6 *RegMem[BSL]   + 0.28  * RegMem[FSL];
RegMem[BSR]  = 0.6 *RegMem[BSR]   + 0.28 * RegMem[FSR];
RegMem[FSL]  = 0.6  *RegMem[FSL]   + 0.4* RegMem[NL];
RegMem[FSR]  = 0.6  *RegMem[FSR]   + 0.4* RegMem[NR];
RegMem[PR]   = 0.5  *RegMem[PR]   + 0.750* RegMem[R];
RegMem[PL]   = 0.5  *RegMem[PL]   + 0.750* RegMem[L];
RegMem[NL]   = 0.3 *RegMem[NL]   + 0.300* RegMem[ML];
RegMem[NML]  = 0.5  *RegMem[NML]   + 0.300* RegMem[ML];
RegMem[NC]   = 0.7  *RegMem[NC]   + 0.400* RegMem[C];
RegMem[NMR]  = 0.5  *RegMem[NMR]   + 0.300* RegMem[MR];
RegMem[NR]   = 0.3  *RegMem[NR]   + 0.300* RegMem[MR];
RegMem[L]    = 0.2  *RegMem[L]    + 0.15 * RegMem[FML];
RegMem[ML]   = 0.2  *RegMem[ML]    + 0.28 * RegMem[FML];
RegMem[C]    = 0.2  *RegMem[C]    + 0.43 * RegMem[FC];
RegMem[MR]   = 0.2  *RegMem[MR]    + 0.28 * RegMem[FMR];
RegMem[R]    = 0.2  *RegMem[R]    + 0.15 * RegMem[FMR];

for (i=0;i<5;i++)
   RegMem[i] *= 0.600;

}// end of D_FORWARD translation equations

if (direction ==D_BACKWARD){
RegMem[FL]   = 0.6  *RegMem[FL]    + 0.2  * RegMem[FaSL];
RegMem[FR]   = 0.6  *RegMem[FR]    + 0.2  * RegMem[FaSR];
RegMem[FML]  = 0.6  *RegMem[FML]    + 0.25 * RegMem[L]
            + 0.28 * RegMem[ML];
RegMem[FMR]  = 0.6  *RegMem[FMR]    + 0.25 * RegMem[R]
            + 0.28 * RegMem[MR];
RegMem[FC]   = 0.6  *RegMem[FC]    + 0.5  * RegMem[C];
RegMem[L]    = 0.4  *RegMem[L]    + 0.15 * RegMem[PL];
RegMem[ML]   = 0.4  *RegMem[ML]    + 0.28 * RegMem[NL]
            + 0.00 * RegMem[NML];
RegMem[C]    = 0.4  *RegMem[C]    + 0.43 * RegMem[NC];
```

```
RegMem[MR]    = 0.4  *RegMem[MR]     + 0.28 * RegMem[NMR]
                + 0.00 * RegMem[NR];
RegMem[R]     = 0.4  *RegMem[R]     + 0.15 * RegMem[PR];
RegMem[NL]    = 0.4  *RegMem[NL]    + 0.300* RegMem[FSL];
RegMem[NML]   = 0.4  *RegMem[NML];
RegMem[NC]    = 0.4  *RegMem[NC];
RegMem[NMR]   = 0.4  *RegMem[NMR];
RegMem[NR]    = 0.4  *RegMem[NR]    + 0.300* RegMem[FSR];
RegMem[FSL]   = 0.6  *RegMem[FSL]    + 0.350* RegMem[BSL];
RegMem[FSR]   = 0.6  *RegMem[FSR]    + 0.350* RegMem[BSR];
RegMem[PR]    = 0.5  *RegMem[PR]    + 0.750* RegMem[SR];
RegMem[PL]    = 0.5  *RegMem[PL]    + 0.750* RegMem[SL];
RegMem[SL]    = 0.7  *RegMem[SL]    + 0.25 * RegMem[BL];
RegMem[SR]    = 0.7  *RegMem[SR]    + 0.25 * RegMem[BR];
RegMem[BSL]   = 0.65 *RegMem[BSL]    + 0.4  * RegMem[BL];
RegMem[BSR]   = 0.65 *RegMem[BSR]    + 0.4  * RegMem[BR];

RegMem[FaSL] = 0.9  *RegMem[FaSL];
RegMem[FaSR] = 0.9  *RegMem[FaSR];
RegMem[B]     = 0.7  *RegMem[B] + 0.2;
RegMem[BL]    = 0.7  *RegMem[BL] +0.15;
RegMem[BR]    = 0.7  *RegMem[BR] +0.15;
}// end D_BACKWARD translation

if (direction ==D_RIGHT){
RegMem[FaSL] = 0.8  *RegMem[FaSL]    + 0.2  * RegMem[FL];
RegMem[FaSR] = 0.95 *RegMem[FaSR]    + 0.05 * RegMem[FR];
RegMem[B]     = 0.7  *RegMem[B]    + 0.1  * RegMem[BSL]
                + 0.1  * RegMem[BL];
RegMem[BL]    = 0.7 *RegMem[BL]    + 0.3  * RegMem[SL]
                + 0.35 * RegMem[BSL];
RegMem[BR]    = 0.6 *RegMem[BR]    + 0.1  * RegMem[SR]
                + 0.15 * RegMem[BSR];
RegMem[SL]    = 0.6  *RegMem[SL]    + 0.30 * RegMem[PL];
RegMem[SR]    = 0.8  *RegMem[SR]    + 0.20 * RegMem[PR];
RegMem[BSL]   = 0.55 *RegMem[BSL]    + 0.34  * RegMem[FSL];
RegMem[BSR]   = 0.65 *RegMem[BSR]    + 0.24  * RegMem[FSR];
RegMem[FSL]   = 0.6 *RegMem[FSL]    + 0.50* RegMem[NL];
RegMem[FSR]   = 0.7 *RegMem[FSR]    + 0.30* RegMem[NR];
RegMem[PR]    = 0.6  *RegMem[PR]    + 0.4  * RegMem[FaSR];
RegMem[PL]    = 0.45 *RegMem[PL]    + 0.850* RegMem[L];
RegMem[NL]    = 0.20 *RegMem[NL]    + 0.300* RegMem[ML]
                + 0.300* RegMem[C];
RegMem[NML]   = 0.25 *RegMem[NML]    + 0.500* RegMem[C];
RegMem[NC]    = 0.3  *RegMem[NC]    + 0.400* RegMem[MR];
RegMem[NMR]   = 0.7 *RegMem[NMR]    + 0.200* RegMem[MR]
                + 0.100* RegMem[R];
RegMem[NR]    = 0.7 *RegMem[NR]    + 0.350* RegMem[R];
RegMem[L]     = 0.16 *RegMem[L]     + 0.25 * RegMem[FML]
```

```
                + 0.1  * RegMem[ML];
RegMem[ML]    = 0.18 *RegMem[ML]    + 0.19 * RegMem[FC]
                + 0.15 * RegMem[FMR]
                + 0.1  * RegMem[C];
RegMem[C]     = 0.20 *RegMem[C]     + 0.43 * RegMem[FMR]
                + 0.1  * RegMem[MR];
RegMem[MR]    = 0.22 *RegMem[MR]    + 0.28 * RegMem[FMR]
                + 0.2  * RegMem[R];
RegMem[R]     = 0.24 *RegMem[R]     + 0.15 * RegMem[FR];
RegMem[FR]   *= .38;
RegMem[FMR]  *= .42;
RegMem[FC]   *= .46;
RegMem[FML]  *= .48;
RegMem[FL]   *= .52;
}

if (direction ==D_LEFT){
RegMem[FaSR] = 0.8  *RegMem[FaSR]   + 0.2  * RegMem[FR];
RegMem[FaSL] = 0.95 *RegMem[FaSL]   + 0.05 * RegMem[FL];
RegMem[B]     = 0.7  *RegMem[B]   + 0.1  * RegMem[BSR]
                + 0.1  * RegMem[BL];
RegMem[BR]    = 0.7 *RegMem[BR]   + 0.3  * RegMem[SR]
                + 0.35 * RegMem[BSR];
RegMem[BL]    = 0.6 *RegMem[BL]   + 0.1  * RegMem[SL]
                + 0.15 * RegMem[BSL];
RegMem[SR]    = 0.6  *RegMem[SR]   + 0.30 * RegMem[PR];
RegMem[SL]    = 0.8  *RegMem[SL]   + 0.20 * RegMem[PL];
RegMem[BSR]   = 0.55 *RegMem[BSR]   + 0.34  * RegMem[FSR];
RegMem[BSL]   = 0.65 *RegMem[BSL]   + 0.24  * RegMem[FSL];
RegMem[FSR]   = 0.6 *RegMem[FSR]   + 0.50* RegMem[NR];
RegMem[FSL]   = 0.7 *RegMem[FSL]   + 0.30* RegMem[NL];
RegMem[PL]    = 0.6  *RegMem[PL]   + 0.4  * RegMem[FaSL];
RegMem[PR]    = 0.45 *RegMem[PR]   + 0.850* RegMem[R];
RegMem[NR]    = 0.20 *RegMem[NR]   + 0.300* RegMem[MR]
                + 0.300* RegMem[C];
RegMem[NMR]   = 0.25 *RegMem[NMR]   + 0.500* RegMem[C];
RegMem[NC]    = 0.35  *RegMem[NC]   + 0.400* RegMem[ML];
RegMem[NML]   = 0.7 *RegMem[NML]   + 0.200* RegMem[ML]
                + 0.100* RegMem[L];
RegMem[NL]    = 0.7 *RegMem[NL]   + 0.350* RegMem[L];
RegMem[R]     = 0.16 *RegMem[R]    + 0.25 * RegMem[FMR]
                + 0.1  * RegMem[MR];
RegMem[MR]    = 0.18 *RegMem[MR]    + 0.19 * RegMem[FC]
                + 0.15 * RegMem[FML]
                + 0.1  * RegMem[C];
RegMem[C]     = 0.20 *RegMem[C]     + 0.43 * RegMem[FML]
                + 0.1  * RegMem[ML];
RegMem[ML]    = 0.22 *RegMem[ML]    + 0.28 * RegMem[FML]
                + 0.2  * RegMem[L];
```

```
RegMem[L]    = 0.24 *RegMem[L]    + 0.15 * RegMem[FL];
RegMem[FL]  *= .38;
RegMem[FML] *= .42;
RegMem[FC]  *= .46;
RegMem[FMR] *= .48;
RegMem[FR]  *= .52;

}

for (i=0;i<28;i++){
    if (RegMem[i] < .01) RegMem[i] = 0.0;
    if (RegMem[i] > 1.0) RegMem[i] = 1.0;
//      disp_text("hey");
//      RegMem[i] = 0.0;
    }

return(1);
}



int WriteCalibration(char *cal_file)
{

// write out calibration information to
// calib.dat file.  This will delete
// all appending comments that were previously
// included so use with care if you want to
// preserve comments
   int i,j;
   int start;
   int foo = 0;
   ofstream Cal(cal_file);
   if (Cal){
      Cal << no_scan_lines << '\n';
      for (i=0;i<no_scan_lines;i++)
         Cal << Lscan_pixel[i] << ' ';
      Cal << '\n';
      for (i=0;i<no_scan_lines;i++)
         Cal << Rscan_pixel[i] << ' ';
      Cal << '\n';
      for (i=0;i<no_scan_lines;i++){
         for (j=0;j<3;j++)
            Cal<<LExpectedPixels[i*3+j]<<' ';
         Cal<<'\n';
      }
      for (i=0;i<no_scan_lines;i++){
         for (j=0;j<3;j++)
            Cal<<RExpectedPixels[i*3+j]<<' ';
```

```
         Cal<<'\n';
      }
      for (i=0;i<9;i++){
         start = i*6;
         for (j=0;j<6;j++)
            Cal << LFuzzPoint[start+j]<<' ';
         Cal<<'\n';
      }
      for (i=0;i<9;i++){
         start = i*6;
         for (j=0;j<6;j++)
            Cal << RFuzzPoint[start+j]<<' ';
         Cal<<'\n';
      }
      Cal << save_data_flag << '\n';
      Cal << save_image_flag << '\n';
      Cal.close();
      foo = 1;
   }//if(Cal)//
return(foo);
}

int ReadCalibration(char *cal_file, RoverData *Values)
{

// read calibration information from specified file
// must be in correct text file format
// ignores comments on a line after data

   int start;
   ifstream Cal(cal_file);
   char inbuff[81];
   int i,j;
   int foo = 0;
   int x,y,theta;

   if (Cal){
      foo = 1;
      Cal.getline(inbuff,81);
      if (strlen(inbuff)>0){
         istrstream ins(inbuff, strlen(inbuff));
         ins >> no_scan_lines;
      }// if(strlen(inbuff)>0) //
      else
         foo = 0;

      //read scan_pixels
      Cal.getline(inbuff,81);
      if (strlen(inbuff)>0){
```

```
   istrstream ins(inbuff, strlen(inbuff));
   ins >> Lscan_pixel[0]
       >> Lscan_pixel[1]
       >> Lscan_pixel[2];
}// if(strlen(inbuff)>0 //
else
   foo = 0;

Cal.getline(inbuff,81);
if (strlen(inbuff)>0){
   istrstream ins(inbuff, strlen(inbuff));
   ins >> Rscan_pixel[0]
       >> Rscan_pixel[1]
       >> Rscan_pixel[2];
}// if(strlen(inbuff)>0) //
else
   foo = 0;

for (i=0;i<3;i++){
   start = i*3;
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> LExpectedPixels[start]
          >> LExpectedPixels[start+1]
          >> LExpectedPixels[start+2];
   }// if(strlen(inbuff)>0) //
   else
      foo=0;
}//for(i=0;i<3;i++)//

for (i=0;i<3;i++){
   start = i*3;
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> RExpectedPixels[start]
          >> RExpectedPixels[start+1]
          >> RExpectedPixels[start+2];
   }
   else
      foo =0;
}
for (i=0;i<9;i++){
   start = i*6;
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      for (j=0;j<6;j++)
```

```
            ins >> LFuzzPoint[start+j];
      }
      else
         foo =0;
   }
   for (i=0;i<9;i++){
      start = i*6;
      Cal.getline(inbuff,81);
      if (strlen(inbuff)>0){
         istrstream ins(inbuff, strlen(inbuff));
         for (j=0;j<6;j++)
            ins >> RFuzzPoint[start+j];
      }
      else
         foo = 0;
   }//for(i=0;i<9;i++)//
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> save_data_flag;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> save_image_flag;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> simulation_flag;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> Values->dx;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> Values->dy;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
   if (strlen(inbuff)>0){
      istrstream ins(inbuff, strlen(inbuff));
      ins >> Values->theta;
   }// if(strlen(inbuff)>0) //
   Cal.getline(inbuff,81);
Cal.close();
}//if(Cal)//
```

```
return(foo);
}//ReadCalibration()//




void init_fuzzy(RoverData *Values){
// initialize all rover parameters
// allocated memory and set up
// frame grabber and ADAC board.
   if (Init_ADAC())
      exit(1);
   init_odx();

   Lscan_pixel = (int *)malloc(6*sizeof(int));
   Rscan_pixel = (int *)malloc(6*sizeof(int));
   RExpectedPixels = (int *)malloc(9*sizeof(int));
   LExpectedPixels = (int *)malloc(9*sizeof(int));
   RFuzzPoint      = (int *)malloc(54*sizeof(int));
   LFuzzPoint      = (int *)malloc(54*sizeof(int));
   if (!image)
      image = (unsigned char far *)farmalloc(ImageW*ImageH);
   if (!image) exit(1);
   if (!(Values->image[I_LEFT]))
      Values->image[I_LEFT] = (unsigned char far
*)farmalloc(ImageW*ImageH);
   if (!(Values->image[I_LEFT])) exit(1);
   if (!(Values->image[I_RIGHT]))
      Values->image[I_RIGHT] = (unsigned char far
*)farmalloc(ImageW*ImageH);
   if (!(Values->image[I_RIGHT])) exit(1);

   ReadCalibration(DEFAULT_CALIB, Values);
   if (simulation_flag){
      // load series number from read series file
      // and open series data file if it exists
      Series = LoadSeries(DEFAULT_READ);
      char data_file[80];
      if (Series < 10)
         sprintf(data_file,"%sd00%d.dat",DATA_PATH,Series);
      else if (Series < 100)
         sprintf(data_file,"%sd0%d.dat",DATA_PATH,Series);
      else
         sprintf(data_file,"%sd%d.dat",DATA_PATH,Series);
      sim_fp = fopen(data_file,"rb");
      if (sim_fp)
         ReadMap(Values, sim_fp); //read first map (always 0's)
```

```
   }else
      Series = LoadSeries(DEFAULT_SERIES);
   Values->x = 0;  //set initial location as origin
   Values->y = 0;
}

void cleanup_fuzzy(RoverData *Values)
{
// shut down everything cleanly before
// exiting program
//
   Lasers(LASER_OFF);
   farfree(image);
   farfree(Values->image[I_RIGHT]);
   farfree(Values->image[I_LEFT]);
   if (simulation_flag)
      fclose(sim_fp);
   free(Values->memvalues);
   free(Values);
}


int WriteImage(unsigned char far *Image, char *image_file){

//write an image that is smaller than 64K.
//this functin will have to be rewritten if
// images get bigger.

FILE *image_fp;
image_fp=fopen(image_file, "wb");
if (image_fp==NULL)
   return 0;
fwrite(Image, sizeof(unsigned char), ImageW*ImageH, image_fp);
fclose(image_fp);
return 1;
}


int ReadImage(unsigned char far *Image, char *image_file){

FILE *image_fp;

if ((image_fp=fopen(image_file, "rb"))==NULL)
   return 0;

fread(Image, sizeof(unsigned char), ImageW*ImageH, image_fp);
fclose(image_fp);
return 1;
```

```
}

int NameImage(char *image_file, int camera,char *diff){
// name image according to series and move numbers
//
   char  name[8];
   char  move_s[3];
   if (Series<10)
      sprintf(name,"00%d",Series);
   else if (Series <100)
      sprintf(name,"0%d",Series);
   else
      sprintf(name,"%d",Series);
   if (move_no <10)
      sprintf(move_s,"00%d",move_no);
   else if (move_no < 100)
      sprintf(move_s,"0%d",move_no);
   else
      sprintf(move_s,"%d",move_no);
   if (camera == C_RIGHT)
      sprintf(image_file,"%s%sR%s%s.img",IMAGE_PATH,name,
               move_s,diff);
   else
      sprintf(image_file,"%s%sL%s%s.img",IMAGE_PATH,name,
               move_s,diff);
return 1;
}

int LoadSeries(char *series_file){
   ifstream Cal(series_file);
   char inbuff[81];
   int flag=1;
   int series;

   if (Cal){
       while(flag){
      flag = 0;
      Cal.getline(inbuff,81);
      if (strlen(inbuff)>0){
         if (inbuff[0]=='#')
            flag=1;
         else{
            istrstream ins(inbuff, strlen(inbuff));
            ins >> series;
         }
      }// if(strlen(inbuff)>0) //
       }
       Cal.close();
   }
```

```
   else series = 0;
   if (!simulation_flag){
      ofstream Calo(series_file);
      if (Calo)
         Calo << (series+1);
      Calo.close();
   }
return series;
}

int WriteMap(RoverData *Values){
// WriteMap will append map data to a file
// named after the series_no
//
// 4/25/94
//
//Append map information to current series file
//
char data_file[30];
FILE *data_fp;
   if (Series < 10)
      sprintf(data_file,"%sd00%d.dat",DATA_PATH,Series);
   else if (Series < 100)
      sprintf(data_file,"%sd0%d.dat",DATA_PATH,Series);
   else
      sprintf(data_file,"%sd%d.dat",DATA_PATH,Series);

   data_fp = fopen(data_file,"ab");
   if (!data_fp)
      return 0;
   fwrite(Values->memvalues,sizeof(float),28,data_fp);
   fwrite(&(Values->direction), sizeof(int),1,data_fp);
   fwrite(&(Values->x),sizeof(float),1,data_fp);
   fwrite(&(Values->y),sizeof(float),1,data_fp);
   fwrite(&(Values->theta),sizeof(float),1,data_fp);
   fwrite(&(Values->dx),sizeof(float),1,data_fp);
   fwrite(&(Values->dy),sizeof(float),1,data_fp);
   fwrite(&(Values->dtheta),sizeof(float),1,data_fp);



   fclose(data_fp);
return 1;
}

int ReadMap(RoverData *Values, FILE *data_fp){
// ReadMap will read one data element from a file that
// is already open and in the correct position
//
```

```
// 4/25/94
//
int error = -34;
   if (data_fp){
      error += fread(Values->memvalues, sizeof(float),28,data_fp);
      error += fread(&(Values->direction), sizeof(int),1,data_fp);
      error += fread(&(Values->x), sizeof(float),1,data_fp);
      error += fread(&(Values->y), sizeof(float),1,data_fp);
      error += fread(&(Values->theta), sizeof(float),1,data_fp);
      error += fread(&(Values->dx), sizeof(float),1,data_fp);
      error += fread(&(Values->dy), sizeof(float),1,data_fp);
      error += fread(&(Values->dtheta), sizeof(float),1,data_fp);
   }

// will return a 1 if successful
return error;
}

int ScoreImage(unsigned char far *image1, unsigned char far *image2){

// dashes ten pixels long one line below each scanline
// row not currently used.

unsigned char far *p1;
unsigned char far *p2;

for (int j=0; j<3; j++){
   p1 = image1 + (long)(Lscan_pixel[j]*ImageW);
   p2 = image2 + (long)(Lscan_pixel[j]*ImageW);
   for(int i=0;i<ImageW;i++){
      if (((i/10) % 2)==0)
         *p1 = 200;
      else
         *p1 = 50;
      *p2 =   0;
      p1++;
      p2++;
   }
}
return 1;
}
```

```
// file: Filter.cpp
// written by: Tim McJunkin
//
// Copyright 1994 Utah State University
//
// last modified July 18, 1994

#include <fuzz_alg.h>

void disp_text(char *);

int FindPixels(int *Pixel, int row, int Camera, unsigned char far
*Image,
         unsigned char far *Img_off)
//         This Function filters the image row
//         for the Pixel Locations of the Center and
//         two onside lasers


{
   int i,j,start;
   int n=0; //number of pixels found
   float filter, ave;
   float max_filter=60;
   float min_thresh=250;
   unsigned char far *pp;
   unsigned char far *p2;
   unsigned char far *pstart;
   unsigned char far *pstart2;

   for (i=0;i<6;i++) Pixel[i] = 0;  // reset Pixel locations

   switch (Camera){
      case C_RIGHT:
         if ((row < ImageH)&&
             (row >= 0  )){
            // start at right edge of image
            // for right camera.  In this way
            // pixel position will always be
            // from the outside edge and make
            // both right and left calculations
            // the same.  i.e. possitive pixel
            // deviation will indicate a protruding
            // obstacle.
            pstart = Image +
                (long)row*ImageW+299L;
            pstart2 = Img_off +
                (long)row*ImageW+299L;
            start = 0;
```

```
         }
         break;
      case C_LEFT:
         if ((row < ImageH)&&
             (row >= 0  )){
            pstart = Image + (long)row*ImageW;
            pstart2= Img_off + (long)row*ImageW;
            start = 0;
         }
         break;
      default:
         exit(0);
      }

   // start line in a bit to avoid offside lasers
   pp=pstart;
   p2=pstart2;
   max_filter = 0;

   min_thresh = 100; //minimum filter value for detection of laser
                     //set to 100 experimentally
   max_filter=0.0;
   int repeat = 0;
   n=0;
   while ((n<4) && (min_thresh > 49) && (repeat<10)){
      repeat++;
      pp = pstart;
      p2 = pstart2;
      n = 0;
      for (i=start; (i < (start+280))&&(n<5); i++){

         if (Camera == C_LEFT)
            filter = pix_filt_dif(pp++,p2++);
         else
            filter = pix_filt_dif(pp--,p2--);


         if (filter>min_thresh){
            //filter is above minimum threshold
            if (filter > max_filter){
               max_filter = filter;
               Pixel[n] = i+10; //add ten so pixel location
                                //is the center of
the filter
            }
         }else{
               if (max_filter > 0.0){ //if a laser has been found and
filter
                  n++;               //drops below threshold start
```

```
looking
                    Pixel[n] = 0;       //for next laser
                 }
              max_filter = 0.0;
              }


          }
          min_thresh -= 10;//reduce threshold if
              //too few lasers are found
              //have to rethink this when
              //we start looking for dips
              //as well as protrusions.
   }


   for (i=n;i<6;i++) Pixel[i] = 0; //set all other pixels to 0.

      return(n);
}


#if 0
//experimental function for finding pixel locations
//not working does not compile
int FindPixelsNew(int *Pixel, int scanline, int Camera, unsigned char
far *Image,
         unsigned char far *Img_off)

{
   int i,j;
   int row;
   int expect=0;
   int next_expect=0;
   float max_filter;
   float thresh;
   float filt;
   unsigned char far *pp;
   unsigned char far *pstart;
   unsigned char far *p2;
   unsigned char far *pstart2;

   switch (Camera){
      case C_RIGHT:
          row = Rscan_pixel[scanline];
          for (i=0;i<3;i++){
        Pixel[i] = 0;
        max_filter = 0.0;
        if (i>0 && Pixel[i-1] > (next_expect-5))
```

```
      expect = Pixel[i-1] + 15;
   else
      expect = next_expect;
   if (i==2)
      next_expect = expect + 100;
   else
      next_expect = RExpectedPixels[scanline*3+i+1];
   pp = Image + (long)row*ImageW + ImageW - expect;
   p2 = Img_off+(long)row*ImageW + ImageW - expect;
   pstart2 = p2;
   for (j=(expect-10);j<(next_expect+10);j++){
      filt = pix_filter(p2--);
      if (filt>max_filter)
         max_filter=filt;
   }
   thresh=max_filter;
   max_filter = 0;
   p2 = pstart2;

   for (j=(expect-10);j<(next_expect +10);j++){
       if (max_filter < thresh || j < next_expect){

      filt = pix_filt_dif(pp--,p2--);

      if (filt > max_filter){
         max_filter = filt;
         Pixel[i] = j;
      }
       }
   }
    }
    break;

case C_LEFT:
    row = Lscan_pixel[scanline];
    for (i=0;i<3;i++){
   Pixel[i] = 0;
   max_filter = 0.0;
   if (i>0 && Pixel[i-1] > (next_expect - 5))
      expect = Pixel[i-1] + 15;
   else
      expect = next_expect;
   if (i==2)
      next_expect = expect + 100;
   else
      next_expect = LExpectedPixels[scanline*3+i+1];
   pp = Image + (long)row*ImageW + (long)expect - 20L;
   p2 = Img_off+(long)row*ImageW + (long)expect - 20L;
   pstart2 = p2;
```

```
         for (j=(expect-10);j<(next_expect+10);j++){
            filt = pix_filter(p2--);
            if (filt>max_filter)
               max_filter=filt;
         }
         thresh=max_filter;
         max_filter = 0;
         p2 = pstart2;

         for (j=(expect-10);j<(next_expect +10);j++){
             if (max_filter < thresh || j < next_expect){

            filt = pix_filt_dif(pp++,p2++);

            if (filt > max_filter){
               max_filter = filt;
               if (filt > thresh);
                  Pixel[i] = j;
            }
             }
         }
          }
          break;
   }
return(1);
}
#endif




//non differenced filter
float pix_filter(unsigned char far* center_pixel){
   unsigned char far *pp = center_pixel;
   float filt;

   filt = -1.0*pp[0] -1.0*pp[1]
        //+0.0*pp[3] +0.0*pp[4] +0.0*pp[5]
        +4.0*pp[10]
        //+0.0*pp[7] +0.0*pp[8] +0.0*pp[9]
        -1.0*pp[19]-1.0*pp[20];
return(filt);
}

// differenced filter
float pix_filt_dif(unsigned char far* center_pixel,
         unsigned char far* cen_off){
   float filt;
```

```
   unsigned char far *pp = center_pixel;
   unsigned char far *p2 = cen_off;

   filt = -1.0*abs((int)(pp[0]-p2[0])) -1.0*abs((int)(pp[1]-p2[1]))
      //        +0.0*pp[3] +0.0*pp[4] +0.0*pp[5]
        +4.0*abs((int)(pp[10]-p2[10]))
      //        +0.0*pp[7] +0.0*pp[8] +0.0*pp[9]
        -1.0*abs((int)(pp[19]-p2[19]))-1.0*abs((int)(pp[20]-p2[20]));
return(filt);
}
```

```
// file: rov_adac.c
// this file will contain all calls
// associated with the ADAC board.
// This includes the laser on/off
// calls and rover movement commands
//
// written by: Kurt Olsen and Tim McJunkin
// Center for Self Organizing and Intelligent
// Systems
//
// Copyright 1994 Utah State University
//
// date: April 21, 1994
//
#include <stdlib.h>
#include <alloc.h>
#include <time.h>
#include <dos.h>
#include <fuzz_def.h>
#include <adliborl.h>
#include <adac.h>
#include <gs.h>
#include "odx.h"
#include "opr.h"
int     Init_ADAC();
void    Lasers(int status);
int     MoveRover(int dir);
int     TakePicture(int Camera, unsigned char far *Image);
void    set_turn(int value);
void    set_move(int value);
void    nap(float ticks);
int     ActCamera(int Camera);
void    GetImage(unsigned char far *Image);
//extern        "C" {BOOL AcquireTM(unsigned int);}
void get_fb(unsigned char far *img);
int init_odx(void);
void WriteTextMessage(char *, ... );

//Frame Grabber Global Variable
extern BYTE     bawl,vhld,vstart,hlf;
BYTE            vhld1,vstart1,hlf1;
unsigned image_seg, image_off;

//Laser/Camera state holders
int camera_state;
int laser_state;
```

```
//Init_ADAC() reads configuration file
// and initializes ADAC board to benign
// states
// 4/21/94 TMc
int Init_ADAC(){
   int foo;

   if (foo=init(CONFIG_FILE))
      exit(1);

   {
   int buf[1];

   buf[0] = MOVE_NEUTRAL;
   aot((int far *)buf, 1 /* count */, MOVE_CHANNEL, 1 /* unit */);
   buf[0] = TURN_NEUTRAL;
   aot((int far *)buf, 1, TURN_CHANNEL, 1);
   buf[0] = 0;
   dot((int far *)buf, 1, 0, 1);
   }
return foo; // should only return if foo == 0 or should exit
}

//Lasers(status) sets state of ADAC board switch
// controlling lasers
// Lasers(LASER_ON); or
// Lasers(LASER_OFF);
//
// keeps track of camera status so it is not
// switched
//
void Lasers(int status){
   laser_state=status;

   {//ADAC
      int buf[1];
      buf[0] = LASER_BIT * laser_state
          | CAMERA_BIT * camera_state;
      dot((int far *)buf, 1, 0, 1);
   }//ADAC
   nap(.1);
}//Lasers

//
//MoveRover(direction) accepts direction and controls
// ADAC board appropriately with delays defined in ticks
// in fuzz_def.h
//
int MoveRover(int dir)
```

```
{
  switch (dir)
    {
    case D_FORWARD:
           set_move(MOVE_FORWARD);
           nap(MOVE_HOLD);
           set_move(MOVE_NEUTRAL);
           break;
    case D_BACKWARD: set_move(MOVE_BACKWARD);
           nap(MOVE_HOLD);
           set_move(MOVE_NEUTRAL);
           break;
    case D_LEFT:
           set_move(MOVE_FORWARD);
           nap(.25);
           set_turn(TURN_LEFT);
           nap(MOVE_HOLD);
           set_turn(TURN_NEUTRAL + 200);
           nap(TURN_LEAD - .5);
           set_turn(TURN_NEUTRAL);
           nap(.25);
           set_move(MOVE_NEUTRAL);

           break;
    case D_RIGHT:
           set_move(MOVE_FORWARD);
           nap(.25);
           set_turn(TURN_RIGHT);
           nap(MOVE_HOLD);
           set_turn(TURN_NEUTRAL - 150);
           nap(TURN_LEAD - .5);
           set_turn(TURN_NEUTRAL);
           nap(.25);
           set_move(MOVE_NEUTRAL);
           break;
    }
    nap(.1);
  return 0;
}

//
//TakePicture(Camera, image)  instructs system
// to switch to appropriate Camera and calls
// imaging routines
//
int TakePicture(int Camera, unsigned char far *Image){
   ActCamera(Camera);
   get_fb(Image);
return (1);
```

```
}


void set_turn(int value)
{
//  char cbuf[32];
//  sprintf(cbuf,"Set_turn(%d)", i);
//  disp_text(cbuf);
   int buff[1];

   buff[0] = value;

   aot((int far *)buff, 1 /* count */, TURN_CHANNEL, 1 /* unit */);

}

void set_move(int value)
{
//  char buf[32];
//  sprintf(buf,"set_move(%d)", i);
//  disp_text(buf);
   int buff[1];

   buff[0] = value;

   aot((int far *)buff, 1 /* count */, MOVE_CHANNEL, 1 /* unit */);

}

void nap(float ticks)
{
  clock_t now;

  now = clock();
#ifndef DEBUG
  while (((float)(clock() - now))/CLK_TCK < ticks)
    ;
#endif
}


int ActCamera(int Camera)
{
   // code to switch ADAC bored
   // to activate correct Camera on
   // the rover

   switch (Camera){
      case C_RIGHT:
```

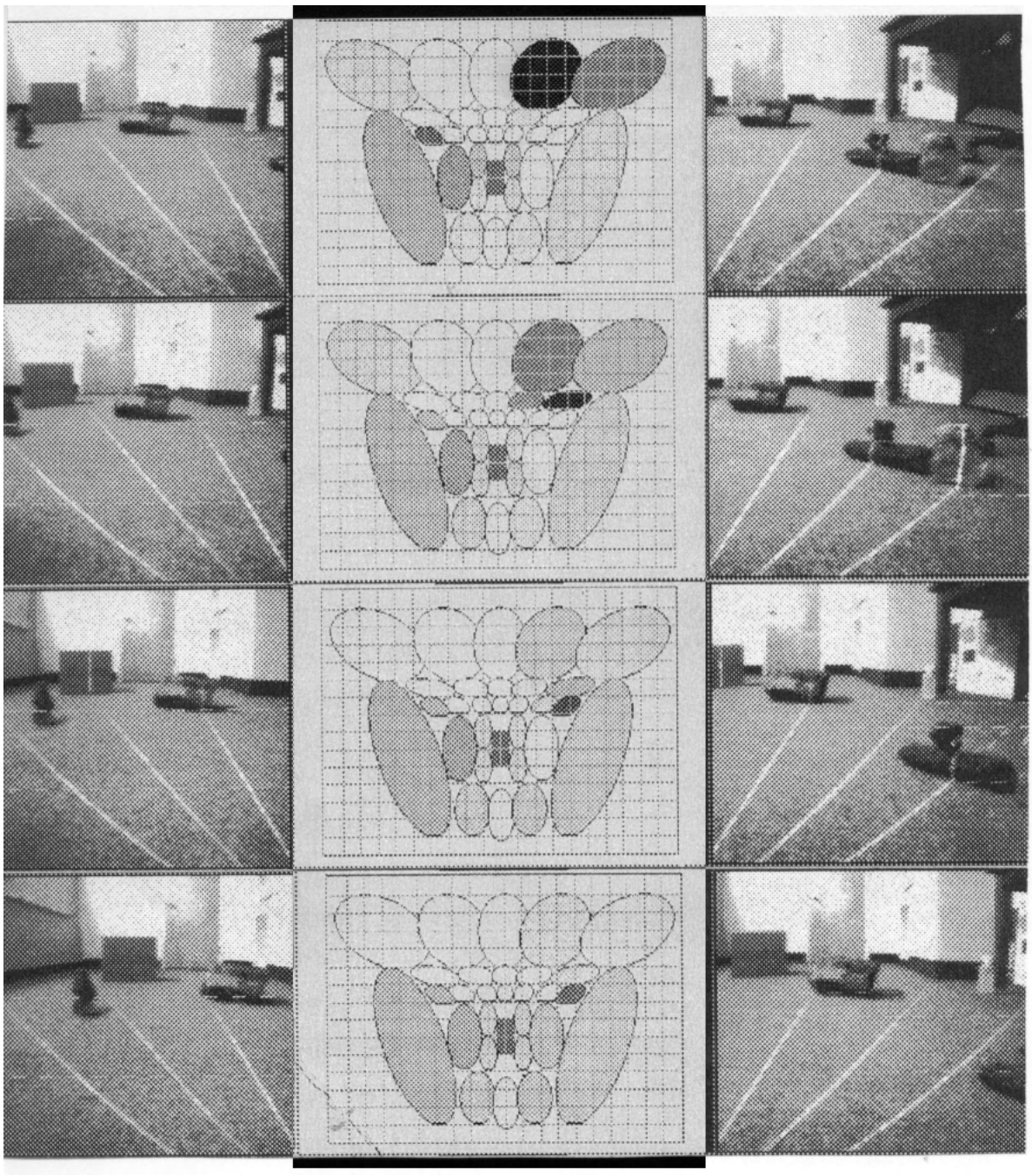
```
            camera_state=I_RIGHT;
            break;
        case C_LEFT:
            camera_state=I_LEFT;
            break;
        default:
            // oops
            exit(0);
            }
    {// ADAC
        int buf[1];
        buf[0] = LASER_BIT * laser_state
            | CAMERA_BIT * camera_state;
        dot((int far *)buf, 1, 0, 1);
    }//ADAC
    nap(.125);  //was .5 7/14/94
return(1);
}

//obsolete
#if 0
void GetImage(unsigned char far *Image)
{
    int                             vs,vh,hl;

    //get single image from boared to Image
        vstart1 = 0x27;
        vstart = 0x27;
        vhld1 = 0x82;
        vhld = 0x82;
        hlf1 = 0x67;
        hlf = 0x67;
        vs = vstart1;
        vh = vhld1;
        hl = hlf1;

image_seg = FP_SEG(Image); // find image semgment and
image_off = FP_OFF(Image); // offset for AcquireTM
AcquireTM(FRAME_PORT);
return;
}
#endif

int init_odx(void)
{
  int rval;

  rval = odxbind();
  setdmajor(0);
```

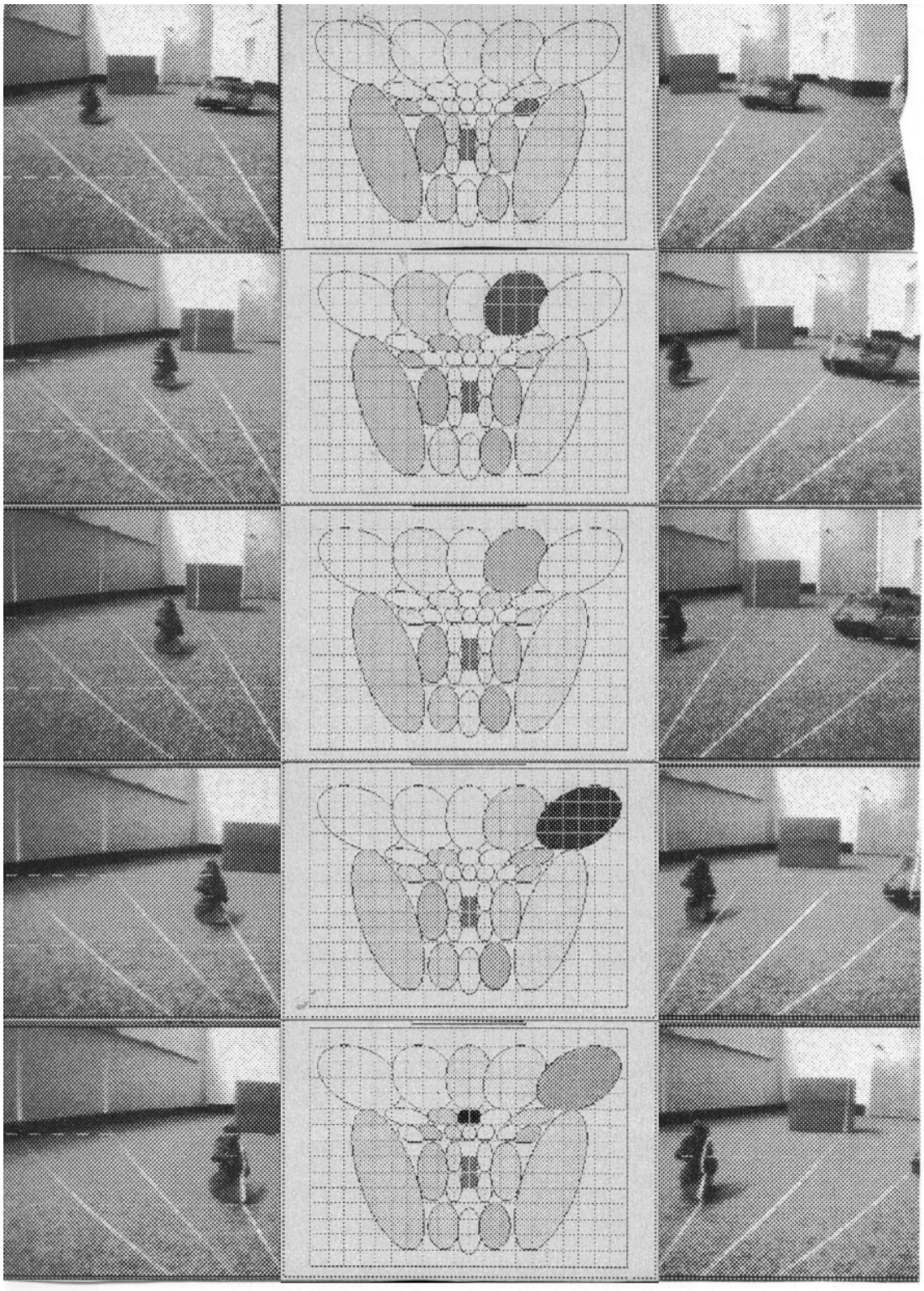
```
  reset();
}

void get_fb(unsigned char far *img)
{
  long y;
  int ps;
  int fb;
  unsigned char far *f1;
  unsigned char far *f2;
  static unsigned char far *temp=NULL;

  if (!temp)
      temp = (unsigned char far*)farmalloc(64000L);

  opr_set(PSTEP, 2);
  ps = opr_inq(PIXSIZ);

  opr_set(DFB, 0);
  opr_set(FFB, 0);
  opr_set(PFB, 0);

  pwin(0,60,640,100);
  dwin(0,0,640,480);
  fwin(0,0,640,480);

  fbgrab(5);
  while(opr_inq(FGON));

  rdbuf(img, 640);

  f1 = img;
  f2 = img;

  for (y=0; y < 32000L; y++)
    {
    *f1 = *f2;
    f1++;
    f2+= 2;
    }

  pwin(0,260,640,100);

  rdbuf(temp, 640);
  f2=temp;
  for (y=0; y < 32000L; y++)
    {
    *f1 = *f2;
    f1++;
```

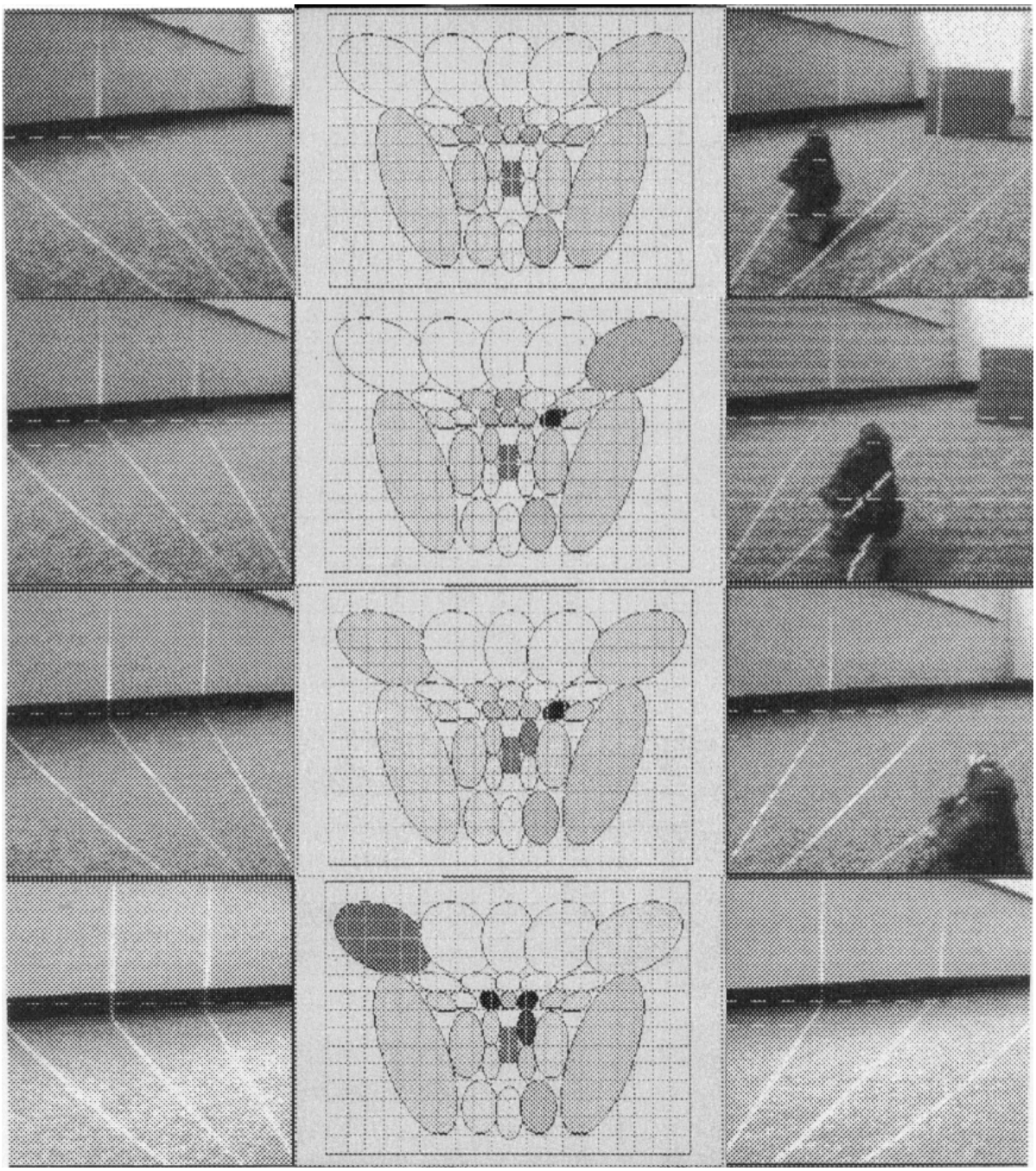
```
    f2+= 2;
    }
//  free(temp);
}
```

# APPENDIX C. ADDITIONAL RESULTS

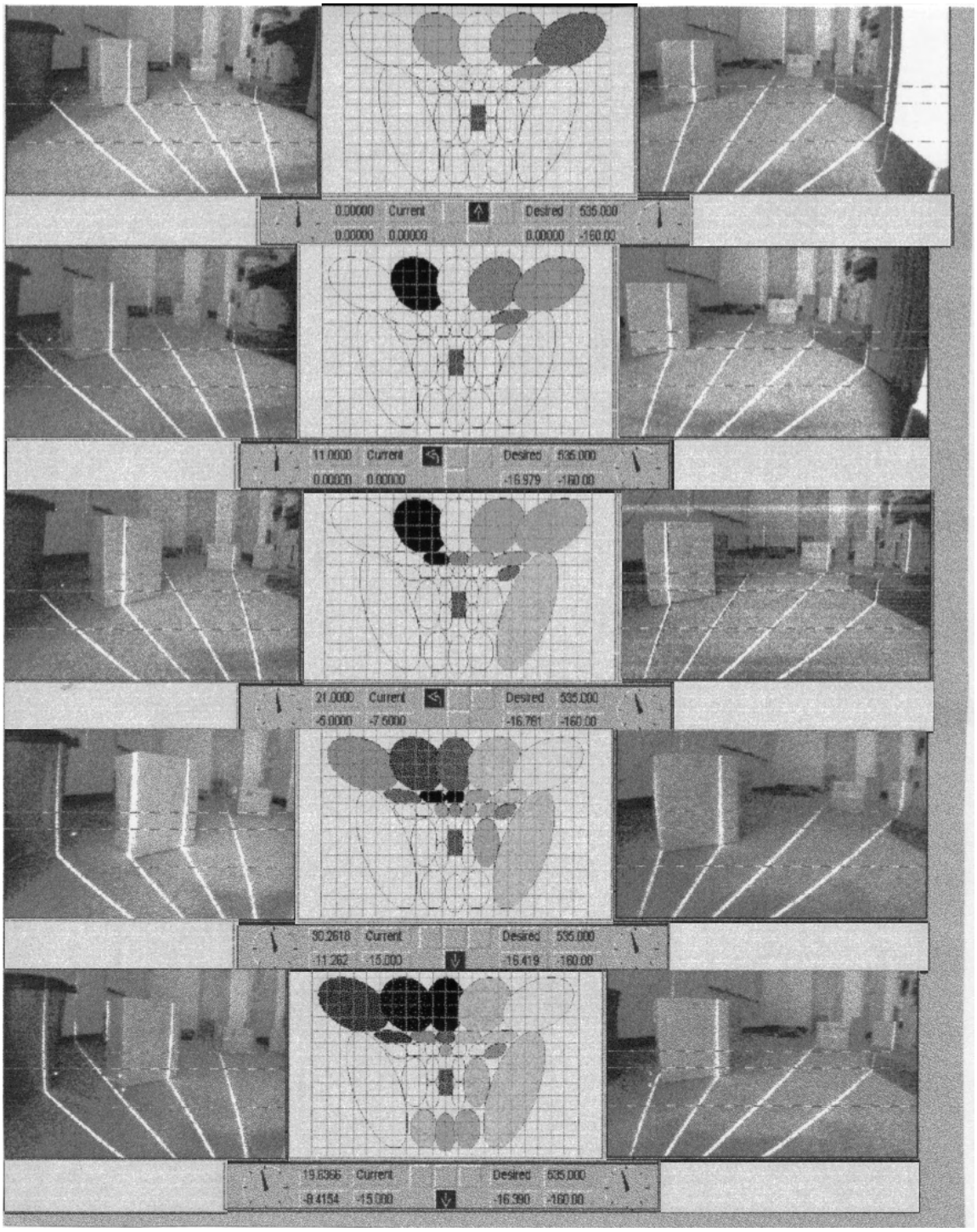
The following is a run by a less experienced driver. This run ends abruptly when the lighting in the final frame washes out the lasers and the map shows obstacles where there are none.

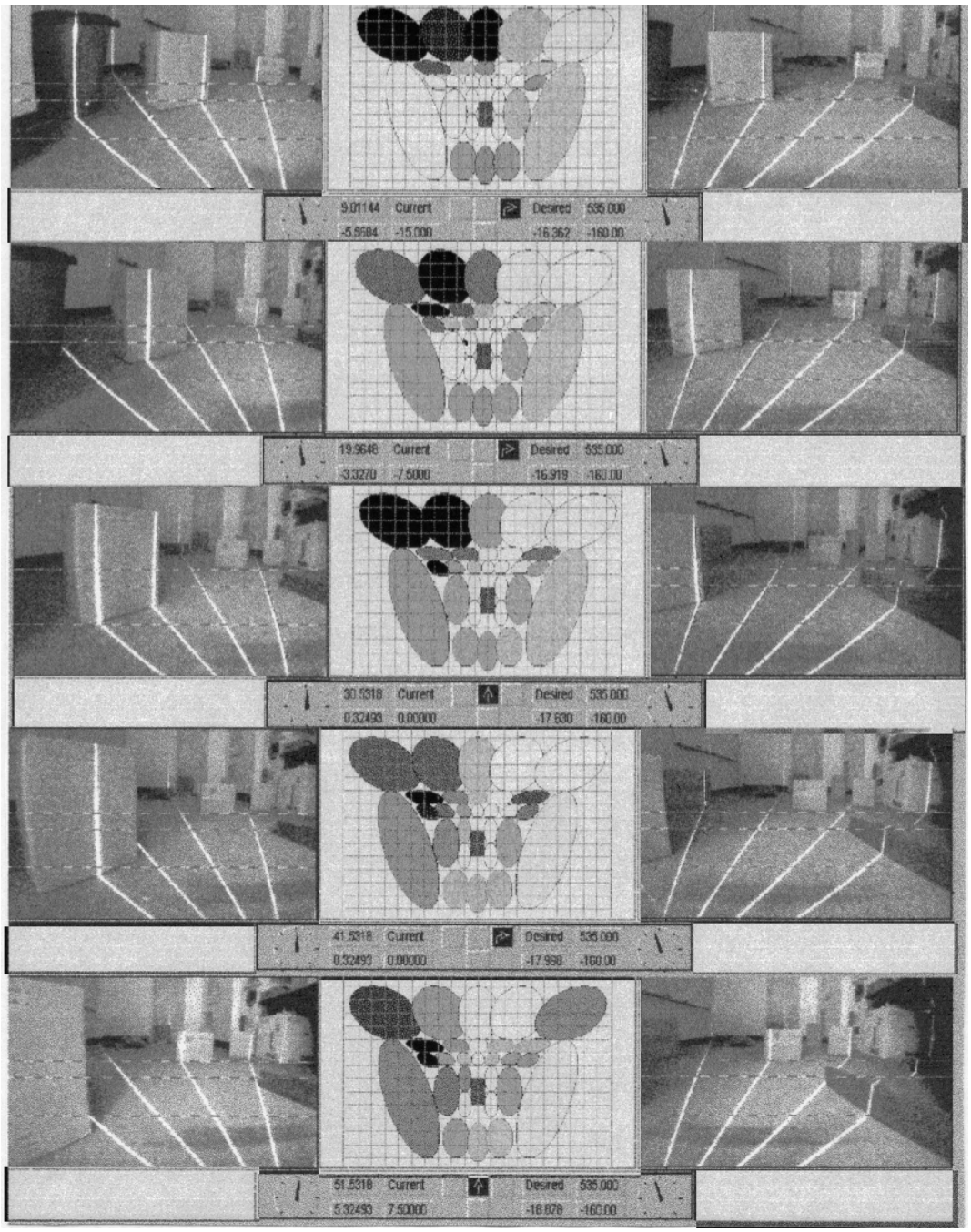


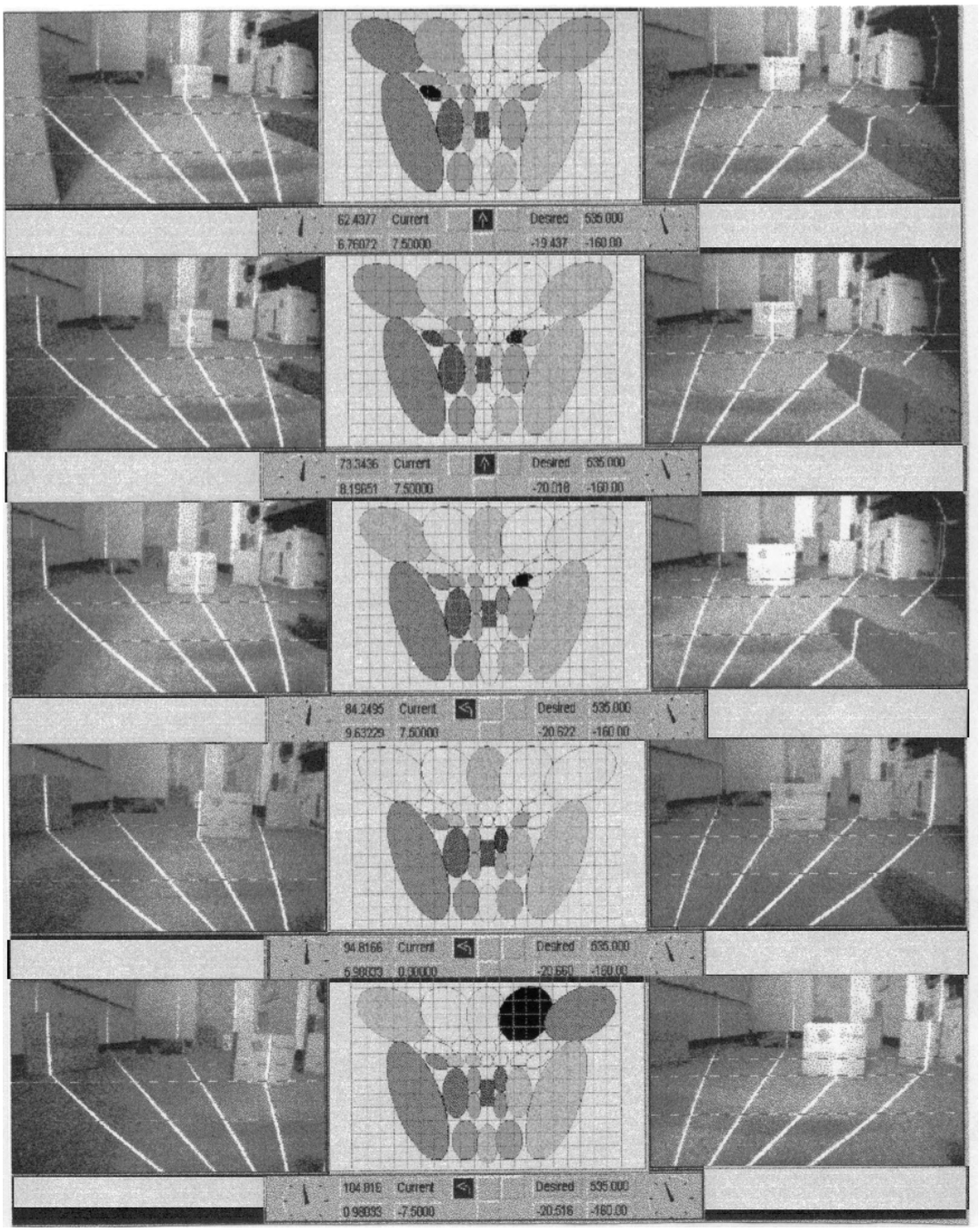


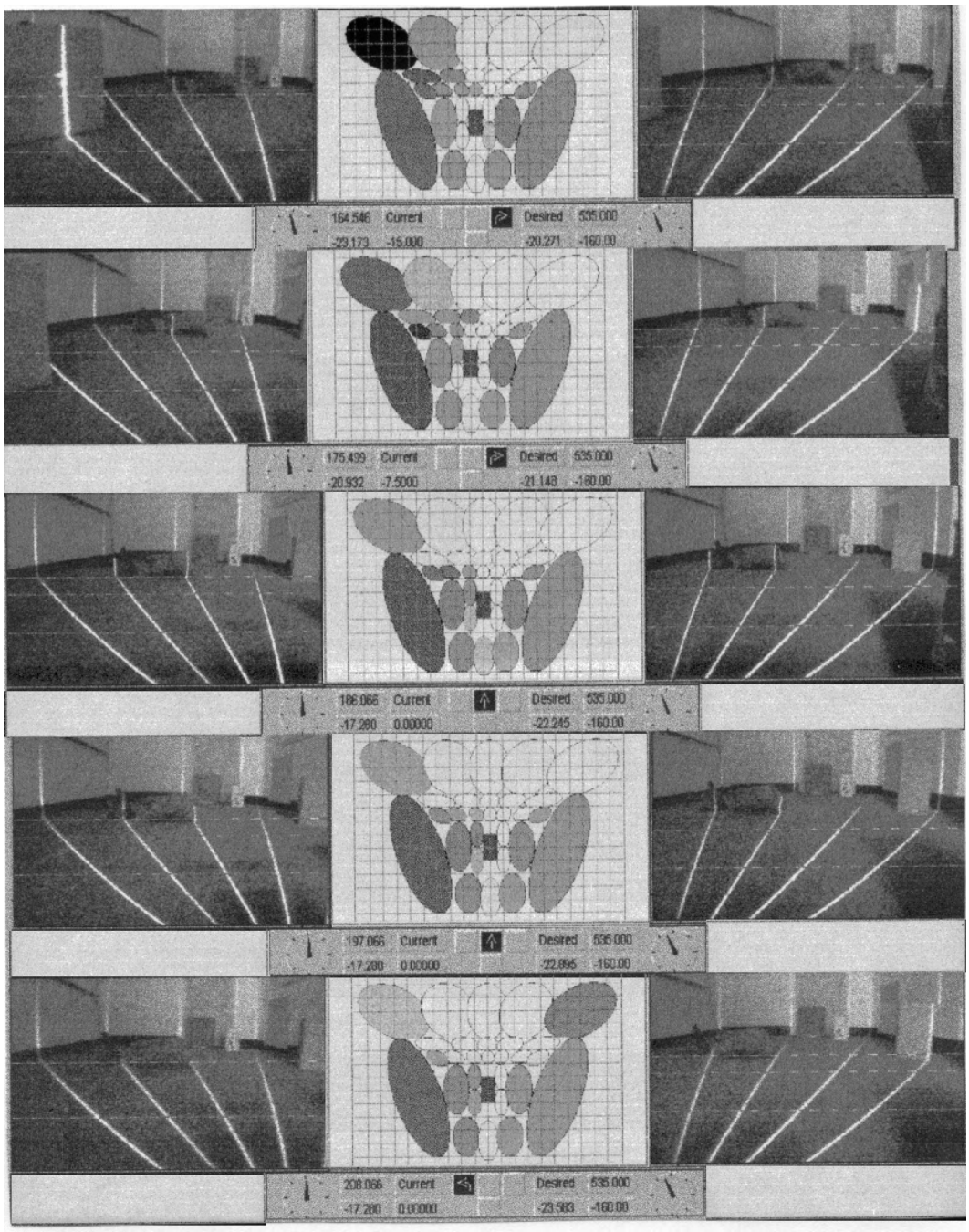


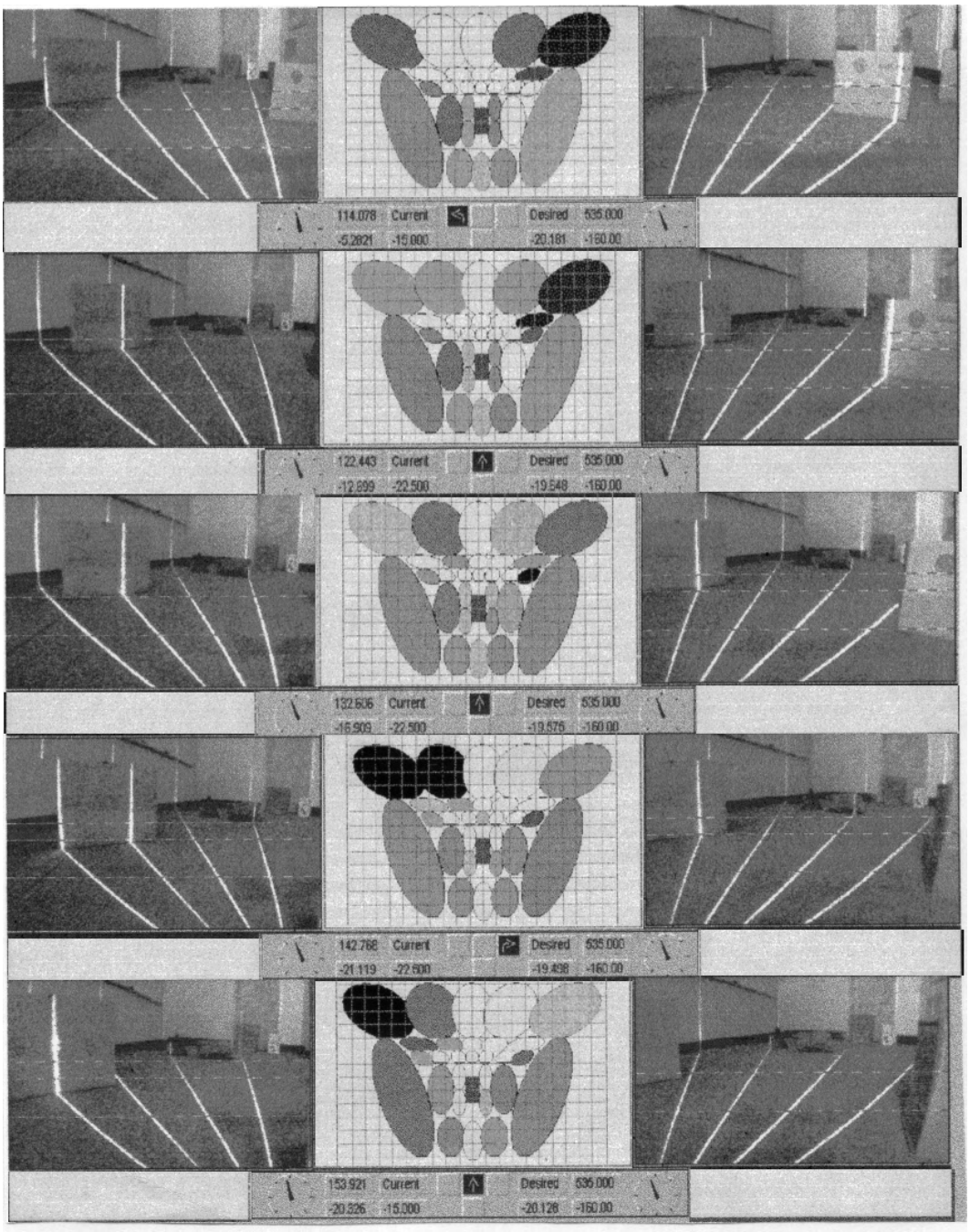
This run shows a complex case with a more experienced driver. The driver finds a reasonable path using only the map.




9.01144 Current
-5.5684 -15.000
Desired 535.000
-16.362 -160.00
19.9648 Current
-3.3270 -7.5000
Desired 535.000
-16.919 -160.00
30.5318 Current
0.32493 0.00000
Desired 535.000
-17.630 -160.00
41.5318 Current
0.32493 0.00000
Desired 535.000
-17.990 -160.00
51.5318 Current
5.32493 7.50000
Desired 535.000
-18.878 -160.00


62.4377 Current
6.76072 7.50000
Desired 535.000
-19.437 -160.00
73.3436 Current
8.19651 7.50000
Desired 535.000
-20.018 -160.00
84.2495 Current
9.63229 7.50000
Desired 535.000
-20.622 -160.00
94.8166 Current
6.98033 0.00000
Desired 535.000
-20.650 -160.00
104.816 Current
0.98033 -7.5000
Desired 535.000
-20.516 -160.00


164.546 Current
-23.173 -15.000
Desired 535.000
-20.271 -160.00
175.499 Current
-20.532 -7.5000
Desired 535.000
-21.148 -160.00
186.066 Current
-17.280 0.00000
Desired 535.000
-22.245 -160.00
197.066 Current
-17.280 0.00000
Desired 535.000
-22.895 -160.00
208.066 Current
-17.280 0.00000
Desired 535.000
-23.583 -160.00


114.078 Current
-5.2821 -15.000
Desired 535.000
-20.181 -160.00
122.443 Current
-12.899 -22.500
Desired 535.000
-19.548 -160.00
132.636 Current
-16.909 -22.500
Desired 535.000
-19.575 -160.00
142.768 Current
-21.119 -22.500
Desired 535.000
-19.498 -160.00
153.921 Current
-20.326 -15.000
Desired 535.000
-20.126 -160.00

# ADVANCED NAVIGATION FOR PLANETARY VEHICLES APPLYING AN APPROXIMATE MAPPING TECHNIQUE

by

Timothy R. McJunkin

A thesis submitted in partial fulfillment
of the requirements for the degree

of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

Approved:

| | |
|---|---|
| Dr. Robert W. Gunderson<br>Major Professor | Dr. Kay D. Baker<br>Committee Member |
| Dr. Todd K. Moon<br>Committee Member | Dr. James P. Shaver<br>Dean of Graduate Studies |

UTAH STATE UNIVERSITY
Logan, Utah

1994