Team 6
Manohar Darapureddi
Pedro Dioz
Joe Garfield
Jianwei Zhuang

Project 2 Robot Code Document

I. Introduction

The basic strategy for this project could be expressed in the following finite state automata (Figure 1).

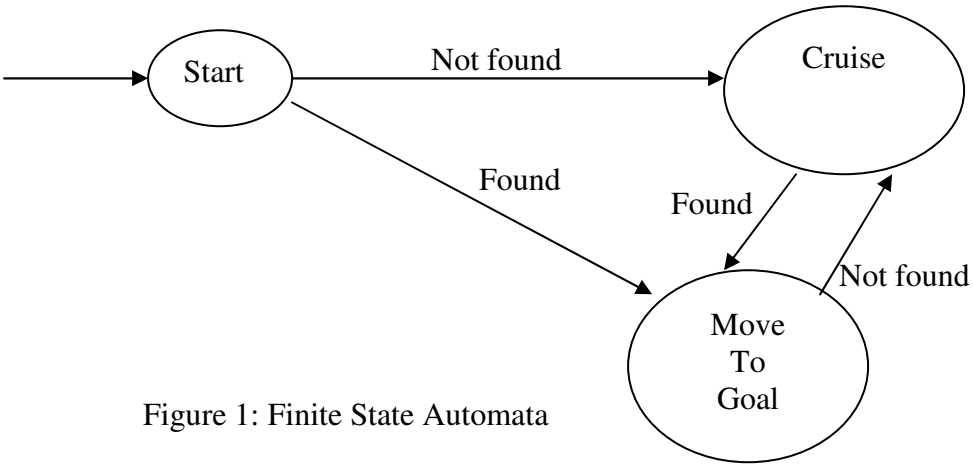

Figure 1: Finite State Automata

The robot started by searching any target that was close enough to it. If a target could be found, it would turn to the target and start moving toward it. Otherwise, it would cruise until it detected a target. During the whole process, the robot would avoid itself from any contact with rocks or hazard objects.

The architecture of the software was based on Schema Theory. It included four main behaviors: **cruise, move to goal, avoid rocks, and avoid hazard objects**. Each behavior consisted of one perceptual schema and one motor schema except for cruise, which only had motor schema (driving forward).

II. Behaviors

1. Move to goal behavior

a. Perceptual Schema: check_light ( )

The robot had four light sensors: two pointing to the front (left front and right front), one pointing to the left, and one pointing to the right. Three variables were set to sense the direction of the target:

- **target_found**: if any reading was lower than a pre-defined value (effective detecting distance), this variable would be set to 1, which meant the target was found by one of the light sensors
- **min**: the minimum reading value of the four sensors
- **direction**: the direction of the minimum reading (part 1, 2, 3 in Figure 2)

Another variable (**delta**) was the difference between two front sensors.

b. Motor Schema: move_to_goal ( )

We divided the view of the robot into three parts (see Figure 2). Only when the **target_found** had been set to 1, the robot would do actions according to the values of other variables.

If the minimum reading was in part 1 or 3, the robot would keep turning left or right until the target entered part 2 (in front of the robot). A **timeout** of 3 seconds was set in order to prevent the robot from turning infinitely in case that the target was blocked when it was turning or simply out of the effective detecting distance.

If the minimum reading was in part 2, delta would be used to adjust the driving direction on its way to the target. A pre-defined value (**delta_thresh**) was used for the sensitivity of the adjustment.

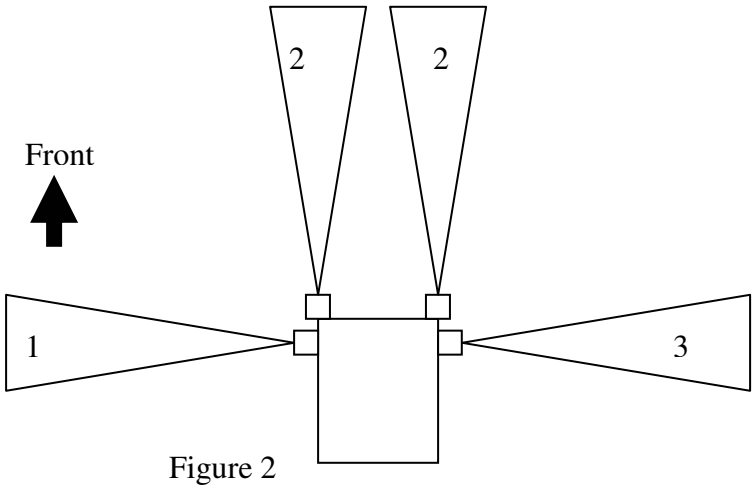

Figure 2

2. Avoid rocks

a. Perceptual schema: check_bumper ( )

Two variables for two touch sensors were used to decide if there was any rock ahead, on the right or the left.

- **left_rock**: set to 1 if left sensor was on
- **right_rock**: set to 1 if right sensor was on

b. Motor schema: avoid_rock ( )

If **left_rock** and **right_rock** were set to 1, the robot would go backward and then take a random turn, either left or right. The turning angle was also random: the sleeping time ranged from 0.3 to 0.9 second. If the **left_rock** were set, the robot would turn to the right; if the **right_rock** were set, turn to the left.

Two counters, **i** and **j**, were used to take a random turn every fourth time when **left_rock** or **right_rock** was set. This was intended to get the robot out of some box canyon situations.

3. Avoid hazard objects

a. Perceptual schema: check_hazobj ( )

This function worked similar to the **check_bumper** except that there was a pre-defined value (**hazobj_min**) used to decide the effective detecting distance to hazard objects.

- **left_haz**: set to 1 if left range finder was within the effective distance
- **right_haz**: set to 1 if right range finder was within the effective distance

Another variable (**hazobj_delta**) was the difference between the readings of two range finders.

b. Motor schema: avoid_hazobj ( )

If both **left_haz** and **right_haz** were set, the robot would take a turn and the direction would depend on the **hazobj_delta**. Otherwise, if only **left_haz** or **right_haz** was set, the robot would take a right or left turn, respectively. Turning angles were random angles similar to **avoid_rock**.

4. Flag variables (avoid_rock_flag, avoid_hazobj_flag, move_to_goal_flag)

The robot wouldn't cruise if it was driving backward or taking turns. The purpose of using these flag variables was to disable/enable the cruise function (driving forward) when the robot was avoiding rocks/hazard objects or deciding which direction should turn.

III. Problems and solutions

The robot run pretty well as we expected. Many problems, such as being stuck by rocks, cable, and so on, mainly because of the construction of robot's bumper. Other software problems were solved during four demos:

1. The balance between attractive force from the target and repulsive forces from obstacles: in order to get the robot out of local minima, in which the summation of forces would cause the robot to keep doing some repetitive things in a very small area

and couldn't move any closer toward the target. Our solution was to use random turning angles, random turning directions, and smaller effective detection distance.

Another very important solution was to make sure the sleeping time after the adjustment of **move_to_goal** was large enough to give the robot enough time to complete the action of avoiding rocks. We met this case in our first demo.

2. Prevent robot from touching hazard object after it turned off the light and moved backward. A similar case would happen when robot was walking between two or more hazard objects. Our solution was to decrease the sleeping time for backward but still keep it large enough so that the robot wouldn't hit hazard object when it turned. It turned out the robot could walk among two hazard objects and some other obstacles that were pretty close to each other, and didn't touch any hazard objects.

IV. Attachment

- **source code (project2.ic)**

/* Project 2 Source Code project2.ic */

```
//Port numbers
#define lleft_light 3
#define left_light 4
#define rright_light 6
#define right_light 5
#define left_range 16
#define right_range 18
#define left_bumper 7
#define right_bumper 8
#define left_motor 0
#define right_motor 2

//flag variables
int avoid_hazobj_flag=0;
int avoid_rock_flag=0;
int move_to_goal_flag=0;

//Motor control behaviors
void forward()
{
   motor(left_motor,100);
   motor(right_motor,100);
}

void left_turn()
{
   motor(left_motor,-100);
   motor(right_motor,100);
}

void right_turn()
{
   motor(left_motor,100);
   motor(right_motor,-100);
}

void left_arc()
{
   motor(left_motor,-50);
   motor(right_motor,100);
}

void right_arc()
{
   motor(left_motor,100);
```

```
   motor(right_motor,-50);
}

void random_turn()
{
   int dir = random(2);

   if (dir) left_turn();
   else right_turn();
}

void backward()
{
   motor(left_motor,-100);
   motor(right_motor,-100);
}

//Light sensor routine
#define target_far 30            // this value decides the highest possible value of the light
int left_target=0;
int lleft_target=0;
int right_target=0;
int rright_target=0;
int target_found=0;
int min;
int direction=0;
int delta;                       // diff between two front light sensors

void check_light() {
   left_target = analog(left_light);
   right_target = analog(right_light);
   lleft_target =  analog(lleft_light);
   rright_target = analog(rright_light);

   if (left_target < target_far || right_target < target_far
      || lleft_target < target_far || rright_target < target_far) {
      target_found = 1;    //  target found

      // find the minimum reading of four light sensors
      //   and decide which direction it is on
      min = lleft_target;
      direction=1;

      if (left_target < min) {
         min = left_target;
         direction=2;
         delta = left_target-right_target;
```

```
        }
        if (right_target < min) {
            min = right_target;
            direction=2;
            delta = left_target-right_target;
        }
        if (rright_target < min) {
            min = rright_target;
            direction = 3;
        }
    }
    else target_found = 0;  // can't find
}

// check bumper sensors
int left_rock=0;
int right_rock=0;

void check_bumper() {
    if (digital(left_bumper)){
        left_rock=1;
    }
    else left_rock=0;

    if (digital(right_bumper)){
        right_rock=1;
    }
    else right_rock=0;
}

//Hazard object sensor constants and variables
#define hazobj_min 65        // within this range means haz obj is close (effective distance)
int left_haz=0;
int right_haz=0;
int hazobj_delta;            // the diff of two range finders if both are within the hazobj_min
int left_dist=0;
int right_dist=0;

void check_hazobj() {
    left_dist=analog(left_range);
    right_dist=analog(right_range);

    // check the diff of two rangefinders
    hazobj_delta = left_dist - right_dist;

    // decide the direction of the haz obj and which turn to take
    if (left_dist > hazobj_min) {
```

```
      left_haz = 1;
   }
   else left_haz = 0;

   if (right_dist > hazobj_min) {
      right_haz = 1;
   }
   else right_haz = 0;
}

// move to the light
#define goal_tick 0.3
#define buff_tick 0.7                    // this tick is to get out of local minima cases
#define delta_thresh 20                  // value to fine-tune the moving toward the target

void move_to_goal()
{
   long t_start,t_end;

   while(1) {
      check_light();

      if(target_found) {                 // If the target is found then check which turn
should take
         move_to_goal_flag = 1;
         check_light();

         if (direction == 2) {           // the target is on course
            if (delta < -delta_thresh) {    // use the delta_thresh to fine-tune the moving
               left_arc();
               sleep(goal_tick);
            }
            else if (delta > delta_thresh) {
               right_arc();
               sleep(goal_tick);
            }
         }
         else if (direction == 1) {      // the target is on the left
            left_turn();                 //   turn to the left
            t_start = mseconds();

            while(direction != 2) {      // stop turning when the target is head on
               check_light();
               t_end = mseconds();
               if ((t_end - t_start) > 3000L) break;    // 3 sec timeout, avoid infinite loop
            }
          }
```

```
            else if(direction == 3) {          // the target is on the right
                right_turn();                  //   turn to the right
                t_start = mseconds();

                while(direction !=2) {         // stop turning when the target is head on
                   check_light();
                   t_end = mseconds();
                   if ((t_end - t_start) > 3000L) break;  // 3 sec timeout, avoid infinite loop
                }
             }
             ao();
          move_to_goal_flag = 0;
       }
       sleep(buff_tick);                       // for the purpose of getting out of local minima
    }
}

// Avoid rocks using bumpers
#define rock_tick 0.3

void avoid_rock() {
   int i=0;
   int j=0;

   while (1) {
      check_bumper();
      avoid_rock_flag = 1;

      if (left_rock && right_rock) {  // rock in the front
         backward();
         sleep(rock_tick);
         random_turn();
         sleep(rock_tick*(float)(random(3)+1));
      }
      else if (right_rock) {                 // rock in the right
           backward();
           sleep(rock_tick);
           i=i+1;
           if (i>3) {
              random_turn();
              i=0;
           }
           else left_turn();
           sleep(rock_tick*(float)(random(2)+1));
        }
       else if (left_rock) {         //  rock in the left
            backward();
```

```
            sleep(rock_tick);
            j=j+1;
            if (j>3) {
               random_turn();
               j=0;
            }
            else right_turn();
            sleep(rock_tick*(float)(random(2)+1));
         }
         avoid_rock_flag = 0;
   }
}

// avoid haz obj using ranger finders
#define hazobj_tick 0.3
#define hazobj_thresh 5

void avoid_hazobj() {
   while (1) {
      check_hazobj();
      avoid_hazobj_flag = 1;

      if (left_haz && right_haz) {                  //  if the haz obj is head on, then
         if (hazobj_delta > hazobj_thresh) {     //    decide which turn to take
            backward();
            sleep(hazobj_tick);
            left_turn();
            sleep(hazobj_tick*(float)(random(2)+1));
         }
         else if (hazobj_delta < -hazobj_thresh) {
              backward();
              sleep(hazobj_tick);
              right_turn();
              sleep(hazobj_tick*(float)(random(2)+1));
          }
          else {
              backward();
              sleep(hazobj_tick);
              random_turn();
              sleep(hazobj_tick);
          }
      }
      else if (left_haz) {                        //  the haz obj is on the left
           right_turn();
           sleep(hazobj_tick*(float)(random(2)+1));
       }
       else if (right_haz) {                      //  the haz obj is on the right
```

```
            left_turn();
            sleep(hazobj_tick*(float)(random(2)+1));
          }
          avoid_hazobj_flag = 0;
    }
}

// cruise function
void cruise()
{
    while (1) {
       if (!avoid_rock_flag && !avoid_hazobj_flag && !move_to_goal_flag)
         {
          // if the robot is not avoiding rocks/haz obj or adjusting the direction to the target
          //   which means it is taking turns or backward
          forward();                          //   then drive forward
       }
    }
}

// main function
void main()
{
    int pid1,pid2,pid3,pid4;

    while (!start_button()) ;

    pid1 = start_process(move_to_goal());    // start by searching the target
    pid2 = start_process(avoid_rock());
    pid3 = start_process(avoid_hazobj());
    pid4 = start_process(cruise());

    while (!stop_button())  ;

    kill_process(pid1);
    kill_process(pid2);
    kill_process(pid3);
    kill_process(pid4);

    ao();
    beep();
}
```