

Systems and Control Engineering Laboratory (SC 626)
Kilobotics

Abhishek Rajopaadhye (193230004)
Harsha Priyanka Guntaka (193236001)
Neelam Patwardhan (193234001)

March 2, 2020

Contents

1	Introduction	4
1.1	Overview	4
1.2	Features of Kilobot	4
1.3	Requirements	5
1.4	Over-Head Controller(OHC)	6
1.5	Motor Calibration	7
2	Orbiting of Kilobots	8
2.1	Objective	8
2.2	With single stationary Kilobot	9
2.2.1	Results and Demonstration	9
2.3	With multiple stationary Kilobots	10
2.3.1	Results and Demonstration	11
3	Gradient Formation	13
3.1	Objective	13
3.2	Results and Demonstration	14
4	Edge following	15
4.1	Objective	15
4.2	Result and Demonstration	16
5	Gradient formation and edge following integration	17
5.1	Objective	17
5.2	Result and Demonstration	18
6	Future scope	20
7	Acknowledgment	21
A	Code for star robot	22
B	Code for planet robot	24
C	Code for planet robot with multiple stars	27

D	Code for reference robot	31
E	Code for gradient formation	33
F	Code for edge following	38
G	Code for gradient formation and edge following integration	44

List of Figures

1.1	Kilobot	4
1.2	IR sensing	6
1.3	Overhead Controller	6
1.4	Motor calibration using KiloGUI	7
2.1	Flowchart for orbiting a Kilobot(Single star)	8
2.2	Orbiting of Kilobot (Single Star)	9
2.3	Planet colliding with one of the stars	10
2.4	Flowchart for orbiting a Kilobot(Multiple star)	11
2.5	Orbiting of Kilobot (Multiple Star, <i>MOTOR_ON_DURATION = 500, TOTAL_NUM_COMMUNICATION = 4</i>)	12
2.6	Orbiting of Kilobot (Multiple Star, <i>MOTOR_ON_DURATION = 800, TOTAL_NUM_COMMUNICATION = 3</i>)	12
3.1	Flowchart for gradient formation	14
3.2	Display of colors as per different ids	14
4.1	Flowchart for Edge following	15
4.2	Edge Following with <i>TOTAL_NUM_COMMUNICATION=5</i> and <i>TOTAL_NUM_COMMUNICATION_ORBIT=5</i>	15
5.1	Integration of gradient formation and Edge following	18
5.2	Gradient formation and Edge following integration with <i>TOTAL_NUM_COMMUNICATION_GRADIENT=20</i> and <i>TOTAL_NUM_COMMUNICATION = 15</i>	19

Chapter 1

Introduction

1.1 Overview

Kilobots(Figure 1.1) are low cost robots designed at Harvard University's Self-Organizing Systems Research Lab <http://www.eecs.harvard.edu/ssr>. The robots are designed to make testing collective algorithms on hundreds or thousands of robots accessible to robotics researchers.



Figure 1.1: Kilobot

Though the Kilobots are low-cost, they maintain abilities similar to other collective robots. These abilities include differential drive locomotion, on-board computation power, neighbor-to-neighbor communication, neighbor-to neighbor distance sensing, and ambient light sensing. Additionally they are designed to operate such that no robot requires any individual attention by a human operator. This makes controlling a group of Kilobots easy, whether there are 10 or 1000 in the group.

1.2 Features of Kilobot

- Processor : ATmega 328p (8bit @ 8MHz)

- Memory :
 - 32 KB Flash used for both user program and bootloader
 - 1KB EEPROM for storing calibration values and other non-volatile data and 2KB SRAM.
- Battery : Rechargeable Li-Ion 3.7V, for a 3 months autonomy in sleep mode. Each Kilobot has a built-in charger circuit, which charges the onboard battery when +6 volts is applied to any of the legs, and GND is applied to the charging tab.
- Charging : Kilobot charger for 10 robots simultaneously (optional).
- Communication : Kilobots can communicate with neighbors up to 7 cm away by reflecting infrared (IR) light off the ground surface.(Figure 1.2)
- Sensing : 1 IR and 1 light intensity.
 - When receiving a message, distance to the transmitting Kilobot can be determined using received signal strength. The distance depends on the surface used as the light intensity is used to compute the value.
 - The brightness of the ambient light shining on a Kilobot can be detected.
 - A Kilobot can sense its own battery voltage.
- Movement : Each Kilobot has 2 vibration motors, which are independently controllable, allowing for differential drive of the robot. Each motor can be set to 255 different power levels.
- Light : Each Kilobot has a red/green/blue (RGB) LED pointed upward, and each color has 3 levels of brightness control.
- Software : The Kilobot Controller software (kiloGUI) is available for controlling the controller board, sending program files to the robots and controlling them.
- Programming : For programming, the open source development software WinAVR combined with Eclipse gives a C programming environment. An API with basic functions such as motor speed, led control, distance measurement is available and some examples are provided.
- Dimensions : diameter: 33 mm, height 34 mm (including the legs, without recharge antenna).

1.3 Requirements

- Hardware :
 - Computer with Microsoft Windows and an USB port (not included)
 - Kilobot robot
 - Over-head controller (OHC)
 - Kilobot charger

- Software : To start programming the Kilobot with the new version from kilobotics, we have two solutions.
 - Online editor <https://www.kilobotics.com/editor>
 - Install WinAVR and Eclipse to compile the whole library on your computer https://github.com/mgauci/kilobot_notes/blob/master/eclipse_winavr_setup/eclipse_winavr_setup.md

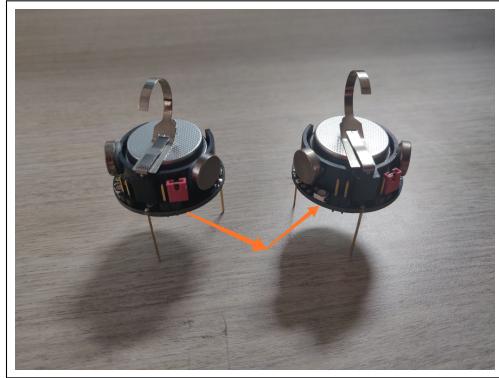


Figure 1.2: IR sensing

1.4 Over-Head Controller(OHC)

In case of Kilobots, instead of plugging in a charging cable for each robot in order to update its program, each can receive a program via an IR communication channel. This allows an over head IR transmitter to program all the robots in the collective in a fixed amount of time, independent of the number of robots.



Figure 1.3: Overhead Controller

1.5 Motor Calibration

Kilobot should be operated on a smooth, flat surface to ensure proper robot mobility. Only one Kilobot can be calibrated at the same time.

1. Place the Kilobot in PAUSE mode.
2. Open KiloGUI interface and then open Calibration mode.(Figure 1.4)
3. The first line (Unique ID) can be used to save an ID in your Kilobot. This can be useful if you want to save all calibration value for each Kilobot.
4. The second line (Turn Left) will configure the kilo_turn_left parameter to set the CCW movement. Set a value for the motor (approximately 70) and press the Test button. Adjust the value to obtain a smooth move.
5. Do the same as explain in point 4, for the right motor (Turn Right line).
6. Next is to set the straight move parameters. Start with the same value for the left and right motor (approximately 60) and press the Test button. Now adjust the two value to move the Kilobot as straight as possible.
7. Finally, when all the parameters are fine, you can press the Save button to write the parameters in the EEPROM of the Robot.

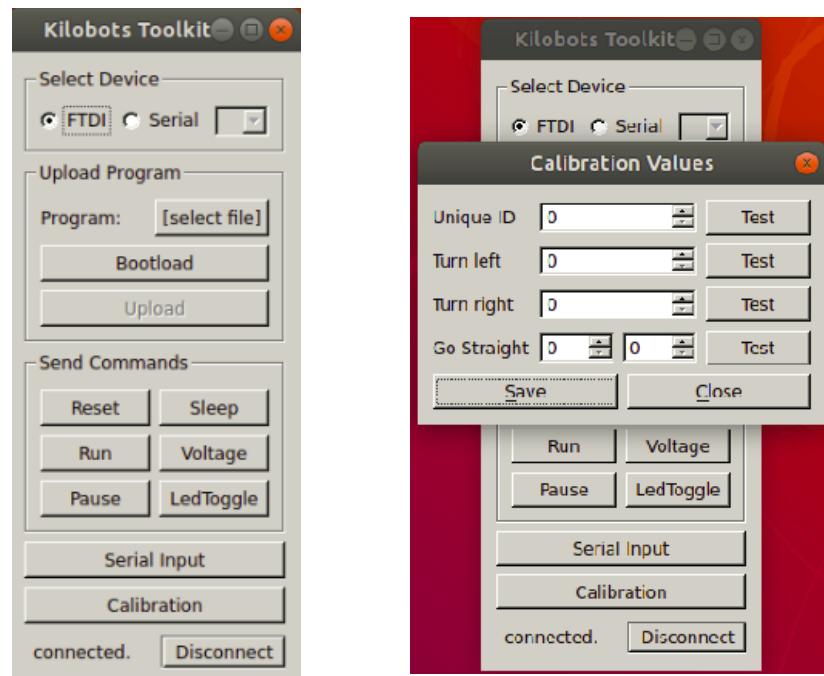


Figure 1.4: Motor calibration using KiloGUI

Chapter 2

Orbiting of Kilobots

2.1 Objective

Our objective is to make a Kilobot(planet) orbit around

- one stationary Kilobot(star)
- multiple stationary Kilobots.

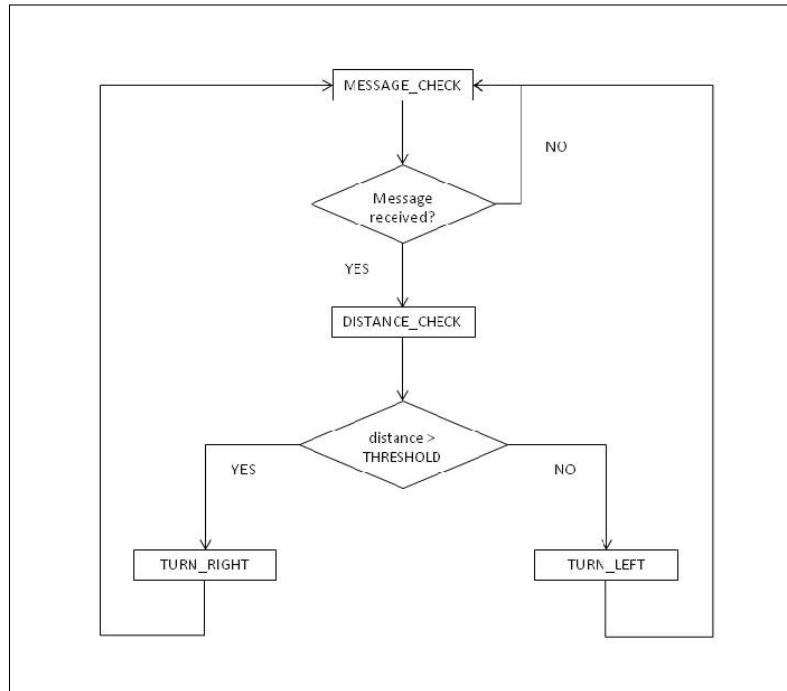


Figure 2.1: Flowchart for orbiting a Kilobot(Single star)

2.2 With single stationary Kilobot

The algorithm for the planet motion with a single star is as follows:

1. Check for the message signal from star Kilobot.
2. If message not received go to step 1.
3. If message is received, calculate the distance.
4. If the distance is greater than Threshold value(fixed radius), move right. Else, move left.
5. Go to step 1.

Flowchart to the corresponding algorithm is illustrated in Figure [2.1](#).

2.2.1 Results and Demonstration

As per the Kilobots manual, the maximum and minimum communication ranges are 110mm and 33mm respectively. So we have taken the orbit radius (*THRESHOLD*) as 50mm, which falls within good communication range. Also, we have given motor on time (*MOTOR_ON_DURATION*) as 500ms.

Video of working demo of problem statement can be accessed from the link in Figure [2.2](#).



Figure 2.2: Orbiting of Kilobot (Single Star)

2.3 With multiple stationary Kilobots

We have placed one more Kilobot within the communication range of the planet and star. It was observed that the planet collides with one of the star due to communication delay between them.

Video of working demo can be accessed from the link in Figure 2.3.

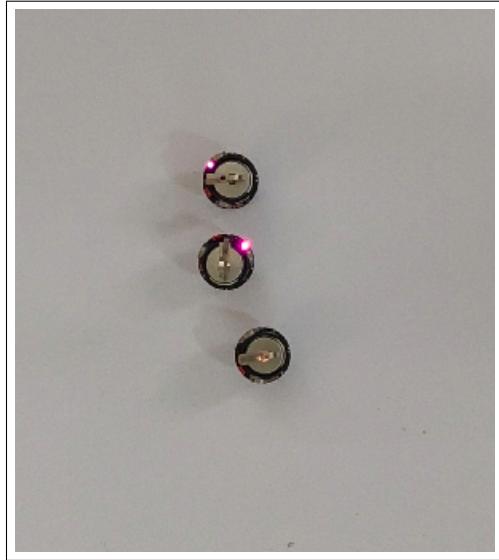


Figure 2.3: Planet colliding with one of the stars

To avoid collision, we have modified the above algorithm by checking the message from the neighbor Kilobots for multiple times. The algorithm is as follows:

1. First initialize a variable (*distance*) corresponding to distance between star and planet to 1000 (or to any other value which is greater than the range of Kilobots).
2. Check for message from neighbor Kilobots. If you receive the message, estimate the distance, then compare it with the value in the *distance* variable and replace the variable with the lowest value.
3. Go to step 2, until the number of messages received are equal to *TOTAL_NUM_COMMUNICATION*.
4. Now, the value of *distance* variable will be our distance between star and planet. If the distance is greater than required orbit radius (*THRESHOLD*) move right. Else, move left.
5. Go to step 1.

The flowchart for the corresponding algorithm is illustrated in Figure 2.4.

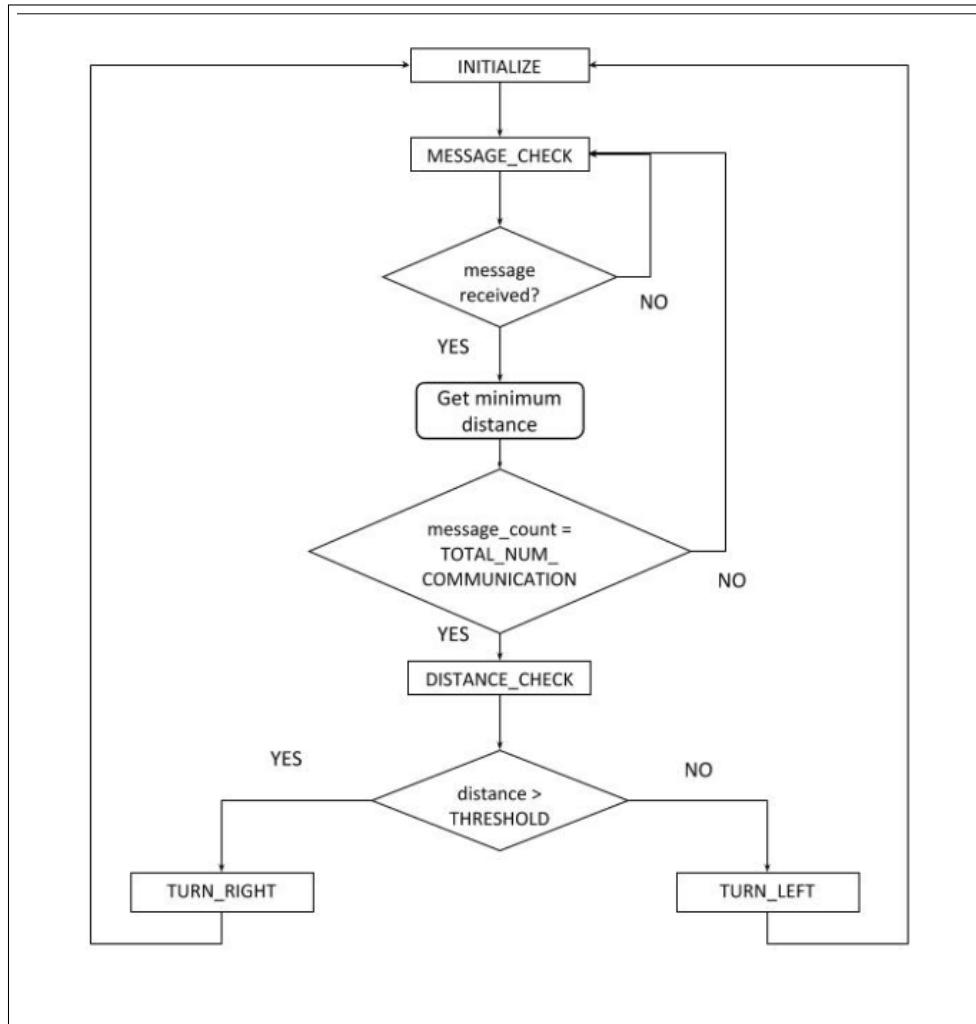


Figure 2.4: Flowchart for orbiting a Kilobot(Multiple star)

2.3.1 Results and Demonstration

We have used the same orbit radius ($THRESHOLD = 50$), and same motor on time ($MOTOR_ON_DURATION = 500$) as in the case of single star. To avoid collision between Kilobots we have checked for the message for multiple times i.e. $TOTAL_NUM_COMMUNICATION = 4$.

Video of working demo for this problem statement can be accessed from the link in Figure 2.5.

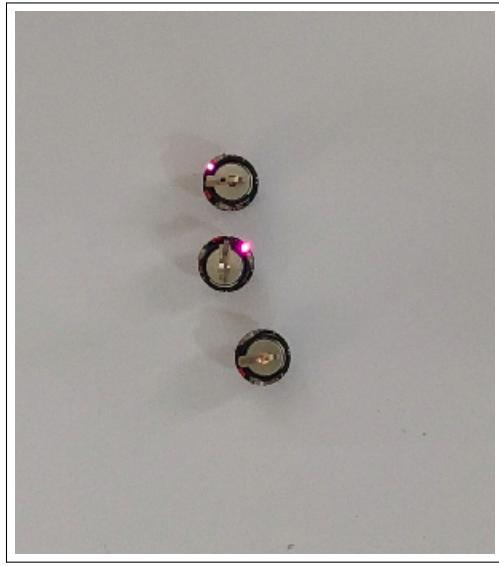


Figure 2.5: Orbiting of Kilobot (Multiple Star, $MOTOR_ON_DURATION = 500$, $TOTAL_NUM_COMMUNICATION = 4$)

It can be observed that in this case the speed of the planet Kilobot is less compared to the case with single stationary Kilobot. It can be increased by increasing the $MOTOR_ON_DURATION$ and decreasing the $TOTAL_NUM_COMMUNICATION$.

Video of working demo for this problem statement can be accessed from the link in Figure 2.6.



Figure 2.6: Orbiting of Kilobot (Multiple Star, $MOTOR_ON_DURATION = 800$, $TOTAL_NUM_COMMUNICATION = 3$)

Chapter 3

Gradient Formation

3.1 Objective

Our objective of this experiment to assign unique ids to kilobots using a reference robot. They will get numbered as per their distance from reference robot. This is how the gradient is formed.

1. First, we initialize the reference kilobot to 0. This kilobot is going to send the message.
2. The id's of the other kilobots are assigned based on the distance threshold.
3. The bot(s) nearest to the reference bot will receive the message and get id 1.
4. The self unique id of all the other kilobots is preset at 251.
5. These bots send data and those within the zone of communication with threshold set as 50mm;their unique ids will be checked and updated to +1.If not, the message is checked again.
6. If there are multiple kilobots within the same zone of communication and falling within the same threshold limit,the one(s) having the least distance are again checked for their id's.
7. The temporary id is compared to the present self unique id and the one with the least id is updated as the new id of the kilobot(s).
8. To distinguish between set of kilobots having same id's,they are represented with colors red,blue,green etc.

Flowchart to the corresponding algorithm is illustrated in Figure [3.1](#).

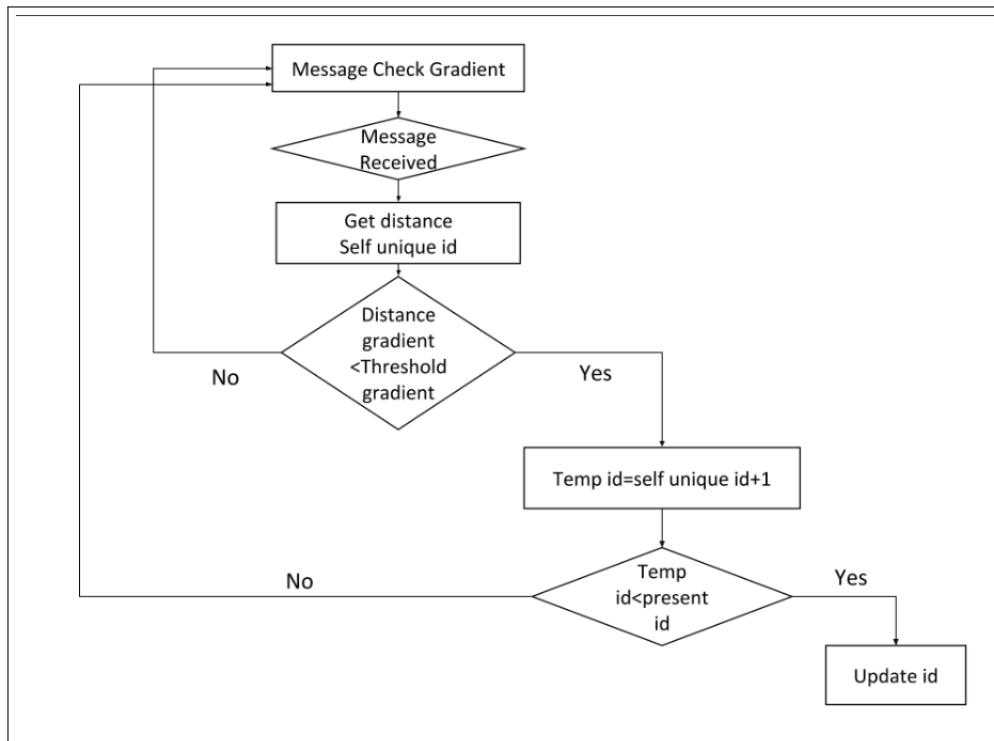


Figure 3.1: Flowchart for gradient formation

3.2 Results and Demonstration

To differentiate between different ids of the kilobots they are converted into binary to represent three colors red, blue and green. Video of working demo of problem statement can be accessed from the link in Figure 3.2.



Figure 3.2: Display of colors as per different ids

Chapter 4

Edge following

4.1 Objective

Our objective is to make the kilobots which are on the outer edge to move along the edge of a group of kilobots by measuring distances without being physically blocked and reach the reference bot.

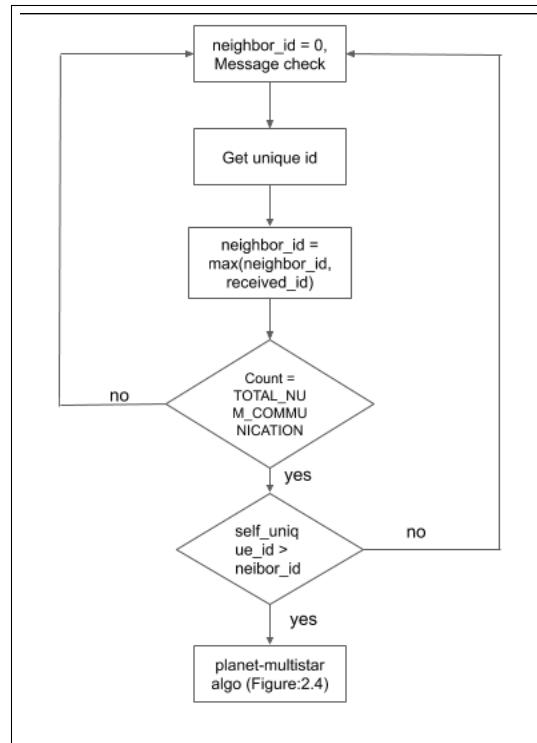


Figure 4.1: Flowchart for Edge following

The algorithm is as follows:

1. Check for the message from neighboring kilobots for TOTAL_NUM_COMMUNICATION times.
2. For each message received get the unique id and store the maximum id.
3. Compare it's own unique id with maximum neighbor id.
4. If self unique id > max. neighbor id, start moving towards the reference bot using *planet multistar* algorithm.
5. If not, go to step 1.

Flowchart to the corresponding algorithm is illustrated in Figure 4.1.

4.2 Result and Demonstration

Kilobots along the outer edge of the initial group are able to move without being physically blocked. They can determine that they are on the outer edge by comparing their gradient values to those of their neighbors. We have used TOTAL_NUM_COMMUNICATION=5, so that robot receives enough communication messages from neighbouring bots to identify the position. These robots can then use edge-following around the stationary initial group to reach the reference kilobot. We have used TOTAL_NUM_COMMUNICATION_ORBIT=3, to make sure bot receives right information about distance while orbiting. Video of working demo of problem statement can be accessed from the link in Figure 4.2.

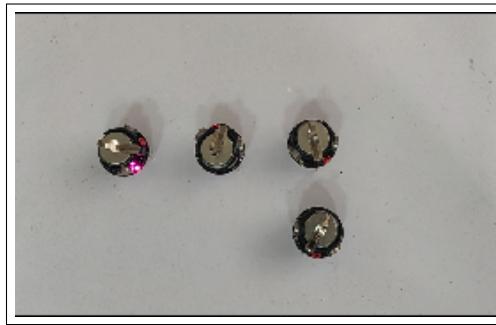


Figure 4.2: Edge Following with TOTAL_NUM_COMMUNICATION=5 and TOTAL_NUM_COMMUNICATION_ORBIT=3

It can be observed that once the outer edge (with max. unique id) kilobot goes out of communication range of the next outer edge kilobot, it will start moving towards the reference bot.

Chapter 5

Gradient formation and edge following integration

5.1 Objective

Our objective of this experiment is to form gradient with respect to reference robot first and then bring the farthest robot near the reference robot by using edge following algorithm. We are integrating the gradient formation and edge following experiments performed earlier.

The algorithm is as follows:

1. We are starting with *gradient formation algorithm* as shown in figure 3.1
2. To make sure that the gradient is stabilized, we have added one more variable called 'count' which will run the *gradient formation algorithm* 20 times.
3. check $\text{count} \leq \text{TOTAL_NUM_COMMUNICATION_GRADIENT}$.
4. If the condition is satisfied then increase the count and run the further loop.
5. When the count is more than $\text{TOTAL_NUM_COMMUNICATION_GRADIENT}$, then start *edge following algorithm* as shown in figure 4.1

Flowchart to the corresponding algorithm is illustrated in Figure [5.1](#).

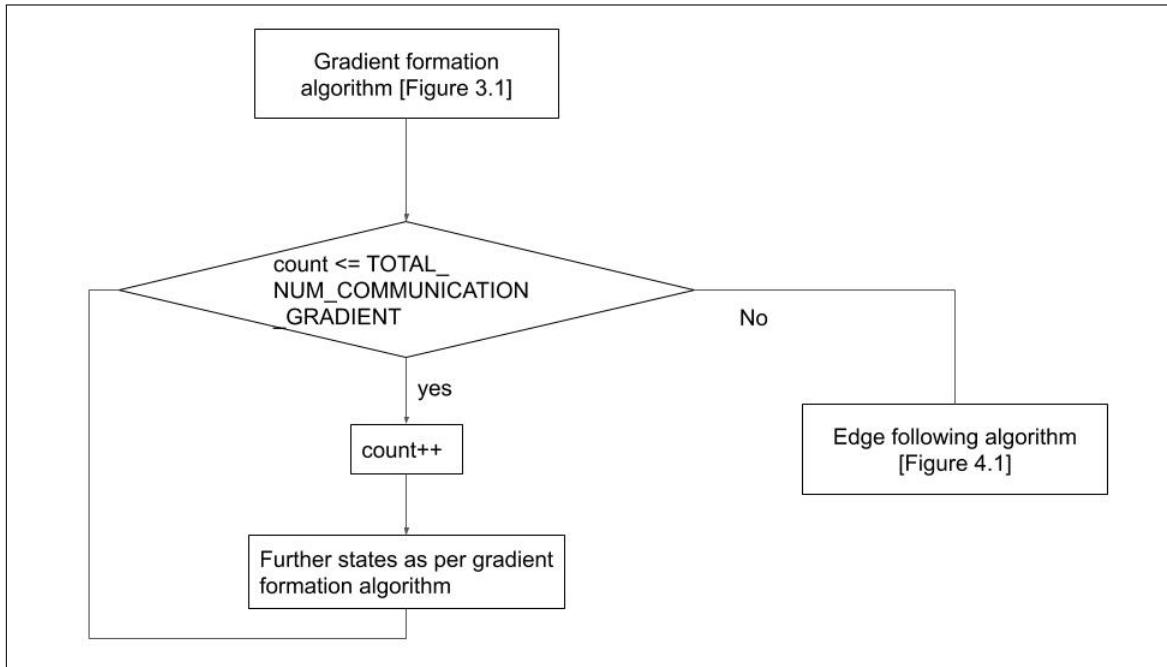


Figure 5.1: Integration of gradient formation and Edge following

5.2 Result and Demonstration

We have arranged the robots in one line. When the code is run, firstly the robots are forming gradient which is visible from the LED color. We have kept `TOTAL_NUM_COMMUNICATION_GRADIENT = 20` to stabilize the gradient. Then the farthest kilobot starts orbiting, and following the edge of the remaining robots, it comes near the reference robot. We kept `TOTAL_NUM_COMMUNICATION = 15`. This has made sure that robots have received signals from maximum robots in the range and avoid simultaneous orbiting. Once we remove this robot from communication range of others, the next farthest robot starts to orbit. This way we are bringing each robot near reference robot. Video of working demo of problem statement can be accessed from the link in Figure 5.2.



Figure 5.2: Gradient formation and Edge following integration with TOTAL_NUM_COMMUNICATION_GRADIENT=20 and TOTAL_NUM_COMMUNICATION = 15

Chapter 6

Future scope

By doing gradient formation and edge following, we are bringing robots in a group near reference robot one by one. So scope for next work is to guide the robot which has come near reference robot to form predefined shape.

Challenges for shape formation extending the same codes are as follows:

1. We are using single reference robot for gradient formation. But when we want to guide a robot to a position in predefined shape, we will need minimum 2-3 reference robots.
2. On laboratory scale we can hardcore the reference robot and extend the same code. But on large scale we will require more complex algorithms.
3. For our experiment, we have kept the robots in straight line initially. So it is easy to identify outermost robot and make it orbit. In case of randomly placed robots, we will require to add few more parameters in the code.
4. Then we can use the same code done by our previous batch and form some shape.

Chapter 7

Acknowledgment

We would like to thank Anurag Gupta for his teaching assistance and clearing our silliest doubts. We would like to thanks lab staff for maintaining a healthy number of working robots. We would also like to thank Prof. Leena Vachhani and Prof. Arpita Sinha for all the support and guidance. We thank Adwiath Vijaykumar for coordination of lab report work and cooperation.

Appendix A

Code for star robot

```
#include <kilolib.h>

message_t message;
// Flag to keep track of message transmission.
int message_sent = 0;

void setup()
{
    // Initialize message:
    // The type is always NORMAL.
    message.type = NORMAL;
    // Some dummy data as an example.
    message.data[0] = 3;
    // It's important that the CRC is computed after the data has been set;
    // otherwise it would be wrong.
    message.crc = message_crc(&message);
}

void loop()
{
    // Blink LED magenta whenever a message is sent.
    if (message_sent == 1)
    {
        // Reset flag so LED is only blinked once per message.
        message_sent = 0;

        set_color(RGB(1, 0, 1));
        delay(100);
        set_color(RGB(0, 0, 0));
    }
}
```

```
message_t *message_tx()
{
    return &message;
}

void message_tx_success()
{
    // Set flag on message transmission.
    message_sent = 1;
}

int main()
{
    kilo_init();
    // Register the message_tx callback function.
    kilo_message_tx = message_tx;
    // Register the message_tx_success callback function.
    kilo_message_tx_success = message_tx_success;
    kilo_start(setup, loop);

    return 0;
}
```

Appendix B

Code for planet robot

```
1 #include <kilolib.h>
2
3 //defining states
4 #define MESSAGE_CHECK 0
5 #define DISTANCE_CHECK 1
6 #define TURN_LEFT 2
7 #define TURN_RIGHT 3
8 #define LEFT 100
9 #define RIGHT 150
10
11 //radius of the orbit
12 #define THRESHOLD 50
13
14 //time delay between message receive and next message check
15 #define MOTOR_ON_DURATION 500
16
17 int state = MESSAGE_CHECK, distance, message_rx_status = 0;
18
19 void message_rx(message_t *m, distance_measurement_t *d)
20 {
21     message_rx_status = 1;                      //initializing the algo
22     once you receive the message
23     distance = estimate_distance(d);           //estimating the distance
24     between planet and star
25 }
26
27 void move(int direction)
28 {
29     switch(direction)                         //defining which motion
30         of motors
```

```

29     {
30     case LEFT:
31         spinup_motors();
32         set_motors(kilo_straight_left, 0);
33         delay(MOTOR_ON_DURATION);
34         set_motors(0, 0);
35     break;
36
37     case RIGHT:
38         spinup_motors();
39         set_motors(0, kilo_straight_right);
40         delay(MOTOR_ON_DURATION);
41         set_motors(0, 0);
42     break;
43
44     default:
45     break;
46 }
47 }
48
49 void setup()
50 {
51
52 }
53
54 void loop()
55 {
56     switch(state)
57     {
58     case MESSAGE_CHECK:
59         if(message_rx_status==1)
60         {
61             message_rx_status = 0;           //to identify the next message
62             from star
63             state = DISTANCE_CHECK;
64         }
65     break;
66
67     case DISTANCE_CHECK:                  //to check which
68         motor should turn ON
69         if(distance>THRESHOLD)
70         {
71             state = TURN_RIGHT;
72         }

```

```

73     else
74     {
75         state = TURN_LEFT;
76     }
77     break;
78
79     case TURN_LEFT:           //right motor turns
80         ON and runs for specified time delay (MOTOR_ON_DURATION)
80     move(LEFT);
81     state = MESSAGE_CHECK;
82     break;
83
84     case TURN_RIGHT:          //left motor turns
85         ON and runs for specified time delay (MOTOR_ON_DURATION)
85     move(RIGHT);
86     state = MESSAGE_CHECK;
87     break;
88
89     default:
90     break;
91 }
92
93 }
94
95
96
97 int main()
98 {
99     kilo_init();           //initializing the Kilobot(
100    planet)
100    kilo_message_rx = message_rx; //register message_rx call back
101    function
101    kilo_start(setup,loop);
102
103    return 0;
104 }
```

Appendix C

Code for planet robot with multiple stars

```
1 #include <kilolib.h>
2
3 //defining states
4 #define MESSAGE_CHECK 0
5 #define DISTANCE_CHECK 1
6 #define TURN_LEFT 2
7 #define TURN_RIGHT 3
8 #define INITIALIZE 4
9 #define LEFT 100
10 #define RIGHT 150
11
12 #define THRESHOLD 50           //orbit radius
13 #define MOTOR_ON_DURATION 500  //time delay between message
                               receive and next message check
14 #define TOTAL_NUM_COMMUNICATION 4 //number of times we have to
                               check for message from star
15
16 int state = INITIALIZE, distance, message_rx_status = 0, count = 0,
     temp_distance;
17
18 void message_rx(message_t *m, distance_measurement_t *d)
19 {
20     message_rx_status = 1;           //initializing the algo
                               once you receive the message
21     temp_distance = estimate_distance(d); //estimating the
                               distance between planet and star
22 }
23
```

```

24
25 void move(int direction)
26 {
27     switch(direction)                                //defining which
28     {
29         case LEFT:
30             spinup_motors();
31             set_motors(kilo_straight_left, 0);
32             delay(MOTOR_ON_DURATION);
33             set_motors(0, 0);
34             break;
35
36         case RIGHT:
37             spinup_motors();
38             set_motors(0, kilo_straight_right);
39             delay(MOTOR_ON_DURATION);
40             set_motors(0, 0);
41             break;
42
43
44     default:
45         break;
46     }
47 }
48
49 void setup()
50 {
51
52 }
53
54 void loop()
55 {
56     switch(state)
57     {
58         case MESSAGE_CHECK:
59             if(message_rx_status == 1)
60             {
61                 message_rx_status = 0;           //to identify the next
62                 message from any of the star
63                 count++;
64                 if(temp_distance < distance)    //to find which star
65                     is nearer to the planet
66                 {
67                     distance = temp_distance;
68                 }

```

```

67         if(count == TOTAL_NUM_COMMUNICATION)      //count for no. of
68             messages received
69         {
70             count = 0;
71             state = DISTANCE_CHECK;
72         }
73     }
74     break;
75
76 case DISTANCE_CHECK:
77     if(distance>THRESHOLD)                      //to check
78         which motor should turn ON
79     {
80         state = TURN_RIGHT;
81     }
82     else
83     {
84         state = TURN_LEFT;
85     }
86
87     break;
88
89 case TURN_LEFT:                                //right motor turns
90     ON and runs for specified time delay (MOTOR_ON_DURATION)
91     move(LEFT);
92     state = INITIALIZE;
93     break;
94
95 case TURN_RIGHT:                               //left motor turns
96     ON and runs for specified time delay (MOTOR_ON_DURATION)
97     move(RIGHT);
98     state = INITIALIZE;
99     break;
100
101 case INITIALIZE:
102     distance = 1000;
103     state = MESSAGE_CHECK;
104     break;
105
106 default:
107     break;
108 }
```

```
109
110
111
112 int main()
113 {
114     kilo_init();                                //initializing the
115     Kilobot(planet)                           //register message_rx
116     kilo_message_rx = message_rx;
117     call back function
118     kilo_start(setup,loop);
119 }
```

Appendix D

Code for reference robot

```
1 #include <kilolib.h>
2
3 message_t message;
4 // Flag to keep track of message transmission.
5 int message_sent = 0;
6
7 void setup()
8 {
9     // Initialize message:
10    // The type is always NORMAL.
11    message.type = NORMAL;
12    // Some dummy data as an example.
13    message.data[0] = 0;
14    // It's important that the CRC is computed after the data has
15    // been set;
16    // otherwise it would be wrong.
17    message.crc = message_crc(&message);
18 }
19 void loop()
20 {
21     // Blink LED magenta whenever a message is sent.
22     if (message_sent == 1)
23     {
24         // Reset flag so LED is only blinked once per message.
25         message_sent = 0;
26
27         set_color(RGB(1, 0, 1));
28         delay(100);
29         set_color(RGB(0, 0, 0));
30     }
}
```

```
31 }
32
33 message_t *message_tx()
34 {
35     return &message;
36 }
37
38 void message_tx_success()
39 {
40     // Set flag on message transmission.
41     message_sent = 1;
42 }
43
44 int main()
45 {
46     kilo_init();
47     // Register the message_tx callback function.
48     kilo_message_tx = message_tx;
49     // Register the message_tx_success callback function.
50     kilo_message_tx_success = message_tx_success;
51     kilo_start(setup, loop);
52
53     return 0;
54 }
```

Appendix E

Code for gradient formation

```
1 #include <kilolib.h>
2
3 #define MESSAGE_CHECK_GRADIENT 0
4 #define DISTANCE_THRESHOLD_GRADIENT 1
5 #define TURN_LEFT 2
6 #define TURN_RIGHT 3
7 #define INITIALIZE 4
8 #define THRESHOLD_GRADIENT 100
9 #define LEFT 100
10 #define RIGHT 150
11 #define MOTOR_ON_DURATION 800
12 #define TOTAL_NUM_COMMUNICATION 3
13
14 int received_msg, message_sent = 0, state = MESSAGE_CHECK_GRADIENT,
     distance, distance_gradient, message_rx_status = 0, count = 0,
     temp_unique_id, self_unique_id = 251, ref_unique_id;
15 message_t message;
16
17 void message_rx(message_t *m, distance_measurement_t *d)
18 {
19     received_msg = (*m).data[0];
20     message_rx_status = 1;
21     distance_gradient = estimate_distance(d);
22     /*set_color(RGB(1,0,1));
23     delay(500);*/
24 }
25
26 void LED_color()
27 {
28     set_color(RGB((self_unique_id % 2), ((self_unique_id / 2) % 2),
                  ((self_unique_id / 4) % 2)));

```

```

29     delay(1000);
30 }
31
32 message_t *message_tx()
33 {
34     return &message;
35 }
36
37 void message_tx_success()
38 {
39     message_sent = 1;
40     set_color(RGB(1, 0, 1));
41     delay(100);
42     set_color(RGB(0, 0, 0));
43     LED_color();
44 }
45
46
47
48 /*void move(int direction)
49 {
50     switch(direction)
51     {
52         case LEFT:
53             spinup_motors();
54             set_motors(kilo_straight_left, 0);
55             delay(MOTOR_ON_DURATION);
56             set_motors(0, 0);
57             break;
58
59         case RIGHT:
60             spinup_motors();
61             set_motors(0, kilo_straight_right);
62             delay(MOTOR_ON_DURATION);
63             set_motors(0, 0);
64             break;
65
66
67         default:
68             break;
69     }
70 }*/
71
72 void setup()
73 {
74     message.type = NORMAL;

```

```

75         message.data[0] = self_unique_id;
76         //message.data[0] = 1;
77         message.crc = message_crc(&message);
78         kilo_message_tx = message_tx;
79         kilo_message_tx_success = message_tx_success;
80
81 }
82
83 void loop()
84 {
85     switch(state)
86     {
87         /*case MESSAGE_CHECK:
88             if(message_rx_status == 1)
89             {
90                 message_rx_status = 0;
91                 count++;
92                 if(temp_distance < distance)
93                 {
94                     distance = temp_distance;
95                 }
96                 if(count == TOTAL_NUM_COMMUNICATION)
97                 {
98                     count = 0;
99                     state = DISTANCE_CHECK;
100                }
101            }
102
103        break;
104
105    case DISTANCE_CHECK:
106        if(distance>THRESHOLD)
107        {
108            state = TURN_RIGHT;
109        }
110        else
111        {
112            state = TURN_LEFT;
113        }
114    }
115
116    break;
117
118    case TURN_LEFT:
119        move(LEFT);
120        state = INITIALIZE;

```

```

121     break;
122
123     case TURN_RIGHT:
124         move(RIGHT);
125         state = INITIALIZE;
126     break;
127
128     case INITIALIZE:
129         distance = 1000;
130         state = MESSAGE_CHECK;
131     break; */
132
133     case MESSAGE_CHECK_GRADIENT:
134 /*set_color(RGB(0,0,1));
135 delay(500);*/
136     if(message_rx_status == 1)
137     {
138         /*set_color(RGB(1,0,0));
139         delay(500);*/
140         message_rx_status = 0;
141         ref_unique_id = received_msg;
142         state = DISTANCE_THRESHOLD_GRADIENT;
143     }
144     else
145     {
146         /*set_color(RGB(0,0,1));
147         delay(500);*/
148         state = MESSAGE_CHECK_GRADIENT;
149     }
150     break;
151
152     case DISTANCE_THRESHOLD_GRADIENT:
153 //set_color(RGB(1,0,0));
154 //delay(500);
155     if(distance_gradient < THRESHOLD_GRADIENT)
156     {
157         temp_unique_id = ref_unique_id + 1;
158         if(temp_unique_id < self_unique_id)
159         {
160             set_color(RGB(0,0,1));
161             self_unique_id = temp_unique_id;
162             message.data[0] = self_unique_id;
163             LED_color();
164         }
165     }
166 /*else

```

```
167     {
168         set_color(RGB(1,0,1));
169         delay(500);
170     }/*
171     state = MESSAGE_CHECK_GRADIENT;
172     break;
173
174     default:
175     break;
176 }
177
178 }
179
180
181
182 int main()
183 {
184     kilo_init();
185     kilo_message_rx = message_rx;
186     kilo_start(setup,loop);
187
188     return 0;
189 }
```

Appendix F

Code for edge following

```
1 #include <kilolib.h>
2
3 //defining states
4 #define MESSAGE_CHECK_EF 0
5 #define ID_CHECK 1
6 #define ORBIT 2
7 #define TURN_LEFT 3
8 #define TURN_RIGHT 4
9 #define DISTANCE_CHECK 5
10 #define INITIALIZE 6
11 #define INFINITE 7
12
13 #define THRESHOLD 50           // distance between the moving
     outer edge bot and the remaining bots
14
15 //Kilobot motion parameters
16 #define LEFT 100
17 #define RIGHT 150
18 #define MOTOR_ON_DURATION 500
19
20 #define TOTAL_NUM_COMMUNICATION 5           //total
     number of communication to identify its position
21 #define TOTAL_NUM_COMMUNICATION_ORBIT 3      //total
     number of communication while moving towards the reference bot
22
23 int received_msg, message_sent = 0, neighbor_id = 0, state =
     MESSAGE_CHECK_EF, temp_distance, distance, message_rx_status =
     0, count = 0, count_orbit = 0, self_unique_id = 251;
24 message_t message;
25
26 void message_rx(message_t *m, distance_measurement_t *d)
```

```

27 {
28     received_msg = (*m).data[0];
        //receives unique_id of communicating bot
29     message_rx_status = 1;
        //indicates message decoded successfully
30     temp_distance = estimate_distance(d);
        //distance estimation of communicating bot
31     /*set_color(RGB(1,0,1));
32     delay(500);*/
33 }
34
35 /*void LED_color()
36 {
37     set_color(RGB((self_unique_id % 2), ((self_unique_id / 2) % 2),
            ((self_unique_id / 4) % 2)));
38     delay(1000);
39 }*/
40
41 message_t *message_tx()
42 {
43     return &message;
44 }
45
46
47 void message_tx_success()
48 {
49     message_sent = 1;
        //message(self unique_id) sent successfully
50     /*set_color(RGB(1, 0, 1));
51     delay(100);
52     set_color(RGB(0, 0, 0));*/
53     // LED_color();
54 }
55
56
57 void move(int direction)
58 {
59     switch(direction)
60     {
61         case LEFT:
62             spinup_motors();
63             set_motors(kilo_straight_left, 0);
64             delay(MOTOR_ON_DURATION);
65             set_motors(0, 0);
66         break;
67

```

```

68     case RIGHT:
69         spinup_motors();
70         set_motors(0, kilo_straight_right);
71         delay(MOTOR_ON_DURATION);
72         set_motors(0, 0);
73     break;
74
75
76     default:
77     break;
78 }
79 }
80
81 void setup()
82 {
83     message.type = NORMAL;
84     message.data[0] = kilo_uid;                                //transmitting
85     unique_id
86     //message.data[0] = 1;
87     message.crc = message_crc(&message);                    //parity check
88     kilo_message_tx = message_tx;
89     kilo_message_tx_success = message_tx_success;
90 }
91
92 void loop()
93 {
94     switch(state)
95     {
96     case MESSAGE_CHECK_EF:
97         //message check for identifying the outer edge bot
98         if(message_rx_status == 1)
99         {
100             message_rx_status = 0;
101             count++;
102             if(received_msg > neighbor_id)
103             {
104                 neighbor_id = received_msg;           //for each message
105                 received_neighbor_id stores max(unique_id)
106             }
107             if(count == TOTAL_NUM_COMMUNICATION)
108             {
109                 count = 0;
110                 state = ID_CHECK;

```

```

109         }
110     }
111
112     break;
113
114     case ID_CHECK:
115         if(kilo_uid > neighbor_id)
116         {
117             //set_color(RGB(1,0,0));
118             state = ORBIT;                                //if self id is
119                                         //more than neighbor id, starts moving towards ref. bot
120             //state = INFINITE;
121         }
122         else
123         {
124             neighbor_id = 0;
125             state = MESSAGE_CHECK_EF;
126         }
127
128     break;
129
130     case ORBIT:                                     //we've
131         used the planet-multistar algo
132         if(message_rx_status == 1)
133         {
134             message_rx_status = 0;                      //to identify the next
135                                         //message from any of the bot
136             count++;
137             if(temp_distance < distance)              //to find which bot
138                 is nearer to it
139             {
140                 distance = temp_distance;
141             }
142             if(count == TOTAL_NUM_COMMUNICATION_ORBIT) //count for
143                 no. of messages received
144             {
145                 count = 0;
146                 state = DISTANCE_CHECK;
147             }
148         }
149
150     break;
151
152     case DISTANCE_CHECK:                           //to check
153         if(distance>THRESHOLD)
154             which motor should turn ON

```

```

149     {
150         state = TURN_RIGHT;
151     }
152     else
153     {
154         state = TURN_LEFT;
155     }
156 }
157
158 break;
159
160 case TURN_LEFT: //right motor turns
161     ON and runs for specified time delay (MOTOR_ON_DURATION)
162     move(LEFT);
163     state = INITIALIZE;
164     break;
165
166 case TURN_RIGHT: //left motor turns
167     ON and runs for specified time delay (MOTOR_ON_DURATION)
168     move(RIGHT);
169     state = INITIALIZE;
170     break;
171
172 case INITIALIZE:
173     distance = 1000;
174     state = ORBIT;
175     break;
176
177 case INFINITE:
178     break;
179
180 default:
181     break;
182 }
183 }
184
185
186
187 int main()
188 {
189     kilo_init();
190     kilo_message_rx = message_rx;
191     kilo_start(setup,loop);
192

```

```
193     return 0;  
194 }
```

Appendix G

Code for gradient formation and edge following integration

```
1 #include <kilolib.h>
2
3 #define MESSAGE_CHECK_EF 0
4 #define ID_CHECK 1
5 #define ORBIT 2
6 #define TURN_LEFT 3
7 #define TURN_RIGHT 4
8 #define DISTANCE_CHECK 5
9 #define INITIALIZE 6
10 #define INFINITE 7
11 #define INITIALIZE_GRADIENT 8
12 #define MESSAGE_CHECK_GRADIENT 9
13 #define DISTANCE_THRESHOLD_GRADIENT 10
14
15 #define THRESHOLD_VALUE_GRADIENT 70
16 #define THRESHOLD 50
17 #define LEFT 100
18 #define RIGHT 150
19 #define MOTOR_ON_DURATION 500
20 #define TOTAL_NUM_COMMUNICATION 15
21 #define TOTAL_NUM_COMMUNICATION_ORBIT 3
22 #define TOTAL_NUM_COMMUNICATION_GRADIENT 20
23
24 int received_msg, distance_gradient, message_sent = 0,
     temp_unique_id, neighbor_id, state = INITIALIZE_GRADIENT,
     temp_distance, distance, message_rx_status = 0, count = 0,
     self_unique_id = 251;
25 message_t message;
```

```

26
27 void message_rx(message_t *m, distance_measurement_t *d)
28 {
29     received_msg = (*m).data[0];
30     message_rx_status = 1;
31     temp_distance = estimate_distance(d);
32     distance_gradient = estimate_distance(d);
33     /*set_color(RGB(1,0,1));
34     delay(500);*/
35 }
36
37 void LED_color()
38 {
39     switch (self_unique_id)
40 {
41     case 1: set_color(RGB(1,0,0));
42     break;
43     case 2: set_color(RGB(0,1,0));
44     break;
45     case 3: set_color(RGB(0,0,1));
46     break;
47     case 4: set_color(RGB(1,1,0));
48     break;
49     case 5: set_color(RGB(1,0,1));
50     break;
51     case 6: set_color(RGB(0,1,1));
52     break;
53     case 7: set_color(RGB(1,1,1));
54     break;
55     default: set_color(RGB(0,0,0));
56     break;
57 }
58     delay(1000);
59 }
60
61 message_t *message_tx()
62 {
63     return &message;
64 }
65
66
67 void message_tx_success()
68 {
69     message_sent = 1;
70     /*set_color(RGB(1, 0, 1));
71     delay(100);
```

```

72     set_color(RGB(0, 0, 0));*/
73 // LED_color();
74 }
75
76
77 void move(int direction)
78 {
79     switch(direction)
80     {
81         case LEFT:
82             spinup_motors();
83             set_motors(kilo_straight_left, 0);
84             delay(MOTOR_ON_DURATION);
85             set_motors(0, 0);
86             break;
87
88         case RIGHT:
89             spinup_motors();
90             set_motors(0, kilo_straight_right);
91             delay(MOTOR_ON_DURATION);
92             set_motors(0, 0);
93             break;
94
95
96         default:
97             break;
98     }
99 }
100
101 void setup()
102 {
103     message.type = NORMAL;
104     message.data[0] = self_unique_id;
105 //message.data[0] = 1;
106     message.crc = message_crc(&message);
107     kilo_message_tx = message_tx;
108     kilo_message_tx_success = message_tx_success;
109 }
110
111
112 void loop()
113 {
114     switch(state)
115     {
116         case INITIALIZE_GRADIENT:
117             message_rx_status = 0;

```

```

118     //state = MESSAGE_CHECK_GRADIENT;
119     if(count <= TOTAL_NUM_COMMUNICATION_GRADIENT)
120     {
121         state = MESSAGE_CHECK_GRADIENT;
122     }
123     else
124     {
125         count = 0;
126         state = MESSAGE_CHECK_EF;
127         //state = INFINITE;
128     }
129     break;
130
131     case MESSAGE_CHECK_GRADIENT:
132     if(message_rx_status == 1)
133     {
134         state = DISTANCE_THRESHOLD_GRADIENT;
135         //state = INFINITE;
136         count++;
137     }
138     else
139     {
140         state = INITIALIZE_GRADIENT;
141     }
142     break;
143
144     case DISTANCE_THRESHOLD_GRADIENT:
145     //state = INFINITE;
146     if(distance_gradient < THRESHOLD_VALUE_GRADIENT)
147     {
148         temp_unique_id = received_msg + 1;
149         if(temp_unique_id < self_unique_id)
150         {
151             //set_color(RGB(0,0,1));
152             self_unique_id = temp_unique_id;
153             message.data[0] = self_unique_id;
154             message.crc = message_crc(&message);
155             LED_color();
156         }
157     }
158     state = INITIALIZE_GRADIENT;
159     break;
160
161     case INFINITE:
162     set_color(RGB(1,1,1));
163     break;

```

```

164
165     case MESSAGE_CHECK_EF:
166         if(message_rx_status == 1)
167     {
168             message_rx_status = 0;
169             count++;
170             if(received_msg > neighbor_id)
171             {
172                 neighbor_id = received_msg;
173             }
174             if(count == TOTAL_NUM_COMMUNICATION)
175             {
176                 count = 0;
177                 state = ID_CHECK;
178             }
179         }
180
181     break;
182
183     case ID_CHECK:
184         if(self_unique_id > neighbor_id)
185     {
186             //set_color(RGB(1,0,0));
187             state = ORBIT;
188
189             //state = INFINITE;
190         }
191         else
192     {
193             neighbor_id = 0;
194             state = MESSAGE_CHECK_EF;
195         }
196
197     break;
198
199     case ORBIT:
200         if(message_rx_status == 1)
201     {
202             message_rx_status = 0;           //to identify the next
203             message from any of the star
204             count++;                     //to find which star
205             if(temp_distance < distance)
206                 is nearer to the planet
207             {
208                 distance = temp_distance;
209             }

```

```

208     if(count == TOTAL_NUM_COMMUNICATION_ORBIT)      //count for
209         no. of messages received
210     {
211         count = 0;
212         state = DISTANCE_CHECK;
213     }
214 }
215 break;
216
217 case DISTANCE_CHECK:
218     if(distance>THRESHOLD)                         //to check
219         which motor should turn ON
220     {
221         state = TURN_RIGHT;
222     }
223 else
224 {
225     state = TURN_LEFT;
226 }
227
228 break;
229
230 case TURN_LEFT:                                //right motor turns
231     ON and runs for specified time delay (MOTOR_ON_DURATION)
232     move(LEFT);
233     state = INITIALIZE;
234 break;
235
236 case TURN_RIGHT:                               //left motor turns
237     ON and runs for specified time delay (MOTOR_ON_DURATION)
238     move(RIGHT);
239     state = INITIALIZE;
240 break;
241
242 case INITIALIZE:
243     distance = 1000;
244     state = ORBIT;
245 break;
246
247 default:
248     break;
249 }
```

```
250 }
251
252
253
254 int main()
255 {
256     kilo_init();
257     kilo_message_rx = message_rx;
258     kilo_start(setup, loop);
259
260     return 0;
261 }
```