Systems and Control Engineering Laboratory (SC 626)
Kilobotics

Anurag (183230006)
Ponala Venkata Eswara Srisai (183230008)
Sudhakar Kumar (183236001)
Kishan Chouhan (183230015)

April 24, 2019

Contents

1 Overview of Kilobots **4**
1.1 Specifications of Kilobot 4
1.2 Hardware/ Software Requirement 5
1.3 Over-head Controller 6

2 Establishing communication between two Kilobots **7**
2.1 Speaker 7
2.2 Listener 9
2.3 Results and Demonstration 10

3 Orbiting of Kilobots **12**
3.1 Introduction 12
3.2 Results and Demonstration 13

4 Moving towards the direction of light source **14**
4.1 Introduction 14
4.2 Results and Demonstration 15

5 Synchronizing phase of blinking LEDs **16**
5.1 Introduction 16
5.2 Results and Demonstration 19

6 Efficient star-planet orbiting using a finite state machine **20**
6.1 Finite state machine (FSM) 20
6.2 Flowchart 21
6.3 Results and Demonstration 23
6.4 Conclusion 26

7 Shape formation using Kilobots **27**
7.1 Framework 27
7.2 Flowchart 28
7.3 Discussion 28
7.4 Demonstration 29

8 Conclusion **31**
8.1 Acknowledgement 31

A Code for orbiting after escape algorithm **32**

B Code for shape formation algorithm **38**

List of Figures

1.2 Overhead Controller (Source: [?]) 6

2.1 Broadcasting of a message 9
2.2 Communication between two Kilobots 11

3.1 Flowchart for orbiting of kilobot 12
3.2 Orbiting of Kilobot 13

4.1 Flowchart for move towards light algorithm 14
4.2 Move towards the light source 15

5.1 Algorithm for synchronizing phase of blinking LEDs 17
5.2 Synchronizing phase of blinking LEDs 19

6.1 Flowchart for orbiting after escape algorithm (Part I) 21
6.2 Flowchart for orbiting after escape algorithm (Part II) 22
6.3 Efficient star-planet orbiting 23
6.4 Escaping too close region of star by planet followed by orbiting 24
6.5 Orbiting of planet around two stars using single communication to estimate minimum distance leads to instability 25
6.6 Orbiting of planet around two stars using two communications to estimate minimum distance 25
6.7 Orbiting of planet around three stars using four communications to estimate minimum distance 26

7.1 Flowchart for shape formation algorithm 28
7.2 First builder taking its position 29
7.3 Second builder taking its position 29
7.4 Shape formation by kilobots (Shape: Rectangle of breadth=2 and length=3) 30

Chapter 1

Overview of Kilobots

Kilobots (Figure 1.1a) 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.

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.

1.1 Specifications of Kilobot

The functional specifications [**?**] of the Kilobot robot are as follow:

- **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
- **Communication**: Kilobots can communicate with neighbors up to 7 cm away by reflecting infrared (IR) light off the ground surface (Figure 1.1b).
- **Sensing**: one IR sensor and one light sensor

  - 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. The movement happens via *stick and slip* mechanism

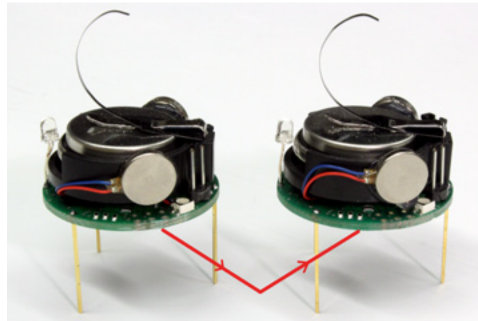
- **Light**: Each Kilobot has a red/green/blue (RGB) LED pointed upward, and each color has 3 levels of brightness control.

- **Software**: The Kilobot Controller software (kiloGUI) is available for controlling the controller board, sending program files to the robots and controlling them.

- **Programming**: C language with WinAVR compiler combined with Eclipse or the online Kilobotics editor (https://www.kilobotics.com/editor)

- **Dimensions**: diameter: 33 mm, height 34 mm (including the legs, without recharge antenna).

(a) Kilobot



(b) IR sensing (Source [?])

1.2 Hardware/ Software Requirement

The required hardware and software to use the board and develop programs are described below.

Required hardware:

- Computer with an USB port
- Kilobot robot
- Over-head controller (OHC)
- Kilobot charger

Required 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.

We have used online editor for our labs.

1.3 Over-head Controller

To make a robot scalable to large collective sizes, all the operations of the robot must work on the collective as a whole [?], and not require any individual attention to the robot, such as pushing a switch or plugging in a charging cable for each robot. In other words, all collective operations must be scalable. In the case of Kilobots, instead of plugging in a programming cable to each robot in order to update its program, each can receive a program via an IR communication channel. This allows an overhead IR transmitter (Figure 1.2) to program all the robots in the collective in a fixed amount of time, independent of the number of robots.

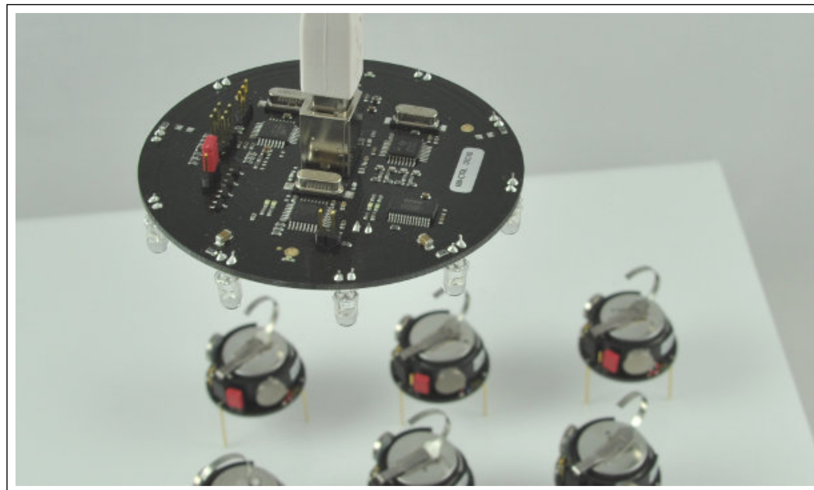


Figure 1.2: Overhead Controller (Source: [?])

Kilobots are unique in that they stay in "sleep mode" until summoned by the overhead controller. A person can turn an entire swarm of Kilobots "ON" by sending out one signal – as opposed to manually switching "ON" every robot.

Chapter 2

Establishing communication between two Kilobots

Here, we will use the ability of two Kilobots to communicate with each other. We will allocate one Kilobot to be the speaker and the other Kilobot to be the listener.

2.1 Speaker

The objective of this part is to broadcast a fixed message, and blink magenta when we transmit. Here we introduce `message_t` data structure, and the kilo_message_tx lback.

- A Kilobot uses IR to broadcast a message within an approximately circular radius of three body lengths.
- Multiple robots packed closely together will interfere the IR signals. So the Kilobots use a variant of the CSMA/CD media access control method to resolve the problems with interference.

  Carrier-sense multiple access with collision detection (CSMA/CD) is a media access control method which uses carrier-sensing to delay transmissions until no other stations are transmitting.

Step 1: Declare a variable called transmit_msg of the structure type message_t and add the function message_tx() that returns the address of the message we declared (return & transmit_msg).

Step 2: We will register this "callback" function in the Kilobot main loop, and every time the Kilobot is ready to send a message it will "interrupt" the main code and call the message_tx() to get the message that needs to be sent.

A callback is any executable code that is passed as an argument to other code, which is expected to call back the argument at a given time.

```
message_t transmit_msg;

message_t *message_tx() {
```

```
    return &transmit_msg;
}
```

Step 3: Use the setup() function to set the initial contents of the message and compute the message CRC value (used for error detection) through the function message_crc(). The code below shows how to initialize a simple message.

A cyclic redundancy check (CRC) is an error-detecting code which is used to detect accidental changes to raw data.

```
void setup() {
    transmit_msg.type = NORMAL;
    transmit_msg.data[0]=0;
    transmit_msg.crc = message_crc(&transmit_msg);
}
```

Step 4: Now we will add some more code so that we can have the LED blink whenever the robot sends a message out. To do this we will declare a variable (often called a "boolean flag") called message_sent. We will set this flag inside a function called message_tx_success() which is another callback function that only gets called when a message is finally successfully sent on a channel. Then we will clear this flag in the main loop where we will blink an LED to indicate that a message was sent.

```
//At the top of the file, declare a "flag" for when a message is sent
int message_sent = 0;

// Add another function definition after your message_tx function
// (note that "void" means the function doesn't return any value)

void message_tx_success() {
   message_sent = 1;
}
```

Step 5: Then, in our program loop, write code to do this

```
// Blink led magenta when you get a message
if (message_sent == 1) {
    message_sent = 0;
    set_color(RGB(1,0,1));
    delay(100);
    set_color(RGB(0,0,0));
}
```

Step 6: Finally, we must register our message transmission functions with the Kilobot library as follows:

```
int main() {
    kilo_init();
    kilo_message_tx = message_tx;
```

```
    kilo_message_tx_success = message_tx_success;
    kilo_start(setup, loop);

    return 0;
}
```

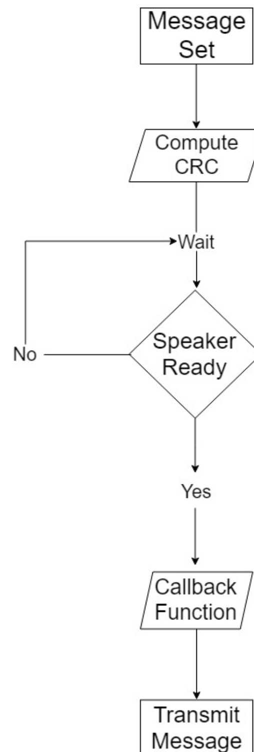


Figure 2.1: Broadcasting of a message

2.2 Listener

The objective of this part is to blink yellow when a new message is received and introduce the kilo_message_rx callback and store incoming messages.

Step 1: First declare a variable rcvd_message of type message_t to store any new incoming messages and a boolean variable called new_message to indicate that a new message has been received.

```
int new_message = 0;
message_t rcvd_message;
```

```
void message_rx(message_t *msg, distance_measurement_t *dist) {
    rcvd_message = *msg;  //store the incoming message
    new_message = 1;      // set the flag to 1 to indicate that a new message arrived
}
```

Step 2: Check the flag in the program loop to see if a new message has arrived. If a new message has arrived, we will blink the LED yellow, and clear the flag.

```
void loop() {
    // Blink led yellow when you get a message
    if (new_message == 1) {
        new_message = 0;
        set_color(RGB(1,1,0));
        delay(100);
        set_color(RGB(0,0,0));
    }
}
```

Step 3: Modify our main section to register the message reception function with the Kilobot library as follows:

```
int main() {
    kilo_init();
    kilo_message_rx = message_rx;
    kilo_start(setup, loop);

    return 0;
}
```

2.3 Results and Demonstration

For this task of establishing communication between two Kilobots, the communication protocol CSMA/CD has been followed. Thus, the speaker Kilobot will first sense and then transmit. Video of working demonstration of problem statement can be accessed using link in the caption of figure 2.2.

Figure 2.2: Communication between two Kilobots

According to the Kilobotics website, by default, every kilobot attempts to send message twice per second. It can also be verified by looking at the frequency of LED blinking. The right Kilobot acts as speaker and the left Kilobot acts as a receiver.

Chapter 3

Orbiting of Kilobots

3.1 Introduction

Our objective is to make one Kilobot orbit around another stationary Kilobot using distance estimation. We will refer to orbiting Kilobot as planet and stationary Kilobot as star. The alogrithm goes as follows:

1. Place star and planet under communication link distance.
2. If planet and star distance $< TOO_LOW$ go to step 6, else go to step 3.
3. If planet and star distance $< DESIRED_DISTANCE$ go to step 4, else go to step 5.
4. Move left. Go to step 2.
5. Move right. Go to step 2.
6. Move straight. Go to step 2.

Flowchart corresponding to orbiting is illustrated in Figure 3.1

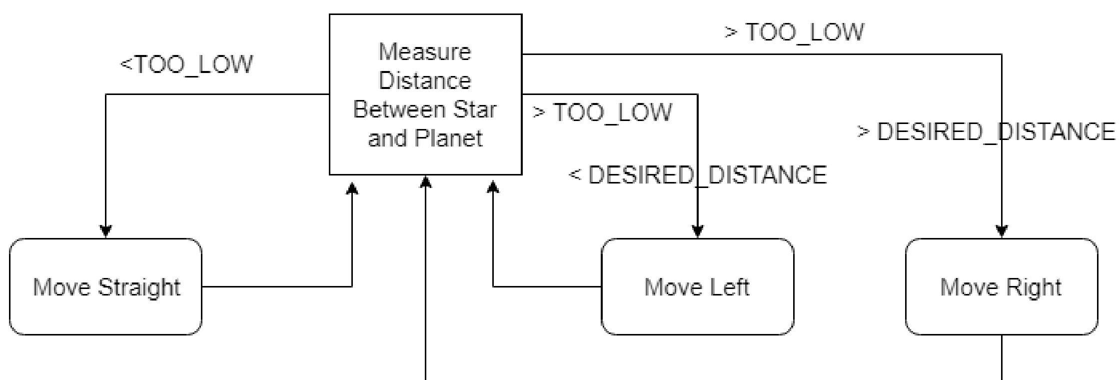


Figure 3.1: Flowchart for orbiting of kilobot

3.2 Results and Demonstration

Star transmits the message to planet continuously while planet receives message from star and it estimates the distance between them using the strength of received signal and then accordingly it follows the algorithm. Following parameters have been used in this algorithm:

- `DESIRED_DISTANCE` = 60. According to the Kilobotics website, this distance of 60mm is a good compromise between the maximum communication range (110mm) and the minimum distance (33mm) when two robots are touching.
- `TOO_CLOSE` = 40. If the current distance estimate to the robot is 4cm or smaller, then the robot is really "too close" to the star. In that case, the robot should just move forward until the distance is no longer too close. This allows the planet robot to get to a reasonable distance away from the star quickly and then start the orbiting. However, if the orientation is not proper, there are chances that the planet might hit the star.

Video of working demo of problem statement can be accessed using link in figure 3.2.

Figure 3.2: Orbiting of Kilobot

We utilize the estimated distances of robot from the strength of received message to determine which direction to move. Furthermore, this algorithm for orbiting of Kilobots can be extended to edge-following if there is a group of stars, then the planets can orbit that group of stars. One such scenario has been presented in the upcoming chapters, which also prevents the planet from hitting the star.

Chapter 4

Moving towards the direction of light source

4.1 Introduction

In this part, our objective is to design an algorithm so that Kilobot approaches a source of light (generated by torch light of smartphone) which may be dynamically moving with very slow speed. The problem statement is similar to that of a line follower with just one onboard sensor [?]. The algorithm for single sensor line follower goes as follows:

1. Check sensor position.
2. If sensor is on line, go to step 3 else step 4.
3. Move right. Go to step 5.
4. Move left.
5. Go to step 1

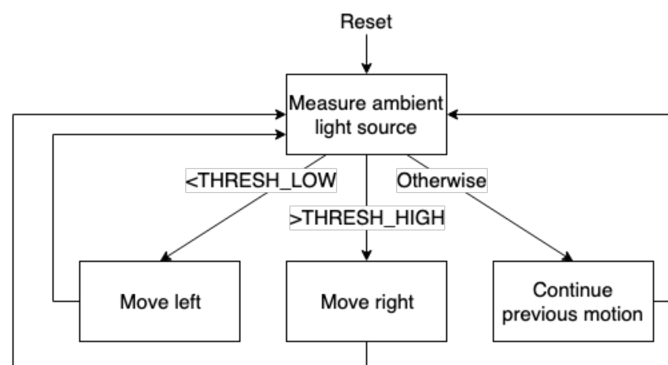


Figure 4.1: Flowchart for move towards light algorithm

On similar lines, following a light source algorithm is implemented using flowcart illustrated in Figure 4.1. Above mentioned algorithm will help us understand why the robot approaches the source of light in a zig zag fashion. We cannot do significant improvement in algorithm given the limitation of onboard ambient light sensor to one.

4.2 Results and Demonstration

Following parameters were used for this algorithm for moving towards the direction of light source:

- Number of samples = 300. In order to measure ambient light, we use the function `get_ambientlight()` to read the current value for the sensor. When a sensor cannot return a good reading, it returns -1. In As such sensors are inherently noisy, we will write a function that "samples" the current light level by averaging 300 sensor readings, while discarding any bad sensor readings.
- `THRESH_LOW` = 300
- `THRESH_HIGH` = 600

Video of working demo of problem statement can be accessed using link in Figure 4.2.

Figure 4.2: Move towards the light source

Different `THRESH_LOW` and `THRESH_HIGH` parameters were chosen to implement hysteresis behaviour, thereby, precluding jittery motion. A high value of `THRESH_HIGH` might ensure that the Kilobot converges closer and closer to the light source. The function `get_ambientlight()` returns a 10-bit measurement (0 to 1023). Accordingly, the value of `THRESH_HIGH` might be set for better convergence towards light source.

Chapter 5

Synchronizing phase of blinking LEDs

5.1 Introduction

Objective: Create a logical synchronous clock between different robots to allow two or more Kilobots to blink an LED in unison roughly every 4 seconds

A large group of decentralized closely cooperating entities, commonly called a collective or **swarm**, can work together to complete a task that is beyond the capabilities of any of its individuals [**?**]. Following are the three basic swarm behaviors [**?**] that Kilobots have mastered:

1. **Foraging** – It involves commanding several robots to disperse and explore the area around them. With Kilobots, the idea is to chip away the time it takes to forage in a particular location.

2. **Formation control** – It means the ability of Kilobots to behave in unison or in a specific part of the swarm. By maintaining communication with each other, Kilobots possess a virtual bearing sensor that gives each one a realistic sense of its position in the group.

3. **Synchronization** – It is often used when coordinating simultaneous actions between many entities, such as robots or sensor networks. We can visualize this by imagining a swarm of 1,000 Kilobots, with each using its LED light to represent a pixel in a larger video that can be viewed from above. To know which color to signal at any given time, every Kilobot must be using the same clock.

Hence, synchronization is one of the swarm behaviours which can be performed by Kilobots. It is often used when coordinating simultaneous actions between many entities, such as robots or sensor networks.

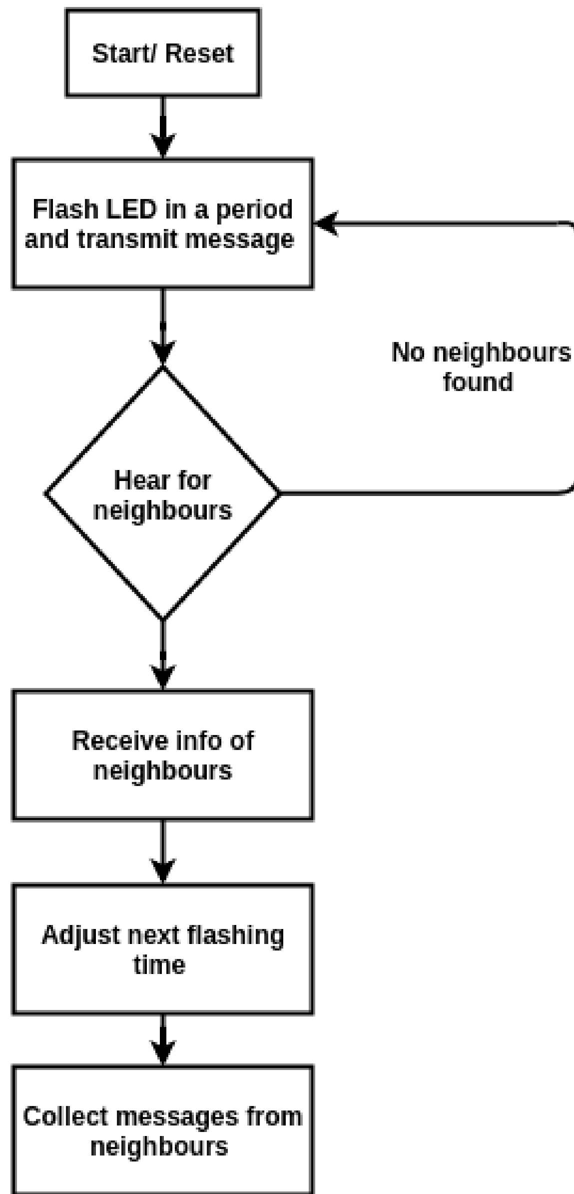


Figure 5.1: Algorithm for synchronizing phase of blinking LEDs

For our objective, we will use a method that relies on averaging. The algorithm for Synchronizing robots is given in the flowchart (Figure 5.1). Each Kilobot acts as an oscillator, flashing its LED in a fixed period `P`. At the same time, it continually transmits a message with its current position in the clock period (i.e. a value between `0` and `P`). In the absence of any neighbors, the Kilobot will simply blink in a fixed period, like a firefly. If the Kilobot hears neighboring Kilobots, then it receives information about their current positions in their own periods. In order to synchronize,

it collects that information and uses the average to make an adjustment to its own next flashing time. The steps can be summarized as given below:

Step 1: Create a robot oscillator that flashes every 2 seconds.

```
#define PERIOD 64
uint32_t reset_time = 0;

// In Program Loop

if (kilo_ticks >= (last_reset + PERIOD)) {
   set LED to red
   last_reset = kilo_ticks
} else {
   turn LED off
}
```

Step 2: Let the Kilobot continually transmit the current position of its clock within its ticking period (i.e. `kilo_ticks - last_reset`). Since we reset our clock every 64 ticks, this value will be less than 64. We want this value to be as accurate as possible, so we can read the clock in the `message_tx` function.

```
message_t message;

message_t *message_tx() {
    message.data[0] = kilo_ticks - last_reset; // current position in PERIOD
    message.crc = message_crc(&message);
    return &message;
}
```

Step 3: Let the Kilobot collect the messages it hears from other neighbors. By comparing the its own current clock position to that of its neighbors (i.e. the first byte of the received message), a Kilobot can tell how much it is out of sync with its neighbors. Each time a new message arrives, the Kilobot will store the value of the adjustment to be made. Then, when the Kilobot completes its own time period and flashes, it will also make one big adjustment for next time's flash.

```
void message_rx(message_t *m, distance_measurement_t *d)
{
    int my_timer = kilo_ticks - last_reset;
    int rx_timer = m->data[0];
    int timer_discrepancy = my_timer - rx_timer;

    // reset time adjustment.
    reset_time_adjustment = reset_time_adjustment +  rx_reset_time_adjustment;
}
```

5.2 Results and Demonstration

Following parameters were used for this algorithm for synchronizing phase:

- `Period` = 50. We use the robot's own clock to check time, by reading the variable `kilo_ticks`. One tick is equivalent to roughly 30 ms, or equivalently there are approximatly 32 clock ticks every second.
- `RESET_TIME_ADJUSTMENT_DIVIDER` = 120. It affects the size of the reset time adjustment for every discrepancy with a neighbor. A larger value means smaller adjustments. Hence, this value should increase with the average number of neighbors each robot has.

Video of working demo of problem statement can be accessed using link in Figure 5.2.

Figure 5.2: Synchronizing phase of blinking LEDs

Chapter 6

Efficient star-planet orbiting using a finite state machine

6.1 Finite state machine (FSM)

Before discussing FSM, let us discuss the three individual terms comprising this concept viz. finite, state and machine. Finite refers to a fixed number, whereas state means the mode or position. For example, in case of a traffic light signal, we have three different states: red, yellow and green. Thus, FSM denotes a machine with fixed number of states. [**?**] views a finite state machine as:

- A set of input events
- A set of output events
- A set of states
- A function that maps states and input to output
- A function that maps states and inputs to states
- A description of the initial state

In our case of star-planet orbiting, we model the problem as a FSM, given its suitability to modularizing different components of algorithm. Our mechanism for orbiting is very similar to those provided at Kilobotics website, and elaborately covered in details in previous chapters, although implementation using a FSM dramatically simplifies integrating macros of large size. Our main contribution in this chapter comprises of a novel and robust obstacle avoidance algorithm to prevent planet from hitting the star if it starts too close to it and also, a simple extension of existing orbiting algorithm for multiple stars setting. Both of these contributions will be dealt with extensively in coming sections.

6.2 Flowchart

Flowchart for orbit after escape algorithm has been divided into two parts comprising of Figure 6.1 and 6.2. First part discusses extension of orbit algorithm for multiple star setting whereas the latter illustrates the algorithm for escaping obstacle.

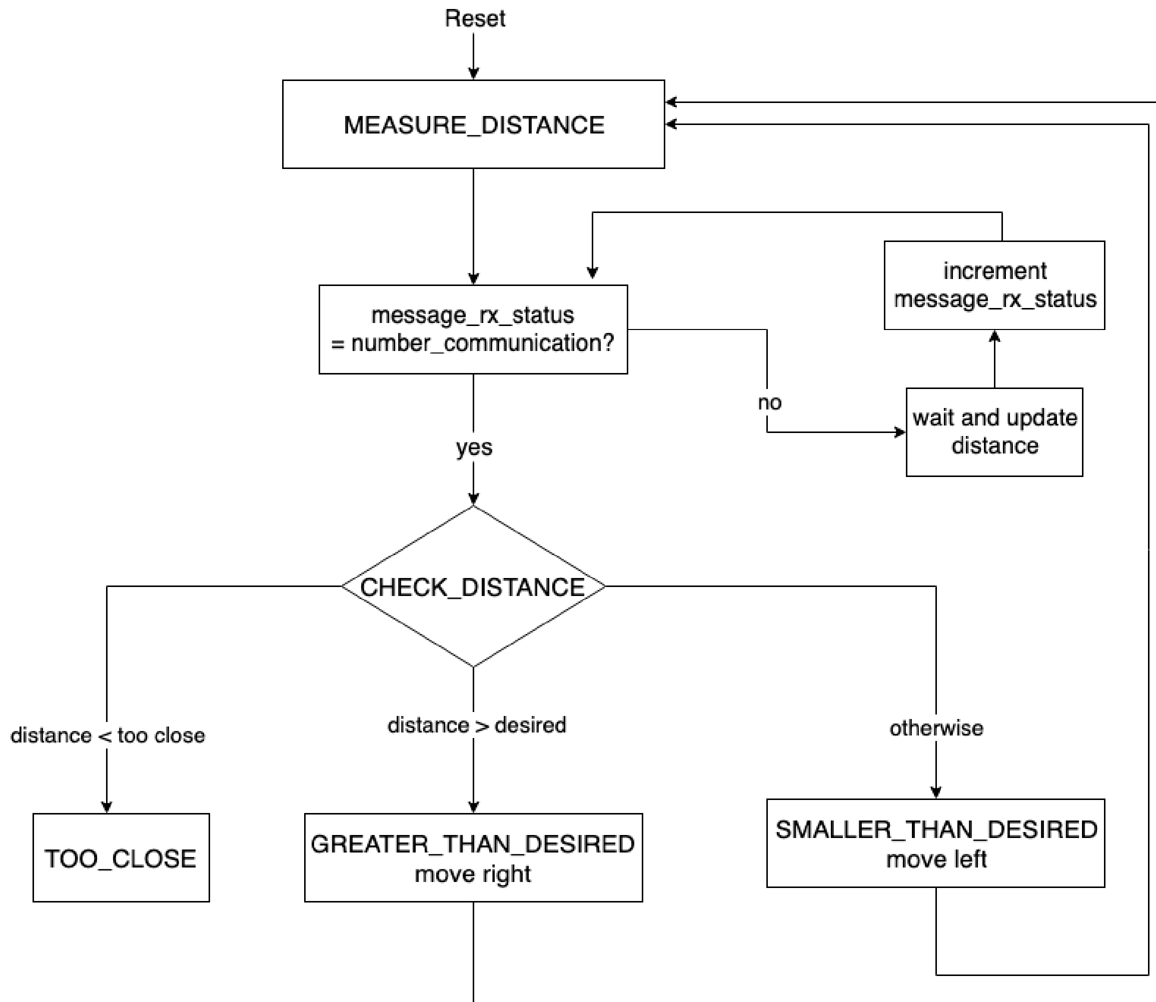


Figure 6.1: Flowchart for orbiting after escape algorithm (Part I)

In star planet orbiting algorithm, while measuring distance between stars and planet, the parameter `NUMBER_COMMUNICATION` defines how many messages should a planet aggregate before making a decision. Accordingly, planet identifies the nearest star based on the strength of signal received in previous step. Considering the nearest star, the planet either moves left or right to maintain `DESIRED_DISTANCE`. The parameter `message_rx_status` is used to keep count of successful communications.

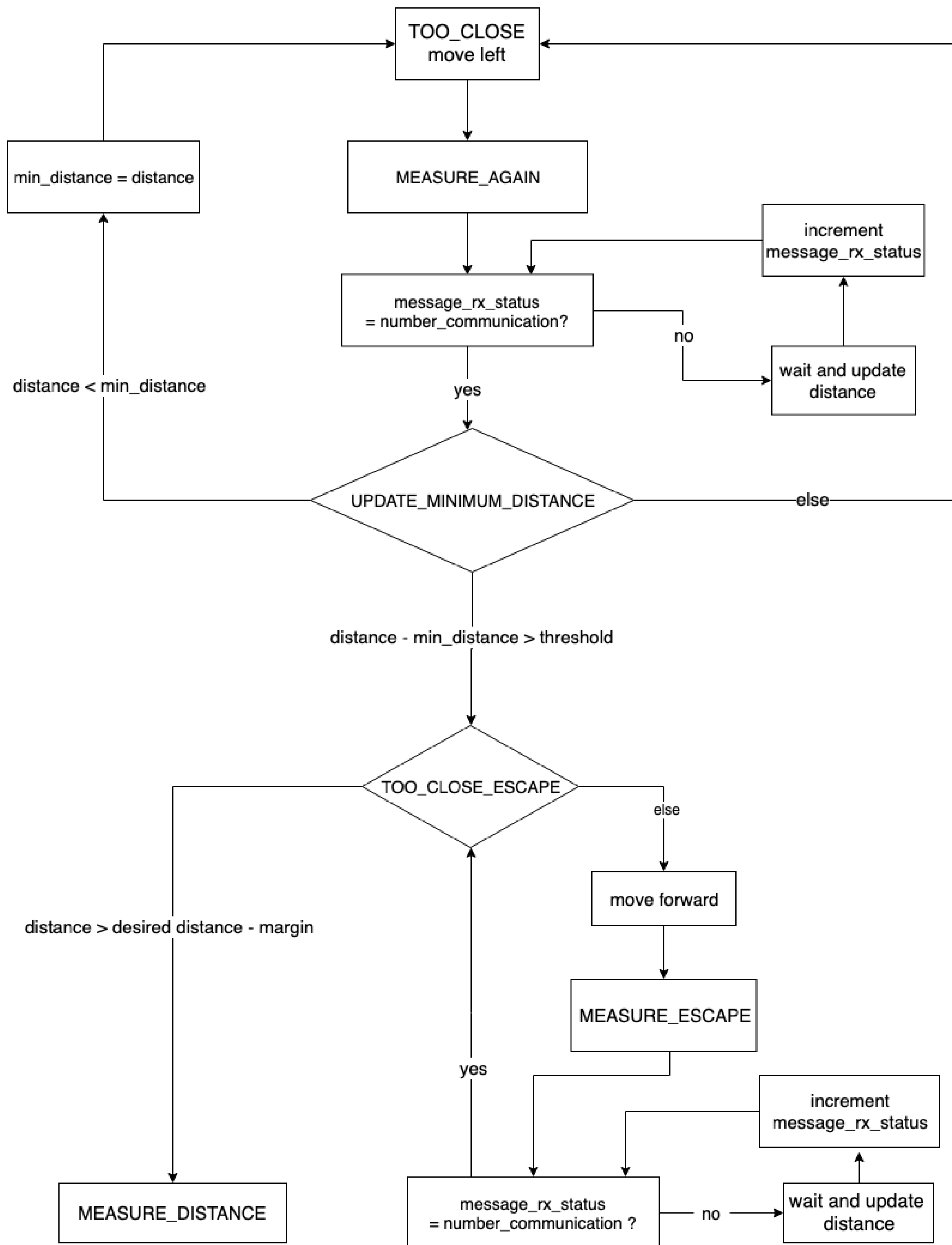


Figure 6.2: Flowchart for orbiting after escape algorithm (Part II)

The parameter `NUMBER_COMMUNICATION` affects speed of the planet. For instance, if the number of stars in communication range is more, the planet moves faster than having only one star in communication range. This is due to the fact that it needs to receive `NUMBER_COMMUNICATION` times of measurements from only one star and then, decides the motion. Here are two observations, which show how the revolution time is getting varied with the number of communications:

- Motor delay = 200, Number of communications = 4, Revolution time = 23 minutes
- Motor delay = 500, Number of communications = 2, Revolution time = 5 minutes

While escaping from `TOO_CLOSE` region, we compare the successive distance measurements to check for the orientation of planet from star. In our early version of algorithm, we tried to detect transition of `distance` from increasing to decreasing state as the instant when planet is oriented away from star but becase of uncertainity in measurements, this led to unwanted behaviour. Hence, we decided to keep track of `minimum_distance` from star and check for the condition `distance - min_distance > THRESHOLD_ROTATE` to hold. By appropriately choosing the parameter `THRESHOLD_ROTATE`, we managed to get the final orientation of planet away from direction of star. Moreover, the modified algorithm sans over-dependence on accuracy of a single measurement, leading to filter like behaviour and an eventual robustness of solution.
It is worth mentioning that the robustness comes at an additional cost, which is, in worst case scenario, planet would require one complete revolution before being able to escape from `TOO_CLOSE` region of star. For more detailed understanding of algorithm, readers are referred to code in Appendix A.

6.3 Results and Demonstration

The caption of figure 6.3 provides link to the demonstration for orbiting of planet around the star, given star starts near the desired distance.

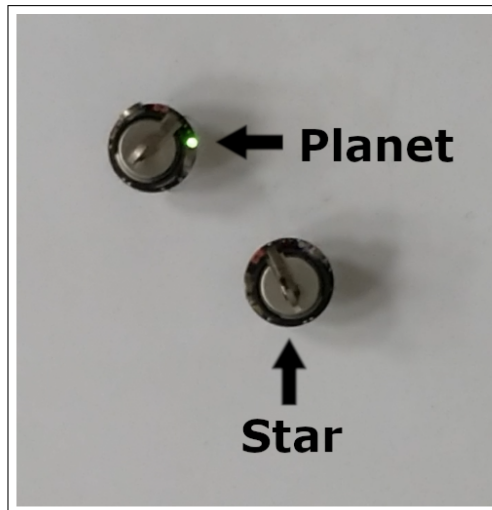


Figure 6.3: Efficient star-planet orbiting

We have hard delays in our current implementation which increases the revolution time. In future, one can replace them with soft delays to alleviate the time of revolution.

The caption of figure 6.4 provides link to the demonstration of algorithm for planet to go away from too close region, which involves using a robust algorithm followed by orbiting.

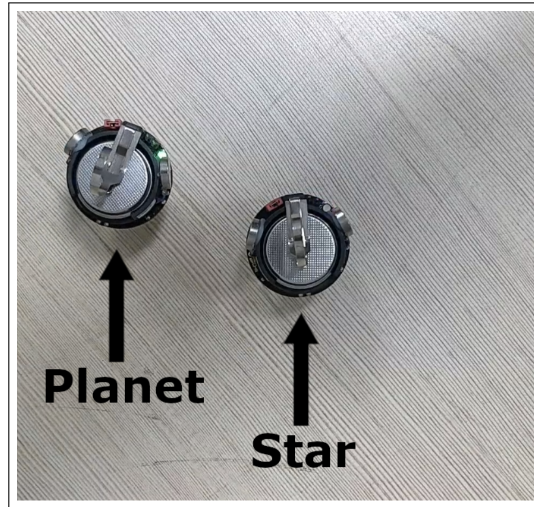


Figure 6.4: Escaping too close region of star by planet followed by orbiting

It is evident from the demonstration that there is a damped oscillation before planet starts orbiting. This can be attributed to difference in centre of rotation of kilobots for clockwise and anti-clockwise turn. If it were to rotate about its centre, the oscillations would not have been as heightened as in current scenario.

The caption of figure 6.5 provides link to the demonstration of algorithm for planet to orbit around two stars. In this case, only one communication was used to identify minimum distance, leading to instability which can be attributed to CSMA/CD [?] communication protocol followed by Kilobots.

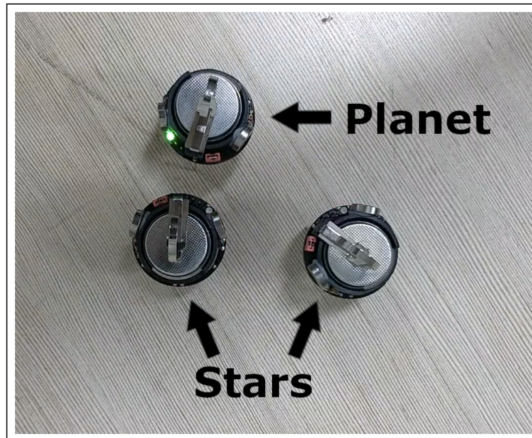


Figure 6.5: Orbiting of planet around two stars using single communication to estimate minimum distance leads to instability

The caption of figure 6.6 provides link to the demonstration of algorithm for planet to orbit around two stars where planet uses two communications to make a descision (estimating the minimum distance).

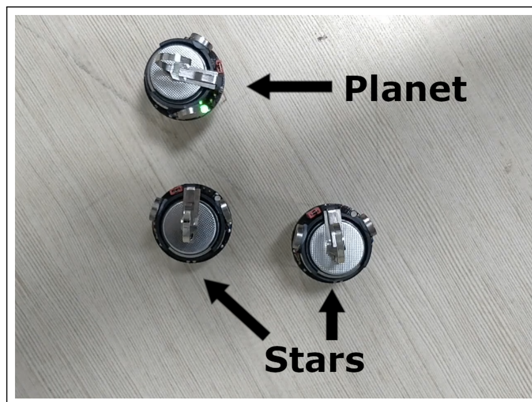


Figure 6.6: Orbiting of planet around two stars using two communications to estimate minimum distance

The caption of figure 6.7 provides link to the demonstration of algorithm for planet to orbit around three stars forming a triangle. In this case, planet uses four communications to make a decision. Though these four communications stabilize the movement of planet, it also increases the total revolution time.

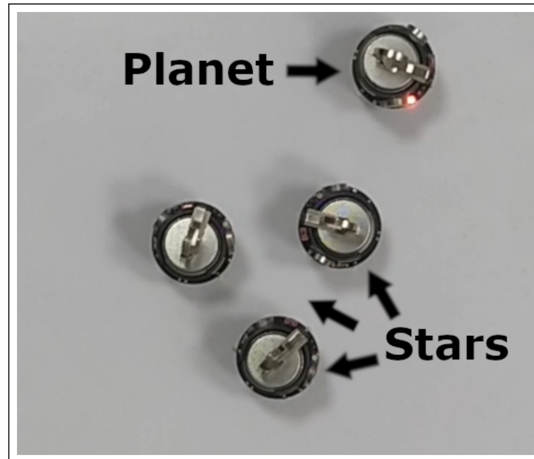


Figure 6.7: Orbiting of planet around three stars using four communications to estimate minimum distance

6.4 Conclusion

We implemented a robust algorithm for star-planet orbiting by using the concepts of FSM. This algorithm was able to prevent planet from hitting the star, even if the planet starts in the very close region of the star. Also, we observed that the movement of a planet can be stabilized by increasing the number of communications from the star. However, it will also increase the total revolution time. Also, the orientation of planets and star played a significant role. If the orientation of the planet is away from that of a star, it might take an additional turn before it begins orbiting.

Chapter 7

Shape formation using Kilobots

In this chapter, we will discuss an algorithm motivated by the work [?] of SSL lab, Harvard University, for shape formation. Rather than implementing the entire algorithm, we will only consider a portion of it, assuming that the next robot to be placed is available near origin.

7.1 Framework

- Three robots (guides) are used as reference for axis orientation.
- A robot (builder) participating in shape formation starts near the left of origin.
- In its effort to reach the desired location, builder orbits around the partial shape using the algorithm presented in last chapter.
- The builder stops orbiting when it reaches the desired location.
- Builder becomes a guide, thereby, helping the next builder.

To participate in shape formation, builders need to decide upon what global shape to form. This is achieved by sharing a shape matrix encapsulating the desired shape as an array of 5-tuple. The 5-tuple contains the following information necessary for establishing builders at desired location:

$$\begin{pmatrix} Index & Neighbour_1 & Desired\ distance_1 & Neighbour_2 & Desired\ distance_2 \end{pmatrix} \tag{7.1}$$

For forming a linear shape of width 2, the shape matrix would look like

$$\begin{bmatrix} 3 & 1 & 1 & 2 & 1 \\ 4 & 2 & 1 & 3 & \sqrt{2} \\ 5 & 3 & 1 & 4 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix} \tag{7.2}$$

It is important to note that we require the shape to ensure two guides for each builder node or else builder would fail to localize correctly.

7.2 Flowchart

Figure 7.1 explains the algorithm for shape formation. Our algorithm differs from [?] in its implementation of shape matrix, and localization to be discussed later in next section.

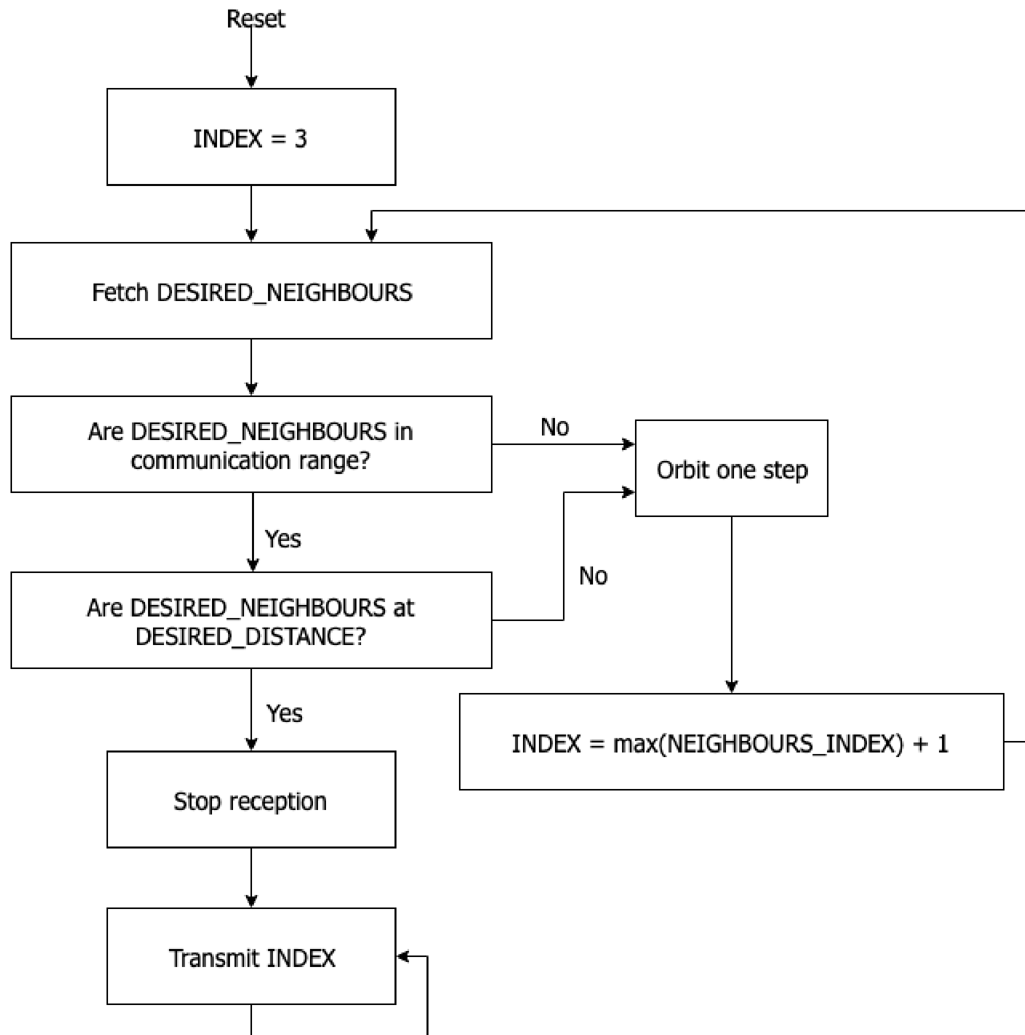


Figure 7.1: Flowchart for shape formation algorithm

7.3 Discussion

Figure 7.2 and 7.3 helps in visualizing the shape formation process for shape matrix (7.2), which corresponds to a line of width 2. Black circles indicate guide robots which continuously transmit their `Index`, whereas grey circles indicate the oncoming builder robot. Shaded circle corresponds to

a builder transforming into a guide. A builder is always in listener mode, whereas a guide is always in speaker mode of communication. Dotted lines trace the path of builder.

When first builder initialized with `Index=3` starts its journey, as shown in Figure 7.2, it never faces a situation which requires `Index` update and hence, it ultimately reaches the desired location corresponding to `Index=3` and transforms into a guide robot.

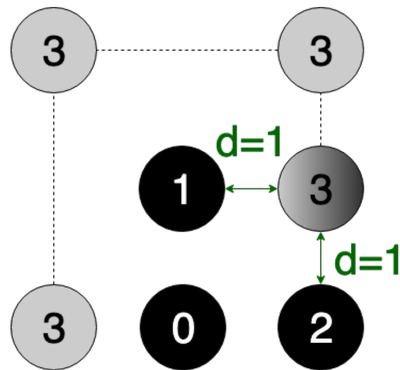


Figure 7.2: First builder taking its position

When second builder arrives with `Index=3`, as n in Figure 7.3, it continues on its journey to occupy the desired location coresponding to `Index=3`, but when it comes in the communication range of new guide with `Index=3`, it realizes that it needs to update its `Index` to `4`. Following which, it travels to the desired location corresponding to `Index=4` and transforms into a guide.

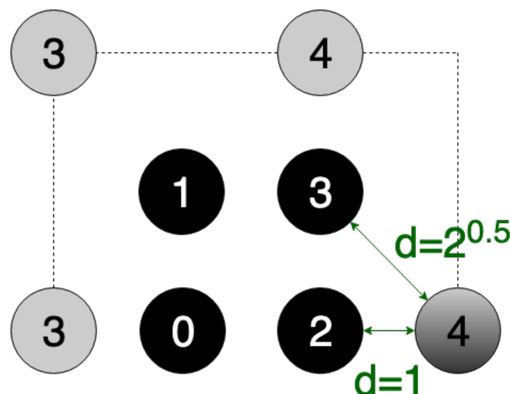


Figure 7.3: Second builder taking its position

7.4 Demonstration

`EPSILON_MARGIN` and `MOTOR_ON_DURATION` plays an essential role in determining the stability and accuracy of shape formation. For large `MOTOR_ON_DURATION`, it is likely for builder to go beyond its desired location and continue orbiting whereas for larger `EPSILON_MARGIN`, we get distortion in

shape. Moreover, we can not choose very small EPSILON_MARGIN due to error in measurements. One way to improve accuracy of shape is by adaptively decreasing the MOTOR_ON_DURATION of builder when it comes in the comunication range of desired neighbours. The two parameters discussed are borrowed from previous chapter. For detailed understanding of these parameters, readers are motivated to go through code in Appendix B

Figure 7.4 provides the link to video of shape formation in action. Shape matrix (7.2) was fed to each builder robots to form a rectangle breadth=2 and length=3. Values for EPSILON_MARGIN and MOTOR_ON_DURATION were assigned to 5 and 500 respectively. For abovementioned values of parameters, the desired shape formation took place in roughly 10 minutes with observable distortions in shape. As the shape becomes larger and larger, accumulations of these errors could prevent a builder from localization, given our naive implementation of the algorithm. We also experimented with smaller values of EPSILON_MARGIN, namely 2, but because of measurement noise, builders failed to establish themselves at desired location with large probability and continued orbiting around guides. Same observations were made for large values of MOTOR_ON_DURATION, namely 1000, as we had implemented hard delays. Although, larger values decreased the time to reach near the desired region, builders failed to localize within desired error margin. This could be attributed to large displacement being made before every localization step.

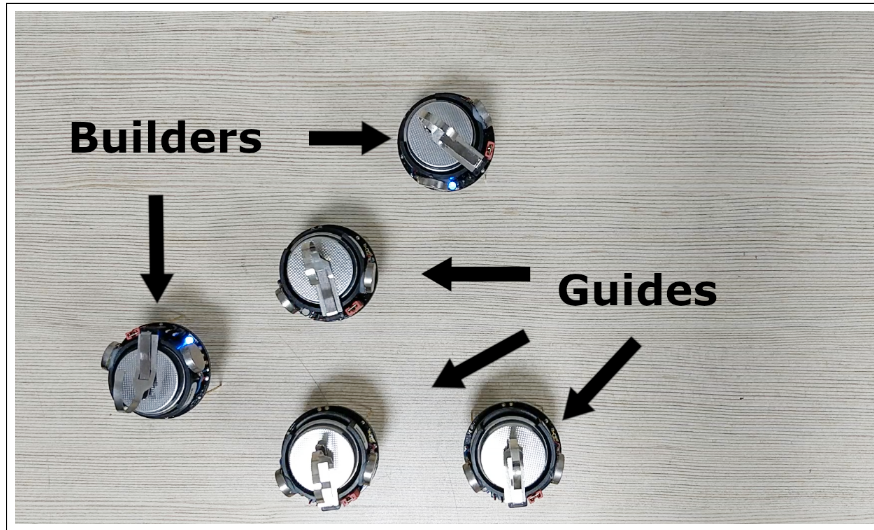


Figure 7.4: Shape formation by kilobots (Shape: Rectangle of breadth=2 and length=3)

Based on our discussion, inclusion of following measures would boost the performance:

- Adaptively decreasing the MOTOR_ON_DURATION as the builder reaches closer to its target location. Doing this would significantly improve the runtime of shape formation and accuracy of localization if calibrated with proper choice of EPSILON_MARGIN.
- Conversion of hard delays to soft delays, thereby, decreasing the overall run time.
- Use of optimization based approach for localization to incorporate uncertainity in measurements.

Chapter 8

Conclusion

During our lab work on Kilobotics, we developed essential building blocks for implementation of shape formation algorithm. In the process, we tested algorithms for efficient orbiting, algorithm for escaping an obstacle, algorithm for orbiting multiple stars, and lastly, we also implemented a rudimentary shape formation algorithm for Kilobots. There's a lot which needs to be achieved in terms of robustness in performance and integration of individual building blocks. Further, one may also pursue development of macros to generate shape matrix from a bitmap image.

8.1 Acknowledgement

We would like to thank Tejdeep Reddy for his teaching assistance and lab staffs for maintaining a healthy number of working robots. Moreover, we would like to thank Prof. Leena Vachhani for her coursework on Automation and Feedback which motivated us to approach the problem using finite state machine. Lastly, thanks to Prof. Leena Vachhani, Prof. Arpita Sinha and Adwaith Vijayakumar for their generous coperation in preponing our presentation dates.

Appendix A

Code for orbiting after escape algorithm

```
1 #include <kilolib.h>
2
3 //----- PARAMETERS FOR SHAPE FORMATION / COMMUNICATION -----
4 #define DESIRED_DISTANCE 65
5 #define EPSILON_MARGIN 5
6 #define TOO_CLOSE_DISTANCE 45
7 #define MOTOR_ON_DURATION 500
8 #define THRESHOLD_ROTATE 12
9 #define NUMBER_COMMUNICATION 1
10
11 //----- DEFINE MOTION -----
12 #define FORWARD 0
13 #define LEFT 1
14 #define RIGHT 2
15
16 //----- DEFINE STATE -----
17 #define MEASURE_DISTANCE 0
18 #define DISTANCE_CHECK_ORBIT 1
19 #define TOO_CLOSE 2
20 #define TOO_CLOSE_ESCAPE 3
21 #define GREATER_THAN_DESIRED 4
22 #define SMALLER_THAN_DESIRED 5
23 #define WAIT 6
24 #define MEASURE_AGAIN 7
25 #define WAIT_AGAIN 8
26 #define UPDATE_MINIMUM_DISTANCE 9
27 #define MEASURE_ESCAPE 10
28 #define WAIT_ESCAPE 11
```

```
29 #define FINISH 12
30
31 //----- VARIABLE DECLARATION -----
32 int state, last_state, distance, last_distance, message_rx_status,
       min_distance, temp;
33 message_t message;
34
35 void measure_distance()
36 {
37     message_rx_status=0;
38 }
39
40 message_t *message_tx()
41 {
42     return &message;
43 }
44
45 //----- ROUTINE FOR RECEPTION -----
46 void message_rx(message_t *m, distance_measurement_t *d)
47 {
48     if(message_rx_status == 0)
49     {
50         distance = 1000;
51     }
52     if(message_rx_status != NUMBER_COMMUNICATION)
53     {
54         temp = estimate_distance(d);
55         if(temp < distance)
56         {
57             distance = temp;
58         }
59         message_rx_status++;
60     }
61 }
62
63 //----- ROUTINE FOR MOTION -----
64 void move(int direction)
65 {
66     switch(direction)
67     {
68         case FORWARD:
69             spinup_motors();
70             set_motors(kilo_straight_left, kilo_straight_right);
71             delay(MOTOR_ON_DURATION);
72             set_motors(0, 0);
73             break;
```

```
74          case LEFT:
75              spinup_motors();
76              set_motors(kilo_straight_left, 0);
77              delay(MOTOR_ON_DURATION);
78              set_motors(0, 0);
79              break;
80          case RIGHT:
81              spinup_motors();
82              set_motors(0, kilo_straight_right);
83              delay(MOTOR_ON_DURATION);
84              set_motors(0, 0);
85              break;
86      }
87  }
88
89  void setup()
90  {
91      //----- RESET FINITE STATE MACHINE -----
92      state = MEASURE_DISTANCE;
93      set_color(RGB(0,0,1));
94  }
95
96  void loop()
97  {
98      switch(state)
99      {
100         case MEASURE_DISTANCE:
101             //----- INITIATE RECEPTION -----
102             message_rx_status = 0;
103             state = WAIT;
104             break;
105         case WAIT:
106             if(message_rx_status == NUMBER_COMMUNICATION)
107             {
108                 state = DISTANCE_CHECK_ORBIT;
109             }
110             break;
111         case DISTANCE_CHECK_ORBIT:
112             //----- CHECK THE REGION OF PLANET -----
113             if(distance < TOO_CLOSE_DISTANCE)
114             {
115                 state = TOO_CLOSE;
116                 min_distance = distance;
117             }
118             else
119             {
```

```
120                 if(distance > DESIRED_DISTANCE)
121                 {
122                     state = GREATER_THAN_DESIRED;
123                 }
124                 else
125                 {
126                     state = SMALLER_THAN_DESIRED;
127                 }
128             }
129             break;
130         case TOO_CLOSE:
131             set_color(RGB(1,0,0));
132             move(LEFT);
133             state = MEASURE_AGAIN;
134             break;
135         case MEASURE_AGAIN:
136             message_rx_status = 0;
137             state = WAIT_AGAIN;
138             break;
139         case WAIT_AGAIN:
140             if(message_rx_status == NUMBER_COMMUNICATION)
141             {
142                 state = UPDATE_MINIMUM_DISTANCE;
143             }
144             break;
145         case UPDATE_MINIMUM_DISTANCE:
146             //----- UPDATE MINIMUM DISTANCE WHILE TRYING TO ESCAPE
                    THE OBSTACLE -----
147             if(distance < min_distance)
148             {
149                 min_distance = distance;
150             }
151             //----- CHECK IF PLANET IS ORIENTED AWAY FROM OBSTACLE
                    -----
152             if(distance - min_distance > THRESHOLD_ROTATE)
153             {
154                 state = TOO_CLOSE_ESCAPE;
155             }
156             else
157             {
158                 state = TOO_CLOSE;
159             }
160             break;
161         case TOO_CLOSE_ESCAPE:
162             //----- CHECK IF PLANET IS NEAR THE ORBITING DISTANCE
                    -----
```

```
163             if(distance > DESIRED_DISTANCE - EPSILON_MARGIN)
164             {
165                 state = MEASURE_DISTANCE;
166             }
167             else
168             {
169                 move(FORWARD);
170                 state = MEASURE_ESCAPE;
171             }
172             break;
173         case MEASURE_ESCAPE:
174             message_rx_status = 0;
175             state = WAIT_ESCAPE;
176             break;
177         case WAIT_ESCAPE:
178             if(message_rx_status == NUMBER_COMMUNICATION)
179             {
180                 state = TOO_CLOSE_ESCAPE;
181             }
182             break;
183         case GREATER_THAN_DESIRED:
184             //----- ROUTINE FOR CLOCKWISE ORBITING -----
185             move(RIGHT);
186             set_color(RGB(0,1,0));
187             state = MEASURE_DISTANCE;
188             break;
189         case SMALLER_THAN_DESIRED:
190             //----- ROUTINE FOR CLOCKWISE ORBITING -----
191             move(LEFT);
192             set_color(RGB(1,0,0));
193             state = MEASURE_DISTANCE;
194             break;
195         default:
196             break;
197     }
198     last_state = state;
199 }
200
201
202
203 int main()
204 {
205     kilo_init();
206     //----- INITIALIZE RECEPTION -----
207     kilo_message_rx = message_rx;
208     kilo_start(setup, loop);
```

```
209
210     return 0;
211 }
```

Appendix B

Code for shape formation algorithm

```
1 #include <kilolib.h>
2
3 //----- PARAMETERS FOR SHAPE FORMATION / COMMUNICATION -----
4 #define DESIRED_DISTANCE 65
5 #define EPSILON_MARGIN 5
6 #define MOTOR_ON_DURATION 500
7 #define NUMBER_COMMUNICATION 3
8
9 //----- DEFINE MOTION -----
10 #define FORWARD 0
11 #define LEFT 1
12 #define RIGHT 2
13
14 //----- DEFINE STATE -----
15 #define NEIGHBOURS_IN_RANGE 1
16 #define COMPARE_DESIRED_DISTANCE 2
17 #define ORBIT_AND_UPDATE_INDEX 3
18 #define FINISH 4
19 #define INFINITY 5
20
21 //----- VARIABLE DECLARATION -----
22 message_t message;
23 int state, message_rx_status, temp, index=3, check = 0, x, y,
    max_index=2, message_sent;
24 float distance, last_distance, min_distance;
25
26 //----- ARRAYS FOR STORING NEIGHBOURS INFORMATION -----
27 int reception_id[3] = {0,0,0};
```

```
28 float reception_distance[3] = {0,0,0};
29
30 //----- SHAPE MATRIX -----
31 int neighbours[8][2] =
     {{0,0},{0,0},{0,0},{1,2},{2,3},{3,4},{4,5},{5,6}};
32 float distance_multiplier[8][2] = {{0,0}, {0,0}, {0,0}, {1,1},
     {1,1.4}, {1,1}, {1,1.4},{1,1}};
33
34 message_t *message_tx()
35 {
36     return &message;
37 }
38
39 //----- ROUTINE FOR TRANSMISSION -----
40 void message_tx_success()
41 {
42     message_sent = 1;
43     set_color(RGB(1, 0, 1));
44     delay(100);
45     set_color(RGB(0, 0, 0));
46 }
47
48 //----- ROUTINE FOR RECEPTION -----
49 void message_rx(message_t *m, distance_measurement_t *d)
50 {
51     if(message_rx_status == 0)
52     {
53         distance = 1000;
54     }
55     if(message_rx_status != NUMBER_COMMUNICATION)
56     {
57         temp = estimate_distance(d);
58         //----- CALCULATE MINIMUM DISTANCE -----
59         if(temp < distance)
60         {
61             distance = temp;
62         }
63         //----- STORE RECEPTION ID -----
64         reception_id[message_rx_status] = (*m).data[0];
65         //----- MAXIMUM INDEX IN CURRENT COMMUNICATION -----
66         if(reception_id[message_rx_status]>max_index)
67         {
68             max_index=reception_id[message_rx_status];
69         }
70         //----- STORE RECEPTION DISTANCE -----
71         reception_distance[message_rx_status] = temp;
```

```
72          message_rx_status++;
73      }
74  }
75
76  void measure_distance()
77  {
78      message_rx_status=0;
79  }
80
81  //----- ROUTINE FOR MOTION -----
82  void move(int direction)
83  {
84      switch(direction)
85      {
86          case FORWARD:
87              spinup_motors();
88              set_motors(kilo_straight_left, kilo_straight_right);
89              delay(MOTOR_ON_DURATION);
90              set_motors(0, 0);
91              break;
92          case LEFT:
93              spinup_motors();
94              set_motors(kilo_straight_left, 0);
95              delay(MOTOR_ON_DURATION);
96              set_motors(0, 0);
97              break;
98          case RIGHT:
99              spinup_motors();
100             set_motors(0, kilo_straight_right);
101             delay(MOTOR_ON_DURATION);
102             set_motors(0, 0);
103             break;
104         default:
105             break;
106     }
107 }
108
109 void setup()
110 {
111     //----- RESET FINITE STATE MACHINE -----
112     state = NEIGHBOURS_IN_RANGE;
113     set_color(RGB(0,0,1));
114 }
115
116 void loop()
117 {
```

```
118     switch(state)
119     {
120         case NEIGHBOURS_IN_RANGE:
121             state = COMPARE_DESIRED_DISTANCE;
122             //----- INITIATE RECEPTION -----
123             message_rx_status = 0;
124             break;
125         case COMPARE_DESIRED_DISTANCE:
126             if(message_rx_status == NUMBER_COMMUNICATION)
127             {
128                 for(int i=0; i<NUMBER_COMMUNICATION; i++)
129                 {
130                     for(int j=i+1; j<NUMBER_COMMUNICATION; j++)
131                     {
132                         //----- CHECK IF DESIRED NEIGHBOURS IN
                               RANGE -----
133                         if(reception_id[i] == neighbours[index][0]
                               && reception_id[j] ==
                               neighbours[index][1])
134                         {
135                              x = i;
136                              y = j;
137                              check = 1;
138                              break;
139                         }
140                     }
141                     if(check == 1)
142                     {
143                         break;
144                     }
145                 }
146 
147                 //----- CHECK IF DESIRED NEIGHBOURS AT DESIRED
                       DISTANCE -----
148                 if((check == 1) && (reception_distance[x] >
                       distance_multiplier[index][0] * DESIRED_DISTANCE
                       - EPSILON_MARGIN && reception_distance[x] <
                       distance_multiplier[index][0] *
                       DESIRED_DISTANCE+EPSILON_MARGIN) &&
                       (reception_distance[y] >
                       distance_multiplier[index][1] *
                       DESIRED_DISTANCE-EPSILON_MARGIN &&
                       reception_distance[y] <
                       distance_multiplier[index][1] * DESIRED_DISTANCE
                       + EPSILON_MARGIN))
149                 {
```

```
150                     state = FINISH;
151                 }
152                 else
153                 {
154                     state = ORBIT_AND_UPDATE_INDEX;
155                 }
156             }
157             break;
158         case ORBIT_AND_UPDATE_INDEX:
159             //----- ALGORITHM FOR ORBITING CLOCKWISE -----
160             if(distance > DESIRED_DISTANCE)
161             {
162                 set_color(RGB(1,0,0));
163                 move(RIGHT);
164                 state = NEIGHBOURS_IN_RANGE;
165             }
166             else
167             {
168                 set_color(RGB(0,0,1));
169                 move(LEFT);
170                 state = NEIGHBOURS_IN_RANGE;
171             }
172             //----- UPDATE INDEX IF REQUIRED -----
173             if(max_index+1>index)
174             {
175                 index=max_index+1;
176             }
177             break;
178         case FINISH:
179             set_color(RGB(0,1,0));
180             //----- START TRANSMISSION AFTER DESIRED LOCATION IS
                    ACHIEVED -----
181             message.type = NORMAL;
182             message.data[0] = index;
183             message.crc = message_crc(&message);
184             kilo_message_tx = message_tx;
185             kilo_message_tx_success = message_tx_success;
186             state = INFINITY;
187             break;
188         case INFINITY:
189             break;
190         default:
191             break;
192     }
193 }
194
```

```
195 int main()
196 {
197     kilo_init();
198     //----- INITIALIZE RECEPTION -----
199     kilo_message_rx = message_rx;
200     kilo_start(setup, loop);
201     return 0;
202 }
```