Exploration of a Finite Area & Object Tracking with Using a 2 DOF Miniature Robot

Vincent Anderson, Muhamad Irfan Bin Muhd Ramiza, Jr
School of Electrical and Electronic Engineering, The University of Sheffield, UK
Email: vanderson1@sheffield.ac.uk, mibinmuhdramiza1@sheffield.ac.uk

***Abstract*—Task 1 required the E-Puck2 robot to explore an obstacle-filled area without getting stuck. Our strategy used front infrared sensors for wall detection, turning away from obstacles, and performing random turns when stuck. The implementation focused on simplified code and iterative sensor calibration. The final behavior enabled smooth navigation and avoidance of loops. The test confirmed consistent performance, achieving a perfect score in line with the assessment criteria. Task 2 was to detect an object within the sensor distance and to detect the object and move towards it when it moved further away. Task 2 required the E-puck2 robot to detect a object within a reasonable proximity and then to move towards the object and track its movements as it moves away from the robot whilst not making contact. The code written focused on three different states which were search, approach and chase to achieve the aforementioned task. The robots performance was satisfactory in this task able to search, approach and chase the robot but had issues with close proximity performance.**

I. Strategies

A. Task 1 - Exploration of a Finite Area

When the task was first presented complex algorithms using mapping techniques with path finding and using all sensors provided would provide the best outcome for the robot. However, this solution is overcomplicated and is beyond the scope of the project in terms of the task as a small set of if statements could achieve the tasks specifications. Using a finite state machine(FSM) with 4 different states: forwards motion, Left turn, Right Turn and random turn. The robots base movements are rotation and longitudinal, so it would make sense for the robot to move in its longitudinal direction and then rotate when certain conditions were met. Due the movement systems used this allows for the amount of sensors and input data to be greatly reduced only needing to use the TOF sensor to detect objects in front of the robot and then a pair of peripheral infrared sensors to detect which direction would be most optimal to move after an object is detected. The conditions mentioned previously would then become the main algorithm for which motion is decided of the robot and can be implemented using simple if and else statements and should result in a random motion within the area without colliding as the sensors are not fully accurate. One of the tests described scenario where the robot would become stuck in certain areas and the strategy is to then chose a random direction and try to go in that direction. A random direction approach was chosen because the TOF sensors seems the most reliable and by trying to go in the random direction of the TOF sensors would make sure the robot does not collide.

B. Task 2 - Object Tracking

The control strategy allows the e-puck2 robot to find, approach, and pursue a cylindrical object collision-free, using a finite state machine (FSM) with three states: Search, Approach, and Chase [3]. During testing the object was placed in random location near the e-puck2 robot,with the Graduate Teaching Assistant (GTA) moving it straight and curve trajectories, challenging the robot's adaptability for diverse initial condition. The three states were chosen by using aggregation of the problem into sequential objectives and it into simple manageable problems[3]. In Search, the robot spins until an object is sensed by any of the eight proximity sensors which is has 360 degree coverage and give directional information as these sensors are spaced around the perimeter of the robot giving the largest range. The Approach state orients the robot to the object by modulating motor speeds according to the angle estimated from the sensor having the highest reading, mapped to 0–360 degrees, using a proportional control law where turning speed `vt = k · |theta - 180|`, with k scaling to 800 steps/s for 180-degree deviations and 400 steps/s for 45-degree deviations, ensuring smooth orientation. In Chase, the ToF distance sensor keeps a 20–50 mm distance as it has a longer range and pointed directly in the front direction, with speed modulation to follow the moving object. Turning speed is proportional to angle deviation (e.g., 800 steps/s for 180-degree turns, 400 steps/s for 45-degree turns) for smoothness. The selector switch changes base speed, offering flexibility.This strategy maximizes sensor synergy, leveraging proximity sensors' wide coverage and and ToF's accuracy, while the FSM's state transition(e.g.: its search to approach when proximity exceeds threshold) form a robust control loop, adaptable to real world variations and dynamic object movement and thus achieving smooth and efficient tracking.

II. Implementation

A. Task 1 - Exploration of a Finite Area

First the base code was imported to give a baseline code that can be used and is known to work and activates all standard sensors and motors to be used. The main code utilizes a infinite loop(while(1)) to continually read sensor data and decides the robots movement in real time.
The code will then get data, parse it into an integer value from the TOF and infrared sensor and defined them to a directional

variable name: TOF - Front; IR6 - Left; IR7 - Front Left; IR0 - Front Right; IR1 - Right. Predefined threshold constant variables are created which can later be adjusted during testing are then used to compare the sensors values to decide the movement. When all the variables are above threshold values the robot will continue to move in a straight line.

- All value are below the threshold values: the robot will execute the random turn function
- When the left values are below the threshold value: the robot will perform a right turn
- When the right values are below the threshold value: the robot will perform a left turn
- Else: the robot will continue to move in a straight line.

The random turn function will choose a random direction to turn and a random time to turn using random number generators from a built in function to break out of patterned behavior. After these if statements are executed a wait function is called upon to allow the robot to continue in that direction as the movement would be too erratic and wouldn't allow smooth motion. Whilst testing, the predefined threshold values were adjusted one at a time using tests to make sure the robot can explore the entire area, have smooth motion and not contact any objects.

B. *Task 2 - Object Tracking*

The Task 2 strategy is implemented in C using the e-puck2 library, with the source code in the appendix (lines 1–156). The implementation initializes hardware components, including eight proximity sensors, a time-of-flight (ToF) sensor, motors, LEDs, and serial communication modules, using standard library functions (lines 2–19, 130-141)[2]. The main code within an infinite loop (while(1), lines 144-147), executing the follow_object function (lines 45–126) every 50 ms (chThdSleepMilliseconds, line 146) to implement a finite state machine (FSM) with three states: Search, Approach, and Chase, defined via an enumeration (line 36).The proximity sensor readings are retrieved using get_calibrated_prox (lines 52-58), identifying the sensor with the highest value to estimate the object's angle (0, 45, 90, 135, 180, 225, 270, 315) through the estimate_angle function (lines 39–42). The ToF sensor (VLS3LOX_get_dist_mm, line 61) measures object distance, and the selector switch (get_selector, line 64) scales the base speed dynamically (line 65). Constants (PROX_THRESHOLD, TARGET_DIST_MIN, TARGET_DIST_MAX, lines 22–24) control state transitions and motor behavior. In the Search state, if no proximity reading exceeds the threshold, the robot rotates by setting opposing motor speeds (left_motor_set_speed, right_motor_set_speed, lines 88–89). In Approach, motor speeds are adjusted based on the object's angle to orient the robot (lines 102–106). In Chase, speeds are modulated to maintain a 20–50 mm distance, reducing to 80% of base speed if too close, less than 20 mm) or increasing to 150% if too far (more than 50 mm, lines 116-–118), with differential turning for alignment (lines 119-123). LEDs provide visual feedback for each state (set_body_led, set_rgb_led, lines 82–83, 94-95, 110–111), and sensor data is logged via Bluetooth (e_send_uart1_char, lines 75—77) for debugging. A calibration routine (calibrate_ir, line 136) ensures reliable proximity sensor performance under varying ambient light conditions. During testing, constants were iteratively tuned to optimize object detection, distance maintenance, and smooth motion, with the 50 ms loop delay minimizing CPU usage while ensuring responsive behavior. However, close-start scenarios required further adjustments to enhance collision avoidance.

III. Results and Discussion

A. *Task 1 - Exploration of a Finite Area*

In Task 1, the robot achieved a 100% score by fully exploring the environment without collisions, stalls, or becoming trapped. Its motion remained smooth throughout, demonstrating the effectiveness of the autonomous exploration strategy. This success resulted from thorough testing and iterative code refinement before the assessment. Proximity sensors were calibrated to keep the robot close to obstacles without touching them. The finite state machine (FSM) guiding the behavior was tuned for robust obstacle avoidance and navigation. Some inefficiencies were noted particularly looped or repetitive movements caused by random direction selection when no clear path was present. Though this ensured full coverage, it increased exploration time. Future improvements could involve real-time map building using a dynamically allocated grid to track obstacles and space. Coupled with a path-planning algorithm, this could improve efficiency and reduce redundancy. However, noisy sensor data may require averaging or filtering, and external factors could still affect performance.

B. *Task 2 - Object Tracking*

In task 2 a score of 30/40 (75%) was achieved, full marks were obtained for "Approaching the object", "Chasing the Object" and "Speed & Efficiency" however no marks were given for "Collision Avoidance". The task was carried out on a table using a 10 cm cylindrical object. The robot changed from Search to Approach within 2 seconds, detecting the object from any direction using proximity sensors. In Chase, it maintained 20-50mm clearance for straight and curved trajectories (up to 90 degrees), smooth turns (180-degree turns in about 1.5 seconds at 800 steps/s, 45-degree turns within 0.8 seconds at 400 steps/s). The selector determined speeds (200–600 steps/s), and the LEDs signaled states. However, collisions were generated when the object was initially less than 20 mm in diameter, since the decrease in speed (80% of the base speed) was insufficient. This limitation, inherent in close-start scenarios, limits collision avoidance. Proposed improvements include stopping the robot when it is less than 20 mm and fusing proximity and ToF data. Calibration ensured reliability, but sensor noise caused minor jitter, suggesting low-pass filtering.

Acknowledgment

Dr Roselina Arelhi & the School of Electrical and Electronic Engineering for providing resources and learning material to conduct this task

Appendix A
Code

Source code from E-Puck2

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "ch.h"
#include "hal.h"
#include "memory_protection.h"
#include <main.h>
#include "leds.h"
#include "spi_comm.h"
#include "sensors/proximity.h"
#include "sensors/VL53L0X/VL53L0X.h"
#include "motors.h"

messagebus_t bus;
MUTEX_DECL(bus_lock);
CONDVAR_DECL(bus_condvar);

#define FRONT_THRESHOLD 5 // mm
#define PROX_THRESHOLD 600 // Proximity sensor threshold 600
#define TURN_SPEED 600  // Speed for turning
#define MOVE_SPEED 800  // Speed for moving forward
#define TURN_TIME 300 // amount of time for turning

void turn_randomly(void) {
   int random_turn = rand() % 2;
   int random_time = rand() % 501; // 0 = left, 1 = right
   if (random_turn == 0) {
     left_motor_set_speed(-TURN_SPEED);
     right_motor_set_speed(TURN_SPEED);
   } else {
     left_motor_set_speed(TURN_SPEED);
     right_motor_set_speed(-TURN_SPEED);
   }
   chThdSleepMilliseconds(random_time); // Short turn duration
}

int main(void) {
   halInit();
   chSysInit();
   mpu_init();

   // Initialize Sensors & Motors
   messagebus_init(&bus, &bus_lock, &bus_condvar);
   proximity_start(0);
   calibrate_ir();
   VL53L0X_start();
   clear_leds();
   spi_comm_start();
   motors_init();

   while (1) {
     int front_dist = VL53L0X_get_dist_mm();
     int prox_left = get_calibrated_prox(6);
     int prox_right = get_calibrated_prox(1);
     int prox_front_left = get_calibrated_prox(7);
     int prox_front_right = get_calibrated_prox(0);

     // Robot Surrounded at the front
     if (front_dist < FRONT_THRESHOLD &&
         prox_front_left < PROX_THRESHOLD &&
         prox_front_right < PROX_THRESHOLD &&
         prox_left < PROX_THRESHOLD && prox_right < PROX_THRESHOLD) {
       turn_randomly();
     }
     // Side obstacles detected - adjust path
     else if (prox_left < PROX_THRESHOLD) {
       left_motor_set_speed(TURN_SPEED);
       right_motor_set_speed(-TURN_SPEED);
       chThdSleepMilliseconds(TURN_TIME);
     }
     else if (prox_right < PROX_THRESHOLD) {
       left_motor_set_speed(-TURN_SPEED);
       right_motor_set_speed(TURN_SPEED);
       chThdSleepMilliseconds(TURN_TIME);
     }
     // Default forward movement
     else {
       left_motor_set_speed(MOVE_SPEED);
       right_motor_set_speed(MOVE_SPEED);
     }
   }
}

#define STACK_CHK_GUARD 0xe2dee396
uintptr_t __stack_chk_guard = STACK_CHK_GUARD;

void __stack_chk_fail(void) {
   chSysHalt("Stack smashing detected");
}
```

Listing 1. Code - Task 1

```c
// Include standard libraries
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// Include ChibiOS headers
#include "ch.h"
#include "hal.h"
#include "memory_protection.h"

// Include hardware and sensor-specific libraries
#include "leds.h"
#include "spi_comm.h"
#include "sensors/proximity.h"
#include "sensors/VLS3LOX/VLS3LOX.h"
#include "motors.h"
#include "selector.h"
#include "epuck1x/uart/e_uart_char.h"
#include "serial_comm.h"

// Thresholds and constants for behavior logic
#define PROX_THRESHOLD 150 // Minimum proximity reading to trigger interest
#define TARGET_DIST_MIN 20 // Minimum desired distance to object (mm)
#define TARGET_DIST_MAX 50 // Maximum desired distance to object (mm)
#define BASE_SPEED 400 // Default motor speed
#define MAX_TURN_SPEED 800 // Maximum speed difference when turning
#define MIN_TURN_SPEED 400 // Minimum speed difference when turning
#define SENSOR_COUNT 8 // Number of proximity sensors

// Declare message bus for inter-thread communication (ChibiOS feature)
messagebus_t bus;
MUTEX_DECL(bus_lock);
```

```c
CONDVAR_DECL(bus_condvar);

// State machine for behavior logic
typedef enum { SEARCH, APPROACH, CHASE } State;

// Convert sensor index to approximate angle in degrees
float estimate_angle(int max_sensor) {
   float angles[SENSOR_COUNT] = {0, 45, 90, 135, 180, 225, 270, 315};
   return angles[max_sensor];
}

// Main object-following behavior function
void follow_object(void) {
   static State state = SEARCH; // Start in SEARCH state

   // Arrays to hold sensor readings
   int prox[SENSOR_COUNT], max_prox = 0, max_sensor = 0;

   // Read proximity sensors and find the one with the highest value
   for (int i = 0; i < SENSOR_COUNT; i++) {
      prox[i] = get_calibrated_prox(i);
      if (prox[i] > max_prox) {
         max_prox = prox[i];
         max_sensor = i;
      }
   }

   // Read distance from Time-of-Flight sensor
   uint16_t dist = VLS3LOX_get_dist_mm();

   // Use selector to vary base speed (for easy tuning)
   int selector = get_selector();
   int base_speed = BASE_SPEED + (selector * 25);

   // Get angle from sensor index
   float angle = estimate_angle(max_sensor);

   // Determine how much to turn (0 if object is directly ahead)
   float turn_factor = fabs(angle - 180) / 180;
   int turn_speed = MIN_TURN_SPEED + (MAX_TURN_SPEED - MIN_TURN_SPEED) * turn_factor;

   // Print debug data to UART
   char str[100];
   int str_length = sprintf(str, "Prox: %d, Dist: %d, State: %d\n", max_prox, dist, state);
   e_send_uart1_char(str, str_length);

   // State machine to control robot behavior
   switch (state) {
      case SEARCH:
         set_body_led(0); // Body LED off
         set_rgb_led(LED2, 10, 0, 0); // Red LED
         if (max_prox >= PROX_THRESHOLD)
            state = APPROACH;
         else {
            // Rotate to scan environment
            left_motor_set_speed(MIN_TURN_SPEED);
            right_motor_set_speed(-MIN_TURN_SPEED);
         }
         break;

      case APPROACH:
         set_body_led(1); // Body LED on
         set_rgb_led(LED2, 0, 10, 0); // Green LED
         if (dist <= TARGET_DIST_MAX && dist >= TARGET_DIST_MIN)
            state = CHASE;
         else if (max_prox < PROX_THRESHOLD)
            state = SEARCH;
         else {
            // Adjust motor speeds to turn toward the object
            int left_speed = base_speed - (angle > 180 ? turn_speed : 0);
            int right_speed = base_speed - (angle < 180 ? turn_speed : 0);
            left_motor_set_speed(left_speed);
            right_motor_set_speed(right_speed);
         }
         break;

      case CHASE:
         set_body_led(1);
         set_rgb_led(LED2, 0, 0, 10); // Blue LED
         if (max_prox < PROX_THRESHOLD || dist > TARGET_DIST_MAX)
            state = SEARCH;
         else {
            // Speed adjustment based on distance to object
            int speed_adjust = (dist < TARGET_DIST_MIN) ? base_speed * 0.8 :
                               (dist > TARGET_DIST_MAX) ? base_speed * 1.5 :
                               base_speed;
            int left_speed = speed_adjust - (angle > 180 ? turn_speed : 0);
            int right_speed = speed_adjust - (angle < 180 ? turn_speed : 0);
            left_motor_set_speed(left_speed);
            right_motor_set_speed(right_speed);
         }
         break;
   }
}

// Main program entry point
int main(void) {
   halInit(); // Hardware abstraction layer init
   chSysInit(); // ChibiOS kernel init
   mpu_init(); // Memory protection init
   messagebus_init(&bus, &bus_lock, &bus_condvar); // Init message bus

   proximity_start(); // Start proximity sensor
   calibrate_ir(); // Calibrate IR proximity sensors
   VLS3LOX_start(); // Start Time-of-Flight sensor
   clear_leds(); // Turn off LEDs
   spi_comm_start(); // Start SPI communication
   motors_init(); // Initialize motors
   serial_start(); // Start serial communication

   // Main control loop
   while (1) {
      follow_object(); // Run behavior function
      chThdSleepMilliseconds(50); // Small delay to prevent CPU overuse
   }
}

// Stack smashing protection (ChibiOS-specific)
#define STACK_CHK_GUARD 0xe2dee396
uintptr_t __stack_chk_guard = STACK_CHK_GUARD;
void __stack_chk_fail(void) {
   chSysHalt("Stack smashing detected");
}
```

Listing 2. Code - Task 2

REFERENCES

[1] w3schools, “C++ Tutorial” W3schools.com, 2020. https://www.w3schools.com/cpp/default.asp

[2] G. Di Sirio, ChibiOS/RT Reference Manual, ChibiOS Project, Rev. 19.1.4, Mar. 2019. https://www.chibios.org/dokuwiki/doku.php?id=chibios:documentation:start

[3] R. Balogh and D. Obdrzalek, ”Using Finite State Machines in Introductory Robotics: Methods and Applications for Teaching and Learning”, 2019. [Online]. Available: https://www.researchgate.net/publication/327389755