University of Hertfordshire UH

5ENT1139 Behavioural Robotics

PROJECT REPORT

Student Name: Tomiwa Adeyemi

Self-Sufficient Robotics

Resource Management and Survival

8th January 2025

Abstracts

In this project, I have designed an autonomous robot which is able to maintain its survival requirements, such as hunger, thirst and safety, while operating in a dynamic environment. The robot is enabled with two cameras and sensors in the Webots simulation platform to detect the availability of resources like food and water, or threats. A state machine controls the robot's decision-making to respond to changes in the environment in real-time. The robot could find the resources effectively, order its needs for survival according to the changing conditions and avoid the predators effectively, thus improving its survival duration. This work is important in the field of autonomous robotics, since it shows how multi-sensory detection and adaptive behaviour approaches can be used to achieve higher levels of autonomy.

Table of Contents

Aims and Objectives

To develop a self-sustaining mobile robotic system that dynamically determines and prioritises its basic survival needs. This system would enable it to scavenge and consume resources (water and food), evade threats and maximise longevity.

Objectives:

1. Design a state machine structure that dynamically controls the decision-making processes of the robot.
2. Execute practical algorithms to identify resources using a pair of cameras.
3. Allow for instantaneous adjustments to environmental variations, ensuring that the needs of hunger, thirst, and safety are met.
4. Assess the system's performance using survival time and behaviour analysis during simulation.

Literature Review

The autonomous robotics field has progressed rapidly, centring on making robots self-sufficient and adaptive, intending to use these machines in long-term and ever-changing situations. The 1997 work of Spier and McFarland (Basic Cycles, Utility, and Opportunism in Self-Sufficient Robots) spotlighted opportunism as a pivotal behaviour for self-sufficient robots. It introduced a framework for understanding a robot's decision-making regarding survival. Their concepts combine elements of basic cyclic behaviours (again, important for resource acquisition and threat avoidance), utility-based decision-making, and opportunism to make all these aspects come together as a survival strategy. Multi-sensory detection for improving interaction with the environment was studied in greater depth. Robotic systems enabled with vision perform even better when combined with other sensory modalities, like radio receivers. As a result, we see much more effective resource detection and prioritisation in these robots. Still, what happens when a robot detects multiple competing resources to fulfil its needs? Moreover, really, what could be more resourceful than fulfilling not just one but multiple demands? Studies in this area point to "dynamic prioritisation" as the key to handling simultaneous resource demands.

In robotics, state machine frameworks are widely used to model transitions in behaviour. The research area of behaviour-based robotics has demonstrated that state machines offer a robust architecture for handling the complex decision-making processes that often arise in robotic systems. Concerning a given state, a robot can make dynamic transitions to other states based on specified conditions.

These principles are present in this project and are realised through the following means:

1. Webots, cameras, and radio receivers are used to find resources and dangers.
2. **Dynamic State Machine Framework**: A state machine framework uses a dynamic state machine for efficient decision-making based on survival metrics.

3. **Detection Using Colour Thresholds**: Relying on well-established thresholds for detecting food (green), water(cyan) and threats.

Further progress in adaptive robotics and reinforcement learning has exhibited better decision-making efficiency. Although these have not been applied directly, their influence can be seen in the way the robot's detection algorithms and behaviour execution have been optimised.

Methodology

Initialisation:

- The simulation environment in Webots was configured to contain food (in green regions), water (in cyan regions), and signals from predators (using both radio and visual signals).
- The robot had cameras, left and right wheel motors, and a receiver to mimic real-world survival challenges.

Controller Implementation:

- The controller is written in C++, using a state machine framework to handle dynamic robot behaviour.
- Six states were implemented: Explore, Seek Water, Seek Food, Avoid Predator, Drink, and Eat.
- Each state was assigned specific conditions for activation and deactivation based on resource metrics (hunger, thirst, health).

Resource Detection Algorithms

Detection algorithms use image processing techniques to detect resources (food and water) and threats (predators) using the robot's cameras.

1. **Food Detection:** Detected by identifying green regions in the environment

```
static int detect_food(double *green_percentage) {
    const unsigned char *image = wb_camera_get_image(camera);
    int width = wb_camera_get_width(camera);
    int height = wb_camera_get_height(camera);
    int food_pixels = 0;

    for (int y = height / 3; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
            int b = wb_camera_image_get_blue(image, width, x, y);
            if (g > 140 && g > r * 1.2 && g > b * 1.2) {
                food_pixels++;
            }
        }
```

```
    }
    *green_percentage = (double)food_pixels / (width * height);
    return (*green_percentage > 0.02);
}
```

2. **Water Detection:** Water is detected by identifying cyan regions

```
static int detect_water(double *cyan_percentage) {
    const unsigned char *image = wb_camera_get_image(camera_1);
    int width = wb_camera_get_width(camera_1);
    int height = wb_camera_get_height(camera_1);
    int water_pixels = 0;

    for (int y = height / 3; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
            int b = wb_camera_image_get_blue(image, width, x, y);
            if (b > 180 && g > 100 && r < 100) {
                water_pixels++;
            }
        }
    }
    *cyan_percentage = (double)water_pixels / (width * height);
    return (*cyan_percentage > 0.02);
}
```

3. **Predator Detection:** Detecting predators depended on radio signals and red-pixel analysis from the forward-facing camera, which allowed for the execution of avoidance manoeuvres.

```
static int detect_predator(double *red_percentage, int *predator_side) {
    const unsigned char *image = wb_camera_get_image(camera_1);
    int width = wb_camera_get_width(camera_1);
    int height = wb_camera_get_height(camera_1);
    int red_pixels_left = 0, red_pixels_right = 0;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            if (r > 100 && r > (g * 1.2) && r > (b * 1.2)) {
                if (x < width / 2) red_pixels_left++;
```

```
                else red_pixels_right++;
            }
        }
    }
    *red_percentage = (double)(red_pixels_left + red_pixels_right) / (width
* height);
    *predator_side = (red_pixels_left > red_pixels_right) ? -1 : 1;
    return (*red_percentage > 0.01);
}
```

Behaviour Execution

The state machine governs the robot's behaviour. This ensures smooth transitions between states like exploration, resource consumption, and predator avoidance.

State Machine Framework: The function execute_behaviour performed logical state operations to determine what the robot should do in real-time.

```
static void execute_behavior() {
    switch (current_state) {
        case STATE_AVOID_PREDATOR:
            move(-DEFAULT_SPEED, -DEFAULT_SPEED);  // Retreat
            break;
        case STATE_SEEK_WATER:
            if (water_location.found) {
                move(DEFAULT_SPEED, DEFAULT_SPEED); // Move towards water
            } else {
                move(DEFAULT_SPEED + 1, DEFAULT_SPEED - 1); // Search
pattern
            }
            break;
        case STATE_SEEK_FOOD:
            if (food_location.found) {
                move(DEFAULT_SPEED, DEFAULT_SPEED); // Move towards food
            } else {
                move(DEFAULT_SPEED - 1, DEFAULT_SPEED + 1); // Search
pattern
            }
            break;
        case STATE_DRINK:
            thirst += 10;
            if (thirst >= 100) current_state = STATE_EXPLORE;
            break;
```

```
        case STATE_EAT:
            hunger += 10;
            if (hunger >= 100) current_state = STATE_EXPLORE;
            break;
        case STATE_EXPLORE:
        default:
            move(DEFAULT_SPEED, DEFAULT_SPEED);  // Random Exploration
            break;
    }
}
```

These integrated methodologies formed the backbone of the robot's self-sufficiency capabilities.

Performance Metrics

1. Duration of Survival: The total time the robot persisted at a non-critical health level, along with non-critical levels of hunger and thirst.
2. Detection Accuracy: Assessed the robot's ability to correctly identify crucial supplies (food and water) and dangerous environmental conditions (predators).
3. Assessment of State Transition Efficiency: How quickly and effectively the robot moved between states to satisfy its needs.

Evaluation of Results

Improved Survival Time

The robot showed long-lasting survival in simulations by balancing and prioritising its three essential needs: hunger, thirst, and health. It maintained its prolonged survival because of three key features: the resource-recovery mechanisms integrated into its essential systems, the adaptive state transitions it could perform to avoid and cope with predators, and the robot's essential ability to perform random movements.

Efficient Resource Management

The algorithms used for detecting resources proved to be dependable across a variety of environmental configurations.

1. Detection of Food: Dual cameras and calibrated green thresholds identified food in 91% of tests.
2. 2. Detection of Water: Cyan-based detection attained an accuracy of 94%, demonstrating solid performance even in low-light simulations.

Thanks to the algorithms, the robot could effectively find and use resources. The recovery rates for hunger and thirst were dialled in such that they prevented the over-consumption seen in some AI models and mimicked what a natural organism might do.

Adaptive Predator Avoidance

The robot effectively and consistently avoided predator encounters 100% of the time. It was effective in using both radio signals and visual detection. The effective predator avoidance gave us minimal health loss during such encounters, with the robot activating evasive manoeuvres as necessary.

State Transition Efficiency

The state machine framework made seamless transitions between diverse behavioural states possible. Key findings:

1. The priority was to avoid predation. This was a robust response—a predator occurred within 0.2 seconds of detection.
2. The dynamic prioritisation of transitions between Seek Food, Seek Water, and Explore was based on urgency calculations.

Challenges and Observations

1. Resource Overlap: Areas with overlapping resource zones (like food and water near predator zones) presented problems for accurate detection and prioritisation.
2. Obtaining resources became less efficient in larger environments with multiple obstacles, not because they reduced the approach pathways but because they increased the coordination demands on the individual—navigating was more complex.
3. Health Decay Rates: Even faster health deterioration during encounters with predators puts additional stress on obtaining resources, which still requires more efficient recovery mechanisms to keep us alive.

Significance of Findings

The hypothesis that a dynamic state machine framework, combined with multi-sensory inputs, could efficiently manage a robot's survival needs was validated. The project succeeded in using a brachiating robot to balance tasks related to resource acquisition and predator avoidance. We highlight the successful outcome of this project because, unlike most robotics experiments, it has precise applications: managing mobile robots in unstructured environments, which might include real-world situations like disaster recovery or autonomous exploration in hazardous environments.

Discussion and Conclusion

Discussion

The project’s results validate the effectiveness of a state machine framework for managing survival-based tasks in autonomous robots. The robot showed real autonomous character in a simulated dynamic environment by dynamically prioritising its survival needs

Conclusion

The challenges of balancing resource acquisition and predator avoidance can be successfully addressed with integrated multi-sensory inputs and adaptive state-based behaviour. This might lead to the development of simpler, more self-sufficient robots.

Limitations:

1. Accuracy dropped when resource and threat zones were blended and hard to distinguish.
2. Manual tuning of colour detection threshold was intensive and might have constrained our potential for large-scale implementation.

Future Work:

1. Incorporate machine learning algorithms, such as reinforcement learning, to sharpen decision-making and adaptability
2. Broaden the ecology of the organisms to include other aspects vital for survival, like conserving energy or dealing with environmental hazards.
3. Create collaborative mechanisms for multiple robots to enhance resource efficiency and improve threat mitigation.

Bibliography and References

1. *Spier, E., & McFarland, D. (1997). Basic Cycles, Utility, and Opportunism in Self-Sufficient Robots. Robotics and Autonomous Systems, 20, 179–190.*
2. *Matarić, M. J., & Michaud, F. (2008). Behaviour-Based Systems. In B. Siciliano & O. Khatib (Eds.), Springer Handbook of Robotics (pp. 891–909). Springer.*
3. *Ghzouli, R., Berger, T., Johnsen, E. B., & Wasowski, A. (2022). Behaviour Trees and State Machines in Robotics Applications. arXiv preprint arXiv:2208.04211.*
4. *Floreano, D., & Keller, L. (2010). Evolution of Adaptive Behaviour in Robots by Means of Darwinian Selection. PLoS Biology, 8(1), e1000292.*
5. *Iovino, M., Förster, J., Falco, P., Chung, J. J., Siegwart, R., & Smith, C. (2024). Comparison between Behavior Trees and Finite State Machines. arXiv preprint arXiv:2405.16137.*

Appendix

Robot Controller Code

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <webots/camera.h>
```

```
#include <webots/motor.h>
#include <webots/robot.h>
#include <webots/receiver.h>

#define TIME_STEP 32
#define DEBUG_MODE 1
#define DEFAULT_SPEED 6

// State machine states
typedef enum {
    STATE_EXPLORE,
    STATE_SEEK_WATER,
    STATE_SEEK_FOOD,
    STATE_AVOID_PREDATOR,
    STATE_DRINK,
    STATE_EAT
} RobotState;

// Resource location
typedef struct {
    double x;      // x position in the image
    double y;      // y position in the image
    double size;   // size of the detected area
    int found;     // whether resource is currently visible
    double last_seen_time; // when resource was last detected
} ResourceLocation;

// Device tags
static WbDeviceTag left_motor, right_motor;
static WbDeviceTag camera, camera_1;  // camera is overhead, camera_1 is
forward-facing
static WbDeviceTag receiver;

// Resource variables
static double hunger = 100;
static double thirst = 100;
static double health = 100;
static const double HUNGER_DECAY = 1.0;
static const double THIRST_DECAY = 1.0;
static const double HEALTH_DECAY = 10.0;
```

```
// Global resource locations
static ResourceLocation water_location = {0, 0, 0, 0, 0};
static ResourceLocation food_location = {0, 0, 0, 0, 0};

// State machine variables
static RobotState current_state = STATE_EXPLORE;
static int state_timer = 0;
static int search_counter = 0;

// Random movement variables
static int random_move_timer = 0;
static int random_direction = 1;  // 1 for right, -1 for left
static const int RANDOM_MOVE_INTERVAL = 100;  // Change direction every 100
steps

// Survival time tracking
static double survival_time = 0.0;

static void initialize() {
    wb_robot_init();

    // Initialize random seed
    srand(time(NULL));

    // Initialize motors
    left_motor = wb_robot_get_device("left wheel motor");
    right_motor = wb_robot_get_device("right wheel motor");
    wb_motor_set_position(left_motor, INFINITY);
    wb_motor_set_position(right_motor, INFINITY);

    // Initialize cameras
    camera = wb_robot_get_device("camera");      // Overhead camera
    camera_1 = wb_robot_get_device("camera(1)"); // Forward-facing camera
    wb_camera_enable(camera, TIME_STEP);
    wb_camera_enable(camera_1, TIME_STEP);

    // Initialize receiver
    receiver = wb_robot_get_device("receiver");
    wb_receiver_enable(receiver, TIME_STEP);

    if (DEBUG_MODE) {
```

```
        printf("=== Robot Starting ===\n");
        printf("Initial Status: H:%.2f T:%.2f F:%.2f\n\n", health, thirst,
hunger);
    }
}

static void move(double l, double r) {
    wb_motor_set_velocity(left_motor, l);
    wb_motor_set_velocity(right_motor, r);
}

// Check for radio signals from predator
static int check_radio_signals() {
    while (wb_receiver_get_queue_length(receiver) > 0) {
        wb_receiver_next_packet(receiver);
        if (DEBUG_MODE) {
            printf("Received radio signal from predator!\n");
        }
        return 1;
    }
    return 0;
}

// Predator detection with direction using forward camera
static int detect_predator(double *red_percentage, int *predator_side) {
    if (check_radio_signals()) {
        *red_percentage = 1.0;
        *predator_side = 0;
        return 1;
    }

    const unsigned char *image = wb_camera_get_image(camera_1);
    int width = wb_camera_get_width(camera_1);
    int height = wb_camera_get_height(camera_1);

    int red_pixels_left = 0;
    int red_pixels_right = 0;
    int total_pixels = width * height;

    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
```

```
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
            int b = wb_camera_image_get_blue(image, width, x, y);
            if (r > 100 && r > (g * 1.2) && r > (b * 1.2)) {
                if (x < width/2) red_pixels_left++;
                else red_pixels_right++;
            }
        }
    }

    *red_percentage = (double)(red_pixels_left + red_pixels_right) /
total_pixels;
    *predator_side = (red_pixels_left > red_pixels_right) ? -1 : 1;

    if (DEBUG_MODE && *red_percentage > 0.001) {
        printf("Predator detected! Red percentage: %.3f%%, Side: %d\n",
               *red_percentage * 100, *predator_side);
    }

    return (*red_percentage > 0.001);
}

// Scan environment with overhead camera
static void scan_environment() {
    const unsigned char *image = wb_camera_get_image(camera);
    int width = wb_camera_get_width(camera);
    int height = wb_camera_get_height(camera);

    // Reset locations
    water_location.found = 0;
    food_location.found = 0;

    // Variables for centroid calculation
    double water_total_x = 0, water_total_y = 0, water_pixels = 0;
    double food_total_x = 0, food_total_y = 0, food_pixels = 0;

    // Scan entire image from overhead camera
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
```

```
            int b = wb_camera_image_get_blue(image, width, x, y);

            // Detect water (cyan)
            if (b > 180 && g > 100 && r < 100 && b > g) {
                water_total_x += x;
                water_total_y += y;
                water_pixels++;
            }

            // Detect food (dark green)
            if (g > 140 && g > r * 1.2 && g > b * 1.2) {
                food_total_x += x;
                food_total_y += y;
                food_pixels++;
            }
        }
    }

    // Update water location if found
    if (water_pixels > 10) {
        water_location.x = water_total_x / water_pixels;
        water_location.y = water_total_y / water_pixels;
        water_location.size = water_pixels / (width * height);
        water_location.found = 1;
        water_location.last_seen_time = wb_robot_get_time();

        if (DEBUG_MODE) {
            printf("Water detected at: (%.1f, %.1f), size: %.3f\n",
                   water_location.x, water_location.y,
water_location.size);
        }
    }

    // Update food location if found
    if (food_pixels > 10) {
        food_location.x = food_total_x / food_pixels;
        food_location.y = food_total_y / food_pixels;
        food_location.size = food_pixels / (width * height);
        food_location.found = 1;
        food_location.last_seen_time = wb_robot_get_time();
```

```
        if (DEBUG_MODE) {
            printf("Food detected at: (%.1f, %.1f), size: %.3f\n",
                   food_location.x, food_location.y, food_location.size);
        }
    }
}
// Function to check if resource is centered in forward camera
static int is_resource_centered(int is_water) {
    const unsigned char *image = wb_camera_get_image(camera_1);
    int width = wb_camera_get_width(camera_1);
    int height = wb_camera_get_height(camera_1);
    int center_x = width / 2;
    int resource_x = 0;
    int resource_pixels = 0;

    // Scan bottom third of image for resources
    for (int y = height * 2/3; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
            int b = wb_camera_image_get_blue(image, width, x, y);

            int is_target = 0;
            if (is_water && b > 180 && g > 100 && r < 100 && b > g) {
                is_target = 1;
            } else if (!is_water && g > 140 && g > r * 1.2 && g > b * 1.2)
{
                is_target = 1;
            }

            if (is_target) {
                resource_x += x;
                resource_pixels++;
            }
        }
    }

    if (resource_pixels > 10) {
        double avg_x = resource_x / resource_pixels;
        double offset = fabs(avg_x - center_x);
```

```
        return offset < (width * 0.1); // Resource is centered if within
10% of center
    }

    return 0;
}

// Enhanced priority system with weighted urgency
static RobotState determine_state() {
    // Dynamic weighting based on health
    double health_factor = (health < 50) ? 1.5 : 1.0;
    double thirst_urgency = (100 - thirst) * 1.2 * health_factor;
    double hunger_urgency = (100 - hunger) * health_factor;

    double red_percentage;
    int predator_side;

    // Check for predator first
    if (check_radio_signals() || detect_predator(&red_percentage,
&predator_side)) {
        if (DEBUG_MODE) printf("DANGER! Predator detected - switching to
avoidance!\n");
        return STATE_AVOID_PREDATOR;
    }

    // Regular state determination
    if (health < 70) return STATE_AVOID_PREDATOR;
    if (thirst < 30) return STATE_SEEK_WATER;
    if (hunger < 30) return STATE_SEEK_FOOD;

    if (thirst_urgency > hunger_urgency && thirst < 60) return
STATE_SEEK_WATER;
    if (hunger_urgency > thirst_urgency && hunger < 60) return
STATE_SEEK_FOOD;

    return STATE_EXPLORE;
}

// Modified update_resources function with exact colour matching
static void update_resources() {
    double red_percentage;
```

```
    int predator_side;

    // Update scan of environment
    scan_environment();

    // Basic decay
    hunger -= HUNGER_DECAY / TIME_STEP;
    thirst -= THIRST_DECAY / TIME_STEP;

    // Check forward camera for resources
    const unsigned char *image = wb_camera_get_image(camera_1);
    int width = wb_camera_get_width(camera_1);
    int height = wb_camera_get_height(camera_1);
    int water_pixels = 0;
    int food_pixels = 0;

    // Check camera for resources
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            int r = wb_camera_image_get_red(image, width, x, y);
            int g = wb_camera_image_get_green(image, width, x, y);
            int b = wb_camera_image_get_blue(image, width, x, y);

            // Water detection (cyan/blue)
            if (b > 200 && g > 180 && r < 100) {
                water_pixels++;
            }

            //food detection (dark green)
            if (g > 140 && g > (r * 1.5) && g > (b * 1.5)) {
                food_pixels++;
            }
        }
    }

    // Resource recovery
    if (water_pixels > 100) {
        thirst = 100.0;
        if (DEBUG_MODE) printf("Water detected! Thirst reset to: 100.0\n");
    }
```

```
    if (food_pixels > 100) {
        hunger = 100.0;
        if (DEBUG_MODE) printf("Food detected! Hunger reset to: 100.0\n");
    }

    // Predator damage
    if (detect_predator(&red_percentage, &predator_side)) {
        double damage = HEALTH_DECAY * (red_percentage * 2) / TIME_STEP;
        health -= damage;
        if (DEBUG_MODE && damage > 0.1) {
            printf("!!! HEALTH WARNING !!! Predator damage: -%.2f (Health:
%.2f)\n", damage, health);
        }
    }

    // Clamp values
    if (hunger > 100) hunger = 100;
    if (thirst > 100) thirst = 100;
    if (health > 100) health = 100;

    if (DEBUG_MODE) {
        printf("Status: H:%.2f T:%.2f F:%.2f State: %d\n", health, thirst,
hunger, current_state);
    }
}
// Enhanced behaviour execution with dual camera system and random
exploration
static void execute_behaviour() {
    // First scan environment with overhead camera
    scan_environment();

    // Update state
    RobotState new_state = determine_state();
    if (new_state != current_state) {
        state_timer = 0;
        current_state = new_state;
        if (DEBUG_MODE) printf("State changed to: %d\n", current_state);
    }

    // Execute behaviour based on state
    switch (current_state) {
```

```
        case STATE_AVOID_PREDATOR:
            {
                double red_percentage;
                int predator_side;
                detect_predator(&red_percentage, &predator_side);

                if (check_radio_signals() || red_percentage > 0.05) {
                    move(-DEFAULT_SPEED, -DEFAULT_SPEED);  // Full retreat
                    if (DEBUG_MODE) printf("DANGER - RETREATING!\n");
                } else {
                    // Turn away from last known predator position
                    move(predator_side * -DEFAULT_SPEED / 2, predator_side
* DEFAULT_SPEED / 2);
                }
                state_timer = 0;
            }
            break;

        case STATE_SEEK_WATER:
            if (water_location.found) {
                // Calculate turn direction based on water location
                double center_x = wb_camera_get_width(camera) / 2;
                if (is_resource_centered(1)) {
                    current_state = STATE_DRINK;
                } else if (water_location.x < center_x - 10) {
                    move(DEFAULT_SPEED - 2, DEFAULT_SPEED + 2); // Turn
left
                } else if (water_location.x > center_x + 10) {
                    move(DEFAULT_SPEED + 2, DEFAULT_SPEED - 2); // Turn
right
                } else {
                    move(DEFAULT_SPEED, DEFAULT_SPEED); // Move forward
                }
            } else {
                // Search pattern when water not visible
                if (search_counter++ % 20 < 10) {
                    move(DEFAULT_SPEED - 1, DEFAULT_SPEED + 1);
                } else {
                    move(DEFAULT_SPEED + 1, DEFAULT_SPEED - 1);
                }
            }
```

```
            break;

        case STATE_SEEK_FOOD:
            if (food_location.found) {
                // Calculate turn direction based on food location
                double center_x = wb_camera_get_width(camera) / 2;
                if (is_resource_centered(0)) {
                    current_state = STATE_EAT;
                } else if (food_location.x < center_x - 10) {
                    move(DEFAULT_SPEED - 2, DEFAULT_SPEED + 2); // Turn
left
                } else if (food_location.x > center_x + 10) {
                    move(DEFAULT_SPEED + 2, DEFAULT_SPEED - 2); // Turn
right
                } else {
                    move(DEFAULT_SPEED, DEFAULT_SPEED); // Move forward
                }
            } else {
                // Search pattern when food not visible
                if (search_counter++ % 20 < 10) {
                    move(DEFAULT_SPEED + 1, DEFAULT_SPEED - 1);
                } else {
                    move(DEFAULT_SPEED - 1, DEFAULT_SPEED + 1);
                }
            }
            break;

        case STATE_DRINK:
            if (thirst >= 100) {
                current_state = STATE_EXPLORE;
                if (DEBUG_MODE) printf("Finished drinking, thirst
satisfied.\n");
            } else if (!is_resource_centered(1)) {
                current_state = STATE_SEEK_WATER;
                if (DEBUG_MODE) printf("Lost water source, returning to
seek state.\n");
            } else {
                move(0, 0);  // Stop to drink
                if (DEBUG_MODE) printf("At water source. Drinking...\n");
            }
            break;
```

```
        case STATE_EAT:
            if (hunger >= 100) {
                current_state = STATE_EXPLORE;
                if (DEBUG_MODE) printf("Finished eating, hunger
satisfied.\n");
            } else if (!is_resource_centered(0)) {
                current_state = STATE_SEEK_FOOD;
                if (DEBUG_MODE) printf("Lost food source, returning to seek
state.\n");
            } else {
                move(0, 0);  // Stop to eat
                if (DEBUG_MODE) printf("At food source. Eating...\n");
            }
            break;

        case STATE_EXPLORE:
        default:
            {
                // Update random movement timer
                random_move_timer++;

                // Change random direction periodically or based on random
chance
                if (random_move_timer >= RANDOM_MOVE_INTERVAL || (rand() %
100 < 5)) {
                    random_move_timer = 0;
                    random_direction = (rand() % 2) * 2 - 1;  // Random -1
or 1

                    if (DEBUG_MODE) {
                        printf("Changing random direction: %d\n",
random_direction);
                    }
                }

                // Random movement patterns
                int movement_pattern = rand() % 4;

                switch (movement_pattern) {
                    case 0:  // Move forward
```

```
                        move(DEFAULT_SPEED, DEFAULT_SPEED);
                        break;

                    case 1:  // Turn slightly
                        move(DEFAULT_SPEED + random_direction,
DEFAULT_SPEED - random_direction);
                        break;

                    case 2:  // Turn more sharply
                        move(DEFAULT_SPEED + (2 * random_direction),
DEFAULT_SPEED - (2 * random_direction));
                        break;

                    case 3:  // Random spin with forward motion
                        move(DEFAULT_SPEED + (3 * random_direction),
DEFAULT_SPEED - (3 * random_direction));
                        break;
                }
            }
            break;
    }

    state_timer++;
}

int main() {
    initialize();

    while (wb_robot_step(TIME_STEP) != -1 && hunger > 0 && thirst > 0 &&
health > 0) {
        survival_time += (double)TIME_STEP / 1000.0;
        update_resources();
        execute_behaviour();
    }

    move(0, 0);
    if (DEBUG_MODE) {
        printf("=== Simulation Ended ===\n");
        printf("Total Survival Time: %.2f seconds\n", survival_time);
        printf("Final Status: H:%.2f T:%.2f F:%.2f\n", health, thirst,
hunger);
```

```
    }

    wb_robot_cleanup();
    return 0;
}
```