Politechnika Wrocławska

Lab04 - Sterowanie nadrzedne

Roboty Mobilne 1

Michał Markuzel, 275417

19.05.2025

Spis treści

1 Wprowadzenie

Celem laboratorium było zaimplementowanie sterownika nadrzędnego dla floty robotów mobilnych, który zarządza ruchem robotów w symulowanym środowisku magazynowym. System został zaprojektowany do obsługi 16 robotów TurtleBot3, które wykonują zadania transportowe między różnymi stanowiskami w magazynie.

2 Środowisko symulacyjne

2.1 Konfiguracja środowiska

Środowisko zostało skonfigurowane zgodnie z instrukcją:

```
mkdir -p robot_ws/src
cd robot_ws/src
git clone https://github.com/arshadlab/turtlebot3_multi_robot.git -b foxy
cd ..
rosdep install --from-paths src -r -y
colcon build --symlink-install
source ./install/setup.bash
```

2.2 Uruchomienie symulacji

Symulacja jest uruchamiana za pomocą polecenia:

```
ros2 launch turtlebot3_multi_robot gazebo_multi_world.launch.py \
enable_drive:=False
```

3 Implementacja

3.1 Architektura systemu

System składa się z następujących głównych komponentów:

- **Zarządzanie kolejką robotów** - implementuje logikę kolejkowania robotów w bazie
- **System przydzielania zadań** - obsługuje przyjmowanie i dystrybucję zadań
- **Kontrola ruchu robotów** - zarządza przemieszczaniem się robotów
- **Komunikacja ROS2** - obsługuje wymianę danych między komponentami

3.2 Kluczowe struktury danych

3.2.1 Struktura zadania

```
struct Task {
    int pickup_point;    // Numer punktu odbioru (0-9)
    int delivery_point;  // Numer punktu dostawy (0-9)
    Point2D pickup;      // Współrzędne punktu odbioru
    Point2D delivery;    // Współrzędne punktu dostawy
};
```

3.2.2 Stan robota

```
struct RobotState {
    geometry_msgs::msg::Pose current_pose;  // Aktualna poza
    Point2D current_position;               // Pozycja 2D
    bool is_busy;                          // Status zajętości
    Task current_task;                     // Aktualne zadanie
    int movement_phase;                    // Faza ruchu
    double last_heading_error;             // Dla kontrolera PID
```

```
    double heading_error_integral;         // Dla kontrolera PID
    enum class State {
        WAITING,            // Oczekiwanie w kolejce
        MOVING_TO_PICKUP,   // Ruch do punktu odbioru
        MOVING_TO_DELIVERY, // Ruch do punktu dostawy
        RETURNING_TO_QUEUE  // Powrót do kolejki
    } state;
};
```

4 Implementacja ruchu robotów

4.1 System kontroli ruchu

Kontrola ruchu robotów została zaimplementowana z wykorzystaniem kontrolera PID dla sterowania kątem oraz prostego sterowania prędkością liniową. System wykorzystuje metrykę miejską do planowania tras, co oznacza, że roboty poruszają się głównie wzdłuż osi X i Y.

4.2 Funkcja move_to_pickup

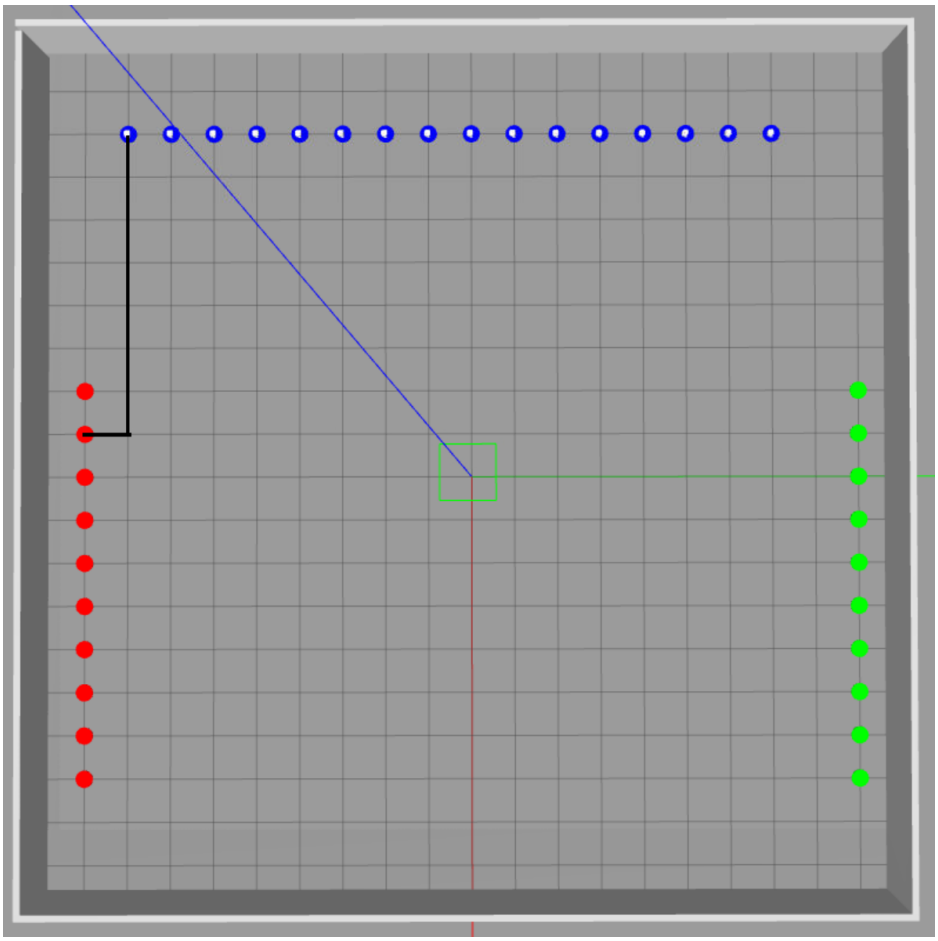
Funkcja realizuje transport robota do punktu odbioru w czterech fazach:

```
bool FleetController::move_to_pickup(int robot_id) {
    auto& robot = robots[robot_id];
    static Point2D intermediate_point;

    switch (robot.movement_phase) {
        case 0:  // Obrót w kierunku osi X
            if (rotate_robot(robot_id, 0.0)) {
                robot.movement_phase = 1;
                intermediate_point = {
                    robot.current_task.pickup.x,
                    robot.current_position.y
                };
            }
            break;
        case 1:  // Ruch wzdłuż osi X
            if (move_straight(robot_id, intermediate_point)) {
                robot.movement_phase = 2;
            }
            break;
        case 2:  // Obrót w kierunku osi Y
            if (rotate_robot(robot_id, -M_PI/2)) {
                robot.movement_phase = 3;
            }
            break;
        case 3:  // Ruch wzdłuż osi Y
            if (move_straight(robot_id,
                robot.current_task.pickup)) {
                robot.movement_phase = 0;
                return true;
            }
            break;
    }
    return false;
}
```

Funkcja implementuje następującą strategię ruchu:

1. Obrót robota w kierunku osi X (0 radianów)
2. Przemieszczenie do odpowiedniej współrzędnej X
3. Obrót w kierunku osi Y (-$\pi/2$ radianów)
4. Przemieszczenie do punktu docelowego wzdłuż osi Y



Rysunek 1: Przykładowa trasa do punktu pick-up nr 3

4.3 Kontroler PID

Dla zapewnienia płynnego ruchu i precyzyjnego sterowania, zaimplementowano kontroler PID:

```
// Parametry PID
double Kp = 0.5;   // Wzmocnienie proporcjonalne
double Ki = 0.01;  // Wzmocnienie całkujące
double Kd = 0.1;   // Wzmocnienie różniczkujące

// Obliczenie sterowania
double angular_control = Kp * heading_error +
                         Ki * heading_error_integral +
                         Kd * heading_error_derivative;
```

4.4 Funkcja move_to_delivery

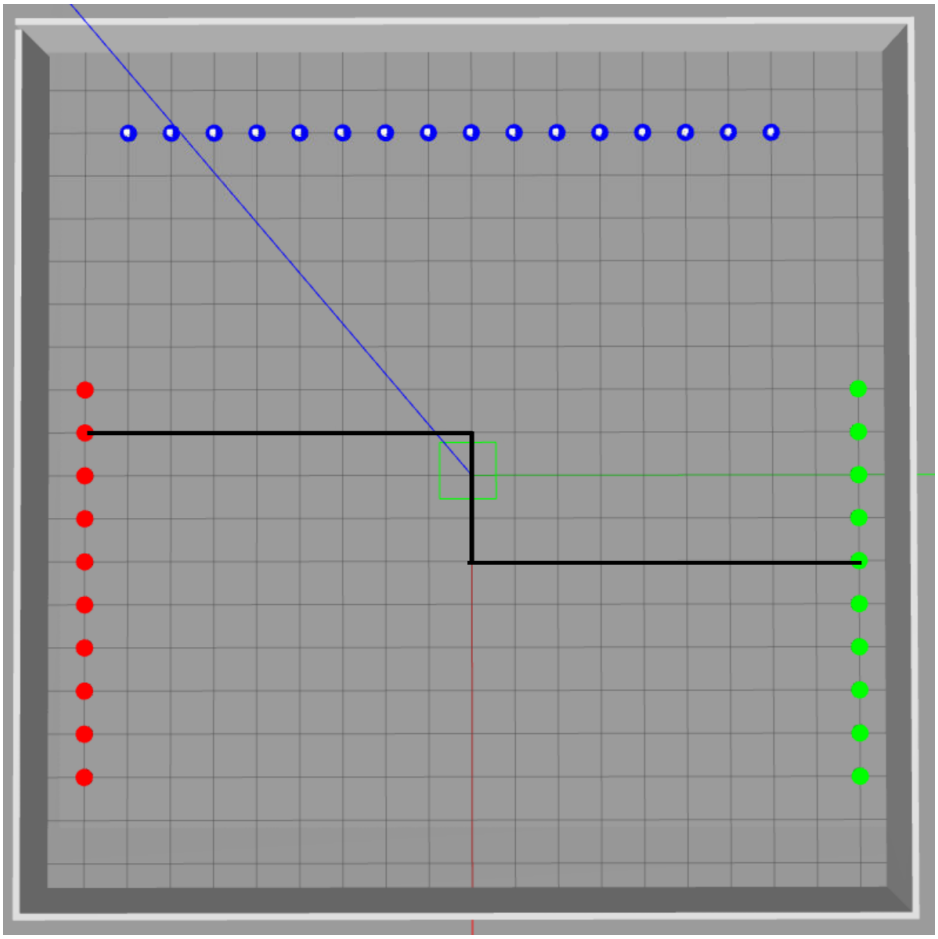
Funkcja odpowiedzialna za transport robota do punktu dostawy realizuje trasę w sześciu fazach:

```
bool FleetController::move_to_delivery(int robot_id) {
    auto& robot = robots[robot_id];
    static Point2D intermediate_point;

    switch (robot.movement_phase) {
        case 0:  // Obrót do pozycji pionowej
            if (rotate_robot(robot_id, M_PI/2)) {
                robot.movement_phase = 1;
                intermediate_point = {
                    robot.current_task.pickup.x,
                    0  // Przejście do Y=0
                };
            }
            break;
        case 1:  // Ruch do Y=0 (główna ścieżka)
            if (move_straight(robot_id, intermediate_point)) {
                robot.movement_phase = 2;
            }
            break;
        case 2:  // Obrót w kierunku X celu
            {
                double target_angle =
                    (robot.current_task.delivery.x >
                     robot.current_position.x) ? 0.0 : M_PI;
                if (rotate_robot(robot_id, target_angle)) {
                    robot.movement_phase = 3;
                    intermediate_point = {
                        robot.current_task.delivery.x,
                        robot.current_position.y
                    };
                }
            }
            break;
        case 3:  // Ruch do X celu
            if (move_straight(robot_id, intermediate_point)) {
                robot.movement_phase = 4;
            }
            break;
        case 4:  // Obrót w kierunku punktu docelowego
            if (rotate_robot(robot_id, M_PI/2)) {
                robot.movement_phase = 5;
            }
            break;
        case 5:  // Końcowy ruch do celu
            if (move_straight(robot_id,
                robot.current_task.delivery)) {
                robot.movement_phase = 0;
                return true;
            }
            break;
    }
    return false;
}
```

Strategia ruchu obejmuje:

1. Obrót w stronę punktów docelowych ($\pi/2$ radianów)
2. Przemieszczenie do środka (Y=0)
3. Obrót w kierunku docelowego X (0 lub π radianów)
4. Ruch do docelowej współrzędnej X
5. Obrót w kierunku punktu docelowego ($\pi/2$ radianów)
6. Końcowe przemieszczenie do punktu dostawy



Rysunek 2: Przykładowa trasa do punktu docelowego nr 5

4.5 Funkcja return_to_queue

Funkcja implementująca powrót robota do kolejki:

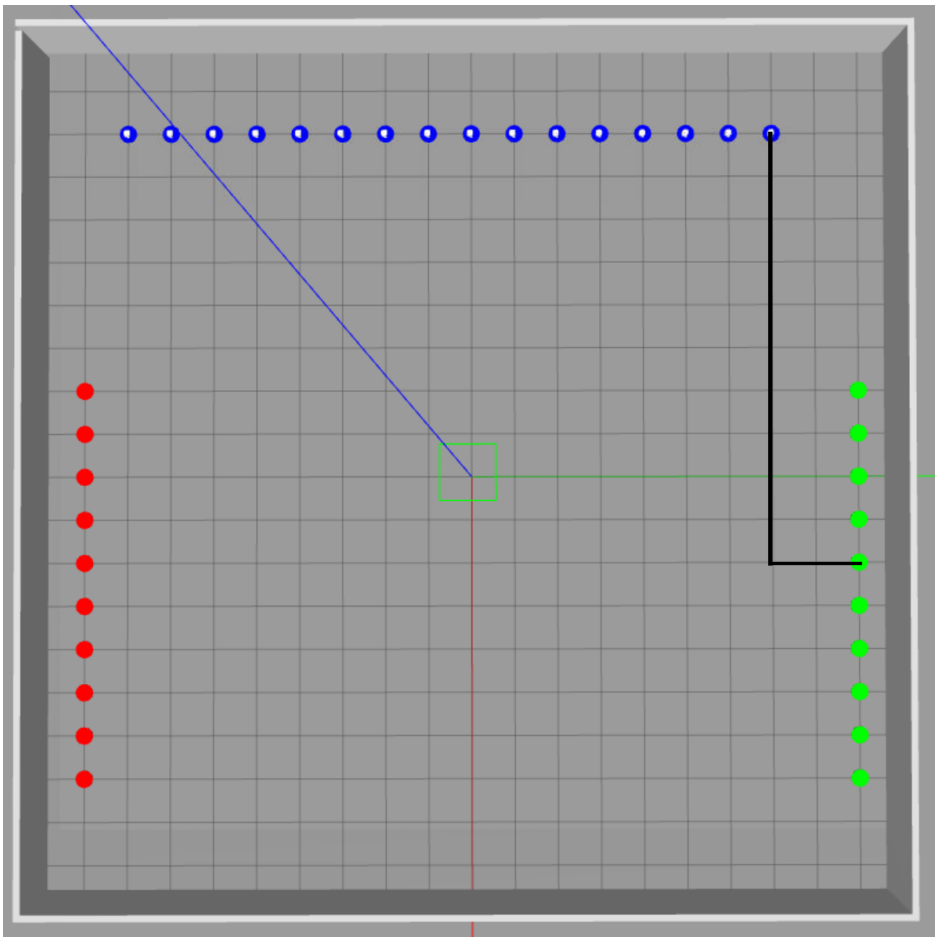
```
bool FleetController::return_to_queue(int robot_id) {
    auto& robot = robots[robot_id];
    static Point2D intermediate_point;
    static int target_queue_position = -1;

    switch (robot.movement_phase) {
        case 0:  // Obrót w dół
            if (rotate_robot(robot_id, -M_PI/2)) {
                robot.movement_phase = 1;
                intermediate_point = {
```

```
                    robot.current_task.delivery.x,
                    robot_bases[15].y  // Pozycja końca kolejki
                };
            }
            break;
        case 1:  // Ruch do Y kolejki
            if (move_straight(robot_id, intermediate_point)) {
                robot.movement_phase = 2;
            }
            break;
        case 2:  // Obrót w prawo
            if (rotate_robot(robot_id, M_PI)) {
                robot.movement_phase = 3;
            }
            break;
        case 3:  // Ruch do X kolejki
            if (move_straight(robot_id, robot_bases[15])) {
                robot.movement_phase = 4;
            }
            break;
        case 4:  // Obrót do orientacji kolejki
            if (rotate_robot(robot_id, -M_PI/2)) {
                robot.movement_phase = 5;
                target_queue_position =
                    find_first_empty_queue_position();
                if (target_queue_position != -1) {
                    intermediate_point =
                        robot_bases[target_queue_position];
                }
            }
            break;
        case 5:  // Zajęcie pozycji w kolejce
            if (target_queue_position != -1 &&
                move_straight(robot_id, intermediate_point)) {
                queue_state[target_queue_position] = robot_id;
                robot.movement_phase = 0;
                return true;
            }
            break;
    }
    return false;
}
```

Proces powrotu do kolejki obejmuje:

1. Obrót w kierunku punktów pick-up (-π/2 radianów)
2. Ruch do poziomu Y kolejki
3. Obrót w kierunku kolejki (π radianów)
4. Ruch do pozycji X kolejki
5. Obrót do orientacji kolejki (-π/2 radianów)
6. Zajęcie ostatniego wolnego miejsca



Rysunek 3: Przykładowa trasa powrotu do kolejki

5 System kolejkowania zadań

5.1 Przyjmowanie nowych zadań

System przyjmuje zadania poprzez topic ROS2 lub umożliwia losowy wybór punktów przy podaniu zerowych argumentów:

```
void FleetController::task_callback(
    const std_msgs::msg::Int32MultiArray::SharedPtr msg) {
    if (msg->data.size() >= 2) {
        Task new_task;
        std::random_device rd;
```

```
        std::mt19937 gen(rd());
        std::uniform_int_distribution<> dist(0, 9);

        // Obsługa punktu odbioru
        if (msg->data[0] == 0) {
            new_task.pickup_point = dist(gen);
        } else {
            new_task.pickup_point = msg->data[0] - 1;
        }

        // Obsługa punktu dostawy
        if (msg->data[1] == 0) {
            do {
                new_task.delivery_point = dist(gen);
            } while (new_task.delivery_point ==
                    new_task.pickup_point);
        } else {
            new_task.delivery_point = msg->data[1] - 1;
        }

        // Dodanie zadania do kolejki
        task_queue.push(new_task);
    }
}
```

5.2 Zarządzanie kolejką robotów

System implementuje automatyczne przesuwanie robotów w kolejce:

```
void FleetController::reorganize_queue() {
    bool changes_made = false;
    for (int i = 0; i < NUM_ROBOTS - 1; i++) {
        if (queue_state[i] == -1) {
            for (int j = i + 1; j < NUM_ROBOTS; j++) {
                if (queue_state[j] != -1) {
                    int robot_id = queue_state[j];
                    if (!robots[robot_id].is_busy &&
                        robots[robot_id].state ==
                        RobotState::State::WAITING) {
                        queue_state[i] = robot_id;
                        queue_state[j] = -1;
                        move_straight(robot_id,
                            robot_bases[i]);
                        changes_made = true;
                        break;
                    }
                }
            }
        }
    }
}
```

6 Wnioski

Zrealizowany system sterownika nadrzędnego skutecznie zarządza flotą robotów, implementując:

- Efektywne kolejkowanie zadań i robotów
- Bezkolizyjny ruch w oparciu o metrykę miejską
- Automatyczną reorganizację kolejki
- Losowy wybór punktów odbioru/dostawy

Główne wyzwania implementacyjne obejmowały:

- Synchronizację ruchu wielu robotów
- Precyzyjne sterowanie z wykorzystaniem PID
- Efektywne zarządzanie kolejką robotów

7 Bibliografia

1. Dokumentacja ROS2: `https://docs.ros.org/en/foxy/`
2. TurtleBot3 Multi Robot: `https://github.com/arshadlab/turtlebot3_multi_robot`
3. ROS2 Navigation: `https://navigation.ros.org/`