ДОДАТОК А

Апробація результатів роботи

TECHNICAL SCIENCES

ANALYSIS OF THE STRUCTURES OF MOBILE PLATFORMS FOR PROMOTER ROBOTS

Bielik Maksym Serhiiovych,
student
Kharkiv National University of Radio Electronics
Sotnik Svitlana Viktorivna,
PhD, associate professor of CITAR department
Kharkiv, Ukraine

Introductions

The modern development of robotics and automation is leading to the active introduction of robotic systems into various areas of human activity, including service, marketing, and promotion [1-3]. Robot promoters (RP) are becoming increasingly popular in shopping centers, exhibitions, presentations, and other public events, where they perform the functions of informing, advising, and attracting the attention of visitors.

A key component of any mobile robot is its mobile platform, which ensures movement in space, maneuverability, and stability of the system. For RP, the following characteristics of the mobile platform are particularly important: reliability of movement in crowded places, ability to position accurately, safety of interaction with people, and aesthetic appearance [4-6].

Designing a mobile platform for a promotional robot requires a comprehensive approach that takes into account specific requirements for operation in public places, the need to ensure autonomous operation, and integration with navigation systems, sensor equipment, and communication modules. Particular attention should be paid to safety, ergonomics, and adaptability to different operating conditions [7-9].

The analysis of existing designs of mobile platforms for promotional robots is

relevant for determining optimal technical solutions that ensure effective functioning in public spaces, taking into account modern requirements for safety, maneuverability, and interaction with people.

Aim

Analyze existing designs of mobile platforms for promotional robots in order to identify their advantages and disadvantages and determine the optimal technical solutions for further design.

Materials and methods

The research methodology includes a systematic analysis of existing mobile platform designs, a comparative analysis of the technical characteristics of different types of drives and control systems, as well as mathematical modeling methods for calculating the motion and stability parameters of the platform. To select the optimal materials and components, a multi-criteria analysis method was used, taking into account factors such as weight, strength, cost, and performance characteristics.

The choice of drive type significantly affects the reliability, energy efficiency, and overall operating cost of a promotional robot, so it is advisable to systematize the main characteristics of the most common types of motors in a comparative table to determine the optimal solution (Table 1).

Table 1

Comparative table of drive types for RP mobile platforms

Parameter	DC motors	Brushless motors (BLDC)
Effectiveness	Up to 80 %	85-95 %
How it works	Use of commutation brushes	Brushless electronic commutation
Service life	Limited by brush wear	Greater due to the absence of brushes
Complexity of control	Simple (linear dependence of speed on voltage)	More complex (requires an electronic controller)
Cost	Low	Higher
Starting point	Good	Excellent
Speed range	Wide	Wide
Noise level	Average	Low
Maintenance	Requires replacement of brushes	Minimum

To determine the optimal drive parameters, it is necessary to establish technical requirements for the designed mobile platform. This work considers a robot promoter with a load capacity of approximately 30 kg, which includes the weight of electronic equipment, control systems, additional modules, and payload. Therefore, the recommended characteristics for a platform with a load capacity of 30 kg are determined (Table 2).

Therefore, BLDC motors are the optimal choice for a promotional robot with a load capacity of 30 kg due to their balance between reliability and efficiency, increased battery life, minimal maintenance requirements, etc.

Table 2

Recommended specifications for a platform with a load capacity of 30 kg

Parameter	DC motors	Brushless motors (BLDC)
Torque	0,5-2 Nm	1-3 Nm
Power	100-300 W	150-400 W
Rotation speed	1500–3000 rpm	1000-4000 rpm
Battery life	Smaller	Greater

The study included a comparative analysis of control systems used in the design of mobile platforms for mobile robots, in particular promotional robots. The main focus was on the following types of control systems:

– proportional-integral-derivative (PID) control;

– microcontroller-based systems (e.g., Arduino, STM32);

– systems using FPGA or single-board computers (Raspberry Pi, Jetson Nano);

– intelligent control systems using artificial intelligence elements (neural networks, fuzzy logic).

The comparison was based on the following criteria: accuracy and stability of motion control; complexity of implementation; flexibility of adaptation to changes in the environment; energy consumption; cost and scalability.

The results of the analysis showed that for a promotional robot, the optimal solution is to use a microcontroller-based control system with PID regulation, which provides sufficient accuracy at a relatively low cost and ease of implementation. In cases where the platform must autonomously navigate a dynamic environment (e.g.,

in a shopping mall), it is advisable to use intelligent control systems with computer vision and decision-making algorithms.

To calculate the motion parameters and ensure platform stability, mathematical modeling methods based on classical mechanics and automatic control theory were used. The motion model is based on kinematic and dynamic modeling of the platform as a wheeled system.

Results and discussion

Based on the described methodology, a comprehensive analysis and calculation of the mobile platform parameters was performed, the results of which are presented below.

The calculations of the drive parameters showed that to ensure reliable operation of the mobile platform robot-promoter with a load capacity of 30 kg, the minimum motor power required is 24,57 W per wheel. The calculation was based on the following parameters:

– total weight of the system – 50 kg (30 kg load + 20 kg own weight);

– maximum speed: 1,2 m/s (safe for indoor use);

– calculated acceleration: 0,5 m/s² (comfortable movement);

– transmission efficiency – 0,85.

The traction force required to overcome resistance to movement is 34,81 N, which includes rolling resistance (9,81 N) and acceleration inertia (25 N). To ensure reliable operation in various operating conditions, it is recommended to increase the calculated power by 3-5 times, which gives an optimal power of 100-125 W per wheel.

The energy analysis of the system included energy consumption calculations:

– peak consumption: ~720 W (30 A at 24 V);

– average consumption in operating mode: ~200-300 W;

– autonomy with a LiFePO4 24V 20Ah battery: 1,5-2 hours;

– the selected type of LiFePO4 batteries provides over 2000 charge-discharge cycles, which significantly reduces operating costs compared to traditional lead-acid batteries.

The results obtained are of practical importance for the development of commercial promotional robots:

– the methodology for calculating parameters can be applied to platforms with different load capacities;

– the proposed architecture is scalable for different applications;

– the standardized Modbus interface simplifies integration with existing systems.

The research results show that the proposed mobile platform design provides an optimal balance between performance, reliability, and cost-effectiveness for use in promotional robots.

Conclusions

As a result of a comprehensive analysis of the designs of mobile platforms for promotional robots, the most effective technical solutions were identified. A comparative analysis of drive types confirmed the advantages of BLDC motors for platforms with a load capacity of 30 kg due to their high efficiency (85-95 %), long service life, and minimal maintenance requirements. Calculations showed the need for 24,57 W of power per wheel with a recommended reserve of 100-125 W to ensure reliable operation in various operating conditions.

Analysis of control systems revealed the optimality of a combined architecture using Raspberry Pi for high-level computing tasks and STM32 for real-time control. This structure provides the necessary performance for implementing SLAM algorithms while maintaining the reliability of critical safety functions. Mathematical modeling confirmed the effectiveness of the proposed motion parameters and energy characteristics of the system.

Energy analysis showed that the use of LiFePO4 batteries provides 1,5-2 hours of autonomous operation with an average consumption of 200-300 W and over 2000 charge-discharge cycles. The developed Modbus interface allows for standardized integration with other robot modules and provides reliable communication at distances of up to 1200 m.

The results obtained are of practical importance for the development of

commercial promotional robots and can be adapted for platforms with different load capacities. The proposed methodology for calculating parameters and selecting components provides an optimal balance between performance, reliability, and cost-effectiveness, which is critical for the successful implementation of robotic systems in the service and marketing sectors.

REFERENCES:

1. Andreiev, A.S. Analysis of robotics platforms for educational and research purposes / A.S. Andreiev, et al. // Комп'ютерні ігри та мультимедіа як інноваційний підхід до комунікації - 2024 / Матеріали IV Всеукраїнської науково-технічної конференції молодих вчених, аспірантів і студентів, Одеса, 26-27 вересня 2024 р., 2024, pp. 25-27.

2. Zarubin, I. Basic principles of building aerial robots / I. Zarubin, et al. // Manufacturing & Mechatronic Systems 2024: Proceedings of VIII st International Conference, Kharkiv, October 25-26, 2024, pp. 32-36.

3. Зарубін, І. С. Ефективність використання роботизованих систем у виробництві / І. С. Зарубін та ін. // Комп'ютерно-інтегрованих технологій, автоматизації та робототехніки 2024: матеріали І-ої Всеукраїнської конференції, Харків, 16-17 травня 2024 (CITAR-2024). – 2024. – С. 150-153.

4. Lashyn, Z. V. Automation capabilities of equipment with built-in robot for manufacture of microelectronics products / Z. V. Lashyn, et al. // Proceedings of the XVII International scientific and practical conference «Information technologies and automation – 2024», 2024. – pp. 283-286/

5. Baker, J.H. Some interesting features of semantic model in Robotic Science / J.H. Baker, V. et al. // International Journal of Engineering Trends and Technologythis link is disabled. – 2021. – 69(7). – P. 38-44.

6. Mohammad, A.S.Y. Neural networks as a tool for pattern recognition of fasteners / A.S.Y. Mohammad et al. // International Journal of Engineering Trends and Technologythis link is disabled. – 2021. – 69(10). – С. 151-160.

7. Lykho, T.A. Pattern recognition and computer vision technologies in

decision support systems of robotic systems / T.A. Lykho, et al. // Proceedings of the XVII International scientific and practical conference «Information technologies and automation – 2024», 2024. – pp. 645-648.

8. Yechevskyi, A. D. Research of orientation methods of autonomous mobile robots in industrial conditions / A. D. Yechevskyi, et al. // «Computer-integrated technologies, automation and robotics» CITAR-2025, 2025. – pp. 115-119.

9. Sotnik, S. V. Safe cobots in development of industrial robotics. Diss. Barca Academy Publishing / S.V. Sotnik, et al. // The 8th International scientific and practical conference "European scientific congress". – 2023. – C. 201-205.

ДОДАТОК Б

Програмний код основного контролера webots

```
# coding: utf-8

import math
import heapq
from typing import Optional, Tuple, List, Dict, Set
from controller import Robot
```

```
class PathPlannerAStar:
    """
    Реалізація алгоритму A* для пошуку маршруту по сітковій карті
    """
```

```
    class Node:
        """Внутрішній клас для представлення вузла в графі пошуку A*."""
        def __init__(self, position: Tuple[int, int], parent=None):
            self.position = position
            self.parent = parent
            self.g = 0  # Вартість шляху від початкового вузла до поточного
            self.h = 0  # Евристична вартість шляху від поточного вузла до цілі
            self.f = 0  # Загальна вартість (g + h)
```

```
        def __eq__(self, other):
            return self.position == other.position
```

```
        def __lt__(self, other):
            return self.f < other.f
```

```
        def __hash__(self):
            return hash(self.position)
```

```
    def __init__(self, controller):
        self.controller = controller
        self.grid_cell_size = 0.2  # Розмір комірки сітки з контролера
        self.obstacle_threshold = 0.6 # Ймовірність сканування комірки на наявність
перешкоди
```

```
    def plan_path(self, start_world: Tuple[float, float], goal_world: Tuple[float,
float]) -> Optional[List[Tuple[float, float]]]:
        start_grid = (int(start_world[0] / self.grid_cell_size), int(start_world[1] /
self.grid_cell_size))
        goal_grid = (int(goal_world[0] / self.grid_cell_size), int(goal_world[1] /
self.grid_cell_size))
```

```
        print(f"A* Planner: Запит на планування від {start_grid} до {goal_grid}")
```

```
        if self._is_cell_blocked(goal_grid):
            print("A* Planner: Ціль знаходиться всередині перешкоди")
            return None
```

```
        start_node = self.Node(start_grid)
        goal_node = self.Node(goal_grid)
```

```
        # Ініціалізація списків
        open_list: List[self.Node] = []
        closed_set: Set[Tuple[int, int]] = set()
```

```
        heapq.heappush(open_list, start_node)
```

```
        max_iterations = 2000
        iterations = 0
        while open_list and iterations < max_iterations:
            iterations += 1
            current_node = heapq.heappop(open_list)
            closed_set.add(current_node.position)

            if current_node == goal_node:
                print(f"A* Planner: Маршрут знайдено за {iterations} ітерацій!")
                return self._reconstruct_path(current_node)
```

```
            for dx, dy in [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1,
1), (-1, -1)]:
                neighbor_pos = (current_node.position[0] + dx,
current_node.position[1] + dy)

                if self._is_cell_blocked(neighbor_pos):
                    continue

                if neighbor_pos in closed_set:
                    continue

                neighbor_node = self.Node(neighbor_pos, current_node)
                neighbor_node.g = current_node.g + (1.414 if abs(dx) == 1 and abs(dy)
== 1 else 1.0)
                neighbor_node.h = math.sqrt((neighbor_node.position[0] -
goal_node.position[0])**2 +
                                            (neighbor_node.position[1] -
goal_node.position[1])**2)
                neighbor_node.f = neighbor_node.g + neighbor_node.h
```

```
                # Якщо сусідня комірка вже є у відкритому списку, але з більшою
вартістю, то пропускаємо
                if any(n for n in open_list if n == neighbor_node and n.g <=
neighbor_node.g):
                    continue
```

```
                heapq.heappush(open_list, neighbor_node)

        if iterations >= max_iterations:
             print("A* Planner: Перевищено ліміт ітерацій для пошуку маршруту.")
        else:
             print("A* Planner: Маршрут не знайдено.")
        return None
```

```
    def _is_cell_blocked(self, grid_pos: Tuple[int, int]) -> bool:
        if self.controller.obstacle_map.get(grid_pos, 0.0) > self.obstacle_threshold:
            return True

        buffer_radius = 1
        for dx in range(-buffer_radius, buffer_radius + 1):
            for dy in range(-buffer_radius, buffer_radius + 1):
                if dx == 0 and dy == 0:
                    continue
                check_pos = (grid_pos[0] + dx, grid_pos[1] + dy)
                if self.controller.obstacle_map.get(check_pos, 0.0) >
self.obstacle_threshold:
                    return True
```

```
        return False
```

```
    def _reconstruct_path(self, current_node: Node) -> List[Tuple[float, float]]:
        path_grid = []
        current = current_node
        while current is not None:
            path_grid.append(current.position)
            current = current.parent
        path_grid.reverse()
```

```
        path_world = [(x * self.grid_cell_size + self.grid_cell_size / 2,
                       y * self.grid_cell_size + self.grid_cell_size / 2) for x, y in
path_grid]

        # Просте згладжування маршруту
        if len(path_world) > 2:
            smoothed_path = [path_world[0]]
            for i in range(1, len(path_world) - 1):
                p1 = smoothed_path[-1]
                p2 = path_world[i+1]
                p_mid = path_world[i]
                try:
                    angle = math.atan2(p2[1]-p_mid[1], p2[0]-p_mid[0]) -
math.atan2(p_mid[1]-p1[1], p_mid[0]-p1[0])
                except (ValueError, ZeroDivisionError):
                    angle = 0
                if abs(angle) > 0.1:
                      smoothed_path.append(p_mid)
            smoothed_path.append(path_world[-1])
            print(f"Шлях згладжено з {len(path_world)} до {len(smoothed_path)}
точок.")
            return smoothed_path

        return path_world
```

```
class SimplePathFollower:
    """Контролер для слідування маршрутом, згенерованим A*"""

    def __init__(self, robot_controller):
        self.controller = robot_controller
        self.current_path: List[Tuple[float, float]] = []
        self.current_target_index: int = 0
        self.goal_tolerance: float = 0.15

    def set_path(self, path: List[Tuple[float, float]]):
        self.current_path = path if path else []
        self.current_target_index = 0
        if self.current_path:
            print(f"Встановлено новий маршрут з {len(self.current_path)} точок")
```

```
    def update(self) -> str:
        if not self.current_path:
            return 'no_path'

        if self.current_target_index >= len(self.current_path):
            print("Маршрут пройдено!")
            self.current_path = []
            return 'completed'
```

```
        target_x, target_y = self.current_path[self.current_target_index]
        current_x, current_y, current_theta = self.controller.x, self.controller.y,
self.controller.theta

        distance = math.sqrt((target_x - current_x)**2 + (target_y - current_y)**2)

        if distance < self.goal_tolerance:
            print(f"Точка {self.current_target_index} ({target_x:.2f},
{target_y:.2f}) досягнута.")
            self.current_target_index += 1
            return self.update()

        target_angle = math.atan2(target_y - current_y, target_x - current_x)
        angle_diff = math.atan2(math.sin(target_angle - current_theta),
math.cos(target_angle - current_theta))

        base_speed = 4.0
        turn_speed = 2.5

        orientation_threshold = math.radians(15) # 15 градусів
```

```
        if abs(angle_diff) > orientation_threshold:
            left_speed = -math.copysign(turn_speed, angle_diff)
            right_speed = math.copysign(turn_speed, angle_diff)
        else:
            turn_correction = angle_diff * 2.0
            left_speed = base_speed - turn_correction
            right_speed = base_speed + turn_correction
```

```
        self.controller.set_motor_speeds(left_speed, right_speed)
        return 'following'
```

```
class EnhancedSLAMController:
    def __init__(self):
        self.robot = Robot()
        self.timestep = int(self.robot.getBasicTimeStep())

        self.obstacle_map: Dict[Tuple[int, int], float] = {}
        self.free_map: Dict[Tuple[int, int], float] = {}
        self.grid_cell_size = 0.2
        self.x, self.y, self.theta = 0.0, 0.0, 0.0
        self.position_history: List[Tuple[float, float]] = []
        self.angle_history: List[float] = []
        self.path_planner = PathPlannerAStar(self)
        self.goal_position: Optional[Tuple[float, float]] = None
        self.path_follower = SimplePathFollower(self)
        self.state: str = "exploring"
        self.state_timer: int = 0
        self.stuck_counter: int = 0
        self.navigation_stats = { 'total_navigations': 0, 'successful_navigations':
0, 'failed_navigations': 0, 'obstacles_encountered': 0, 'stuck_events': 0 }
        self.current_sensor_readings: Dict[str, float] = {}
```

```
        self._avoidance_active = False
        self._avoidance_turn_direction = 0

        if not self.init_devices():
            raise RuntimeError("Проблема під час ініціалізації")

        print("Систему було ініціалізовано")
```

```
    def init_devices(self):
```

```
        motor_names = ['motor_left_front', 'motor_left_rear', 'motor_right_front',
'motor_right_rear']
        self.motors = {}
        for name in motor_names:
            motor = self.robot.getDevice(name)
            if motor:
                motor.setPosition(float('inf'))
                motor.setVelocity(0)
                self.motors[name] = motor
        if not self.motors:
            return False

        self.left_encoder = self.robot.getDevice('encoder_left_front')
        self.right_encoder = self.robot.getDevice('encoder_right_front')
        if not self.left_encoder or not self.right_encoder:
            print("Енкодери не знайдено.")
            self.encoders_available = False
        else:
            self.left_encoder.enable(self.timestep)
            self.right_encoder.enable(self.timestep)
            self.encoders_available = True
            self._encoder_initialized = False
```

```
        sensor_names_angles_map = {
            'ds_front_left': 0.5,
            'ds_front': 0.0,
            'ds_front_right': -0.5,
            'ds_left': 1.57,
            'ds_right': -1.57,
            'ds_back_left': 2.62,
            'ds_back_right': -2.62,
            'ds_back': 3.14159
        }

        self.sensors = {}
        self.sensor_angles = {}
        initialized_sensors = 0

        for name, angle in sensor_names_angles_map.items():
            sensor = self.robot.getDevice(name)
            if sensor:
                sensor.enable(self.timestep)
                self.sensors[name] = sensor
                self.sensor_angles[name] = angle
                initialized_sensors += 1
                print(f"Датчик {name:15} | Кут: {angle:6.2f}")
            else:
                print(f"Датчик {name:15} | НЕ ЗНАЙДЕНО !")

        print(f"Ініціалізовано датчиків: {initialized_sensors}/8")

        self.imu = self.robot.getDevice('imu')
        if self.imu:
            self.imu.enable(self.timestep)
            self._imu_initialized = False
        else:
            print("IMU не знайдено")

        self.wheel_radius = 0.085
        self.track_width = 0.50

        self._odometry_initialized = False
```

```
        self._initialization_steps = 0

        return True
```

```
    def navigate_to_goal(self, x: float, y: float) -> bool:
        """
        Метод для запуску навігації до точки
        """
```

```
        print(f"\nВстановлення нової глобальної цілі: ({x:.2f}, {y:.2f})")
        self.goal_position = (x, y)
        self.navigation_stats['total_navigations'] += 1
```

```
        planned_path = self.path_planner.plan_path((self.x, self.y),
self.goal_position)

        if planned_path:
            self.path_follower.set_path(planned_path)
            self.state = "navigating"
            self.state_timer = 0
            print("Маршрут сплановано. Режим навігації увімкнено")
            return True
        else:
            self.navigation_stats['failed_navigations'] += 1
            print("Не вдалося побудувати маршрут")
            self.goal_position = None
            self.state = "exploring"
            return False
```

```
    def run(self):
        step_counter = 0
        while self.robot.step(self.timestep) != -1:
            self.update_odometry()
            self.update_map()

            self.run_state_machine()
```

```
            if step_counter % 50 == 0:
                self.print_status()

            step_counter += 1
```

```
    def run_state_machine(self):
        self.state_timer += 1
        sensors_data = self._get_sensors_data()
```

```
        if self.detect_stuck():
            if self.state != "stuck":
                print(f"ВИЯВЛЕНО ЗАСТРЯГАННЯ! Перехід у режим порятунку.")
                self.navigation_stats['stuck_events'] += 1
                self.state = "stuck"
                self.state_timer = 0
            self._handle_stuck_state()
            return

        obstacle_threshold = 0.8
        front_sensors = ['ds_front_left', 'ds_front', 'ds_front_right', 'ds_left',
'ds_right']
        obstacle_detected = any(
            sensors_data.get(sensor_name, 4.0) < obstacle_threshold
```

```
            for sensor_name in front_sensors
        )

        if obstacle_detected and self.state != "avoiding_obstacle":
            print(f"Перешкода попереду! Активація режиму об'їзду.")
            self.navigation_stats['obstacles_encountered'] += 1
            self.state = "avoiding_obstacle"
            self.state_timer = 0
            self._avoidance_active = False

        if self.state == "avoiding_obstacle":
            self._handle_avoidance_state(sensors_data)
        elif self.state == "navigating":
            self._handle_navigation_state(sensors_data)
        elif self.state == "stuck":
            self._handle_stuck_state()
        else: # exploring
            self._handle_exploration_state(sensors_data)

    def _get_sensors_data(self) -> Dict[str, float]:
        sensors_data = {}
        for name, sensor_obj in self.sensors.items():
            val = sensor_obj.getValue()
            if math.isnan(val) or val < 0.01:
                sensors_data[name] = sensor_obj.getMaxValue()
            else:
                sensors_data[name] = val

        self.current_sensor_readings = sensors_data.copy()
        return sensors_data
```

```
    def _handle_navigation_state(self, sensors_data):

        if not self.path_follower.current_path:
            if self.goal_position:
                print("Перепланування маршруту")
                self.navigate_to_goal(*self.goal_position)
            else:
                print("Перехід в режим дослідження")
                self.state = "exploring"
            return
```

```
        follow_result = self.path_follower.update()

        if follow_result == 'completed':
            print("Глобальна ціль досягнута")
            self.navigation_stats['successful_navigations'] += 1
            self.stop()
            self.goal_position = None
            self.state = "exploring"
        elif follow_result == 'no_path':
            print("Маршрут втрачено. Увімкнення режиму дослідження")
            self.state = "exploring"
```

```
    def _handle_avoidance_state(self, sensors_data):
        if not self._avoidance_active:
            self.stop()
            self._avoidance_active = True
```

```
            left_freedom = sensors_data.get('ds_left', 0.0) +
sensors_data.get('ds_front_left', 0.0)
            right_freedom = sensors_data.get('ds_right', 0.0) +
sensors_data.get('ds_front_right', 0.0)

            if left_freedom > right_freedom:
                self._avoidance_turn_direction = -1 # Поворот ліворуч
                print("Поворот ліворуч")
            else:
                self._avoidance_turn_direction = 1 # Поворот праворуч
                print("Поворот праворуч")

        turn_duration = 60
        forward_duration = 80
```

```
        if self.state_timer <= turn_duration:
            turn_speed = 3.5
            left_speed = turn_speed * self._avoidance_turn_direction
            right_speed = -turn_speed * self._avoidance_turn_direction

            if self.state_timer % 25 == 0:
                  print(f"Об'їжджаємо. Напрямок: {self._avoidance_turn_direction}")
            self.set_motor_speeds(left_speed, right_speed)
```

```
        elif self.state_timer <= turn_duration + forward_duration:
            if self.state_timer == turn_duration + 1: # Повідомлення один раз
                print(f"Об'їжджаємо. Рух уперед")
            self.set_motor_speeds(4.0, 4.0)
```

```
        else:
            print("Перешкоду об'їхано")
            self.stop()
            self._avoidance_active = False
```

```
            if self.goal_position:
                print("Поточний стан: НАВІГАЦІЯ")
                self.state = "navigating"
                self.path_follower.set_path([])
            else:
                print("Поточний стан: ДОСЛІДЖЕННЯ")
                self.state = "exploring"

            self.state_timer = 0
            return
```

```
    def _handle_stuck_state(self):
        backup_duration = 50
        turn_duration = 70

        if self.state_timer == 1:
            left_freedom = self.current_sensor_readings.get('ds_left', 4.0) +
self.current_sensor_readings.get('ds_front_left', 4.0)
            right_freedom = self.current_sensor_readings.get('ds_right', 4.0) +
self.current_sensor_readings.get('ds_front_right', 4.0)

            if left_freedom > right_freedom:
                self._avoidance_turn_direction = -1
                print("поворот ліворуч, для вивільнення робота із застрягання")
            else:
                self._avoidance_turn_direction = 1
                print("поворот праворуч, для вивільнення робота із застрягання")
```

```
        if self.state_timer <= backup_duration:
            self.set_motor_speeds(-5.0, -5.0)
        elif self.state_timer <= backup_duration + turn_duration:
            turn_dir = self._avoidance_turn_direction if
self._avoidance_turn_direction != 0 else 1
            self.set_motor_speeds(5.0 * turn_dir, -5.0 * turn_dir)
        else:
            print("Агресивне вивільнення із застрягання завершено")
            self._exit_and_replan()

    def _handle_exploration_state(self, sensors_data):
        self.set_motor_speeds(3.5, 3.5)
```

```
    def _exit_and_replan(self):
        self.stop()
        self._avoidance_active = False
        self.stuck_counter = 0

        if self.goal_position:
            print("Відновлення навігації до цілі через перепланування...")
            self.navigate_to_goal(*self.goal_position)
        else:
            print("Об'їзд завершено. Повернення до режиму дослідження.")
            self.state = "exploring"
            self.state_timer = 0

    def update_odometry(self):
        if not self._odometry_initialized:
            self._initialization_steps += 1

            # Чекаємо на стабілізацію всіх сенсорів
            if self._initialization_steps < 20:
                # У перші кроки просто читаємо значення для стабілізації
                if self.encoders_available:
                    self.left_encoder.getValue()
                    self.right_encoder.getValue()
                if self.imu:
                    try:
                        self.imu.getRollPitchYaw()
                    except:
                        pass
                return

            # На 20-му кроці ініціалізуємо базові значення
            elif self._initialization_steps == 20:
                print("Ініціалізація одометрії...")

                # Ініціалізація енкодерів реальними значеннями
                if self.encoders_available:
                    try:
                        left_val = self.left_encoder.getValue()
                        right_val = self.right_encoder.getValue()

                        if not (math.isnan(left_val) or math.isnan(right_val)):
                            self.prev_left_encoder_val = left_val
                            self.prev_right_encoder_val = right_val
                            self._encoder_initialized = True
                            print(f"Енкодери ініціалізовані: L={left_val:.3f},
R={right_val:.3f}")
                        else:
                            print("Енкодери повернули NaN, повторна спроба...")
                            return
```

```
                    except Exception as e:
                        print(f"Помилка ініціалізації енкодерів: {e}")
                        return

                # Ініціалізація IMU
                if self.imu:
                    try:
                        roll_pitch_yaw = self.imu.getRollPitchYaw()
                        if roll_pitch_yaw and not any(math.isnan(v) for v in
roll_pitch_yaw):
                            self.theta = roll_pitch_yaw[2]
                            self._imu_initialized = True
                            print(f"IMU ініціалізовано:
кут={math.degrees(self.theta):.1f}°")
                        else:
                            print("IMU повернув некоректні дані, використовую
кінематичну одометрію")
                            self.theta = 0.0
                    except Exception as e:
                        print(f"Помилка ініціалізації IMU: {e}")
                        self.theta = 0.0
                else:
                    self.theta = 0.0

                # Ініціалізація позиції
                self.x, self.y = 0.0, 0.0
                self.position_history = [(0.0, 0.0)]

                # Завершення ініціалізації
                self._odometry_initialized = True
                print("Одометрію повністю ініціалізовано")
                return

        dt = self.timestep / 1000.0

        # Оновлення кута через IMU (якщо доступний та ініціалізований)
        if self.imu and self._imu_initialized:
            try:
                roll_pitch_yaw = self.imu.getRollPitchYaw()
                if roll_pitch_yaw and not any(math.isnan(v) for v in roll_pitch_yaw):
                    self.theta = roll_pitch_yaw[2]
            except Exception:
                pass

        # Оновлення позиції через енкодери
        if self.encoders_available and self._encoder_initialized:
            try:
                left_val = self.left_encoder.getValue()
                right_val = self.right_encoder.getValue()

                # Перевірка на коректність значень
                if math.isnan(left_val) or math.isnan(right_val):
                    print("Енкодери повернули NaN, пропускаємо оновлення")
                    return

                # Розрахунок переміщень
                dist_left = (left_val - self.prev_left_encoder_val) *
self.wheel_radius
                dist_right = (right_val - self.prev_right_encoder_val) *
self.wheel_radius

                # Захист від аномально великих значень (можуть бути в перші цикли)
```

```
                max_reasonable_dist = 0.5  # максимальне розумне переміщення за один цикл
                if abs(dist_left) > max_reasonable_dist or abs(dist_right) > max_reasonable_dist:
                    print(f"Аномально велике переміщення виявлено: L={dist_left:.3f}, R={dist_right:.3f}, пропускаємо")
                    self.prev_left_encoder_val = left_val
                    self.prev_right_encoder_val = right_val
                    return

                # Оновлення значень енкодерів
                self.prev_left_encoder_val = left_val
                self.prev_right_encoder_val = right_val

                # Кінематичний розрахунок
                delta_dist = (dist_left + dist_right) / 2.0
                delta_theta = (dist_right - dist_left) / self.track_width
```

```
                # Оновлення позиції
                self.x += delta_dist * math.cos(self.theta + delta_theta / 2.0)
                self.y += delta_dist * math.sin(self.theta + delta_theta / 2.0)

                # Якщо IMU недоступний, оновлюємо кут кінематично
                if not (self.imu and self._imu_initialized):
                    self.theta += delta_theta

            except Exception as e:
                print(f"Помилка в оновленні одометрії: {e}")
                return

        # Нормалізація кута
        self.theta = math.atan2(math.sin(self.theta), math.cos(self.theta))

        # Історія позицій для детектора застрягання
        self.position_history.append((self.x, self.y))
        if len(self.position_history) > 30:
            self.position_history.pop(0)
```

```
        self.angle_history.append(self.theta)
        if len(self.angle_history) > 30:
            self.angle_history.pop(0)
```

```
    def update_map(self):
        for name, sensor in self.sensors.items():
            distance = sensor.getValue()
            if math.isnan(distance) or distance >= sensor.getMaxValue() * 0.99:
                continue

            # Координати точки перешкоди
            sensor_global_angle = self.theta + self.sensor_angles[name]
            obs_x = self.x + distance * math.cos(sensor_global_angle)
            obs_y = self.y + distance * math.sin(sensor_global_angle)
            key = (int(obs_x / self.grid_cell_size), int(obs_y / self.grid_cell_size))
            self.obstacle_map[key] = min(1.0, self.obstacle_map.get(key, 0.0) + 0.25)

            # Оновлення вільного простору вздовж променя
            for step in range(int(distance / self.grid_cell_size) - 1):
                dist_along_ray = (step + 0.5) * self.grid_cell_size
                free_x = self.x + dist_along_ray * math.cos(sensor_global_angle)
                free_y = self.y + dist_along_ray * math.sin(sensor_global_angle)
```

```
                free_key = (int(free_x / self.grid_cell_size), int(free_y /
self.grid_cell_size))
                if self.obstacle_map.get(free_key, 0.0) < 0.5:
                    self.free_map[free_key] = min(1.0, self.free_map.get(free_key,
0.0) + 0.1)
```

```
    def detect_stuck(self) -> bool:
        history_len = 30
        if len(self.position_history) < history_len: return False

        # 1. Перевірка переміщення
        path_length = 0
        for i in range(1, len(self.position_history)):
            dx = self.position_history[i][0] - self.position_history[i-1][0]
            dy = self.position_history[i][1] - self.position_history[i-1][1]
            path_length += math.sqrt(dx*dx + dy*dy)
```

```
        # 2. Перевірка обертання
        total_rotation = 0
        for i in range(1, len(self.angle_history)):
            # Обчислюємо найкоротшу різницю кутів
            diff = self.angle_history[i] - self.angle_history[i-1]
            total_rotation += abs(math.atan2(math.sin(diff), math.cos(diff)))
```

```
        # Пороги, підібрані експериментально
        dist_threshold = 0.1  # Метри
        rot_threshold = math.radians(10) # 10 градусів
```

```
        # Спрацьовуємо, тільки якщо ОБИДВА показники низькі
        if path_length < dist_threshold and total_rotation < rot_threshold:
            self.stuck_counter += 1
            # Збільшуємо поріг спрацьовування для надійності
            return self.stuck_counter > 70
        else:
            self.stuck_counter = 0
            return False
```

```
    def set_motor_speeds(self, left_speed: float, right_speed: float):
        """Встановлює швидкість моторів."""
        max_vel = self.motors['motor_left_front'].getMaxVelocity()
        safe_left = max(-max_vel, min(max_vel, left_speed))
        safe_right = max(-max_vel, min(max_vel, right_speed))

        for name, motor in self.motors.items():
            if 'left' in name: motor.setVelocity(safe_left)
            if 'right' in name: motor.setVelocity(safe_right)
```

```
    def stop(self):
        self.set_motor_speeds(0, 0)
```

```
    def print_status(self):
        """Виводить поточний статус робота."""
        print(f"\n{'='*60}")
        print(f"СТАТУС: {self.state.upper()} | Час: {self.robot.getTime():.1f}с |
Таймер стану: {self.state_timer}")
        print(f"Позиція: X={self.x:.2f}м, Y={self.y:.2f}м,
Кут={math.degrees(self.theta):.1f}°")
        if self.goal_position:
            dist_to_goal = math.sqrt((self.x-self.goal_position[0])**2 + (self.y-
self.goal_position[1])**2)
            print(f"Ціль: ({self.goal_position[0]:.2f}, {self.goal_position[1]:.2f})
| Відстань: {dist_to_goal:.2f}м")
```

```
        else:
            print("Ціль: не встановлена")
        stats = self.navigation_stats
        print(f"Навігації:
{stats['successful_navigations']}/{stats['total_navigations']} | "
              f"Перешкод: {stats['obstacles_encountered']} | Застрягань:
{stats['stuck_events']}")

        if self.current_sensor_readings:
            print(f"Датчики ({len(self.current_sensor_readings)}/8):")
            s_data = []
            for name, distance in sorted(self.current_sensor_readings.items()):
                 s_data.append(f"{name:<14}: {distance:5.2f}м")
            for i in range(len(s_data) // 2):
                print(f"   {s_data[i]:<30} | {s_data[i + len(s_data)//2]}")

        print('='*60)
```

```
    def is_robot_stable(self) -> bool:
        """Перевіряє, чи стабілізувалися всі системи робота."""
        if not self._odometry_initialized:
            return False

        # Перевірка IMU
        if self.imu:
            try:
                rpy = self.imu.getRollPitchYaw()
                if not rpy or any(math.isnan(v) for v in rpy):
                    return False
            except:
                return False

        # Перевірка енкодерів
        if self.encoders_available:
            try:
                left_val = self.left_encoder.getValue()
                right_val = self.right_encoder.getValue()
                if math.isnan(left_val) or math.isnan(right_val):
                    return False
            except:
                return False

        # Перевірка дистанційних сенсорів
        for sensor in self.sensors.values():
            try:
                val = sensor.getValue()
                if math.isnan(val):
                    return False
            except:
                return False

        return True
```

```
def run_smart_navigation():
    # Створення контролера
    controller = EnhancedSLAMController()

    print("Ініціалізація та стабілізація систем...")

    initialization_steps = 0
```

```
    max_init_steps = 100  # Максимум 100 кроків (залежить від timestep) на
ініціалізацію

    while initialization_steps < max_init_steps:
        if controller.robot.step(controller.timestep) == -1:
            break

        # Оновлюємо одометрію (яка сама контролює ініціалізацію)
        controller.update_odometry()

        # Перевіряємо готовність після мінімального часу
        if initialization_steps > 30 and controller.is_robot_stable():
            print(f"Системи стабілізовані за {initialization_steps} кроків")
            break

        initialization_steps += 1

        # Індикатор прогресу
        if initialization_steps % 20 == 0:
            print(f"   Ініціалізація... {initialization_steps}/{max_init_steps}")

    if initialization_steps >= max_init_steps:
        print("Перевищено час ініціалізації, але продовжуємо роботу")

    # Додаткові кроки для повної стабілізації
    print("Додаткова стабілізація...")
    for step in range(20):
        if controller.robot.step(controller.timestep) == -1:
            break
        controller.update_odometry()
        controller.update_map()

    print(f"\nРОБОТ ГОТОВИЙ ДО РОБОТИ")
    print(f"   Початкова позиція: X={controller.x:.3f}м, Y={controller.y:.3f}м")
    print(f"   Початковий кут: {math.degrees(controller.theta):.1f}°")
    print(f"   Енкодери: {'Так' if controller._encoder_initialized else 'Ні'}")
    print(f"   IMU: {'Так' if controller._imu_initialized else 'Ні'}")
    print(f"   Датчики: {len(controller.sensors)}/8")

    print("\n" + "="*40)
    print("ЗАПУСК НАВІГАЦІЙНОЇ МІСІЇ")
    print("="*40)

    success = controller.navigate_to_goal(4.0, 0)

    if not success:
        print("Не вдалося почати навігацію, перехід у режим дослідження")

    controller.run()
```

```
if __name__ == "__main__":
    run_smart_navigation()
```

ДОДАТОК В

Програмний код контроллеру з API для роботи із командний центром

```
import json
import threading
import time
import math
from http.server import HTTPServer, BaseHTTPRequestHandler
import socketserver
from typing import Tuple, List
from queue import Queue, Empty
from best_variant import EnhancedSLAMController


class NonBlockingHTTPHandler(BaseHTTPRequestHandler):
    def do_POST(self):
        if self.path == '/api/set_goal':
            try:
                content_length = int(self.headers['Content-Length'])
                post_data = self.rfile.read(content_length)
                data = json.loads(post_data.decode('utf-8'))

                # Поміщаємо команду в чергу замість прямого виконання
                self.server.command_queue.put({
                    'type': 'set_goal',
                    'x': float(data['x']),
                    'y': float(data['y']),
                    'timestamp': time.time()
                })

                self._send_response(200, {'status': 'success', 'message': 'Ціль
додано до черги'})

            except Exception as e:
                self._send_response(400, {'status': 'error', 'message': str(e)})

        elif self.path == '/api/set_mode':
            try:
                content_length = int(self.headers['Content-Length'])
                post_data = self.rfile.read(content_length)
                data = json.loads(post_data.decode('utf-8'))

                self.server.command_queue.put({
                    'type': 'set_mode',
                    'mode': data['mode'],
                    'timestamp': time.time()
                })

                self._send_response(200, {'status': 'success', 'message': f'Режим
{data["mode"]} додано до черги'})

            except Exception as e:
                self._send_response(400, {'status': 'error', 'message': str(e)})
        else:
            self._send_response(404, {'status': 'error', 'message': 'Endpoint не
знайдено'})
```

```
    def do_GET(self):
        if self.path == '/api/status':
            try:
                with open('robot_status.json', 'r') as f:
                    status = json.load(f)
                self._send_response(200, status)
            except (FileNotFoundError, PermissionError, json.JSONDecodeError):
                self._send_response(503, {'status': 'error', 'message': 'Файл статусу
тимчасово недоступний, спробуйте пізніше.'})

        elif self.path == '/api/sensors':
            try:
                with open('robot_sensors.json', 'r') as f:
                    sensors = json.load(f)
                self._send_response(200, sensors)
            except (FileNotFoundError, PermissionError, json.JSONDecodeError):
                self._send_response(503, {'status': 'error', 'message': 'Дані
датчиків тимчасово недоступні, спробуйте пізніше.'})
```

```
        elif self.path == '/api/obstacles':
            try:
                with open('robot_obstacles.json', 'r') as f:
                    obstacles = json.load(f)
                self._send_response(200, obstacles)
            except (FileNotFoundError, PermissionError, json.JSONDecodeError):
                self._send_response(503, {'status': 'error', 'message': 'Дані про
перешкоди тимчасово недоступні, спробуйте пізніше.'})
```

```
        else:
            self._send_response(404, {'status': 'error', 'message': 'Endpoint не
знайдено'})
```

```
    def _send_response(self, code, data):
        self.send_response(code)
        self.send_header('Content-type', 'application/json')
        self.send_header('Access-Control-Allow-Origin', '*')  # CORS
        self.send_header('Access-Control-Allow-Methods', 'GET, POST')
        self.send_header('Access-Control-Allow-Headers', 'Content-Type')
        self.end_headers()
        self.wfile.write(json.dumps(data).encode())
```

```
    def do_OPTIONS(self):
        # CORS preflight
        self.send_response(200)
        self.send_header('Access-Control-Allow-Origin', '*')
        self.send_header('Access-Control-Allow-Methods', 'GET, POST')
        self.send_header('Access-Control-Allow-Headers', 'Content-Type')
        self.end_headers()
```

```
    def log_message(self, format, *args):
        pass
```

```
class ThreadedHTTPServer(socketserver.ThreadingMixIn, HTTPServer):
    """ВИПРАВЛЕНО: Сервер тепер зберігає пряме посилання на контролер"""
    def __init__(self, server_address, RequestHandlerClass, controller_instance):
        self.command_queue = Queue()
        self.controller = controller_instance # Пряме посилання на об'єкт контролера
        super().__init__(server_address, RequestHandlerClass)
```

```
class EnhancedSLAMControllerWithAPI(EnhancedSLAMController):
```

```
    """
    контролер з неблокуючим API та експортом даних датчиків
    """
    def __init__(self, api_port=8080):
        super().__init__()
```

```
        # Компоненти API
        self.api_port = api_port
        self.command_queue = Queue()
        self.api_server = None
        self.api_thread = None
```

```
        # Нові режими
        self.scanning_mode = False
        self.scanning_grid_size = 1.0  # метри між точками сканування
        self.scanning_path = []
        self.scanning_current_index = 0
```

```
        # розширена карта для експорту з даними датчиків
        self.detailed_map = {
            'obstacles': {},
            'free_space': {},
            'scan_points': [],
            'robot_path': [],
            'sensor_data': {                # дані датчиків
                'current_readings': {},
                'danger_zones': [],
                'safe_zones': [],
                'last_update': 0
            },
            'obstacle_detections': [],
            'metadata': {
                'grid_size': self.grid_cell_size,
                'last_update': 0,
                'scan_coverage': 0,
                'total_sensors': len(self.sensors),
                'sensor_names': list(self.sensors.keys()) if hasattr(self, 'sensors')
else []
            }
        }

        self._start_api_server()
        print(f"API сервер запущено на порту {api_port}")
```

```
    def _start_api_server(self):
        """Запускає API сервер в окремому потоці"""
        try:
            def handler(*args, **kwargs):
                NonBlockingHTTPHandler(*args, **kwargs)
```

```
            self.api_server = ThreadedHTTPServer(
                ('localhost', self.api_port),
                NonBlockingHTTPHandler,
                self # Передаємо екземпляр контролера
            )
            self.api_server.command_queue = self.command_queue
```

```
            self.api_thread = threading.Thread(target=self.api_server.serve_forever,
daemon=True)
            self.api_thread.start()
```

```
            print(f"API сервер готовий: http://localhost:{self.api_port}")
```

```
        except Exception as e:
            print(f"Помилка запуску API сервера: {e}")
```

```
    def process_api_commands(self):
        try:
            while True:
                command = self.command_queue.get_nowait()
                self._execute_command(command)
        except Empty:
            pass  # Черга порожня - це нормально
```

```
    def _execute_command(self, command):
        cmd_type = command.get('type')

        if cmd_type == 'set_goal':
            x, y = command['x'], command['y']
            print(f"API команда: Встановлення цілі ({x:.2f}, {y:.2f})")
            success = self.navigate_to_goal(x, y)
            if not success:
                print("Не вдалося встановити ціль")
```

```
        elif cmd_type == 'set_mode':
            mode = command['mode']
            print(f"API команда: Встановлення режиму '{mode}'")

            if mode == 'scanning':
                self.start_scanning_mode()
            elif mode == 'navigation':
                self.stop_scanning_mode()
            elif mode == 'exploring':
                self.state = 'exploring'
                self.goal_position = None
            elif mode == 'stop':
                self.stop()
                self.state = 'exploring'
```

```
        else:
            print(f"Невідома команда: {cmd_type}")
```

```
    def start_scanning_mode(self):
        print("ЗАПУСК РЕЖИМУ СИСТЕМАТИЧНОГО СКАНУВАННЯ")

        self.scanning_mode = True
        self.state = "scanning"

        # Генеруємо сітку точок для сканування (спіральний патерн)
        self.scanning_path = self._generate_scanning_path()
        self.scanning_current_index = 0

        if self.scanning_path:
            print(f"Створено маршрут сканування з {len(self.scanning_path)} точок")
            # Йдемо до першої точки
            self._go_to_next_scan_point()
        else:
            print("Не вдалося створити маршрут сканування")
            self.scanning_mode = False
```

```
    def stop_scanning_mode(self):
        print("ЗУПИНКА РЕЖИМУ СКАНУВАННЯ")
        self.scanning_mode = False
        self.state = "exploring"
        self.scanning_path = []
```

```
    def _generate_scanning_path(self) -> List[Tuple[float, float]]:
        """Генерує спіральний шлях для систематичного сканування"""
        scan_points = []

        # Починаємо від поточної позиції робота
        start_x, start_y = self.x, self.y

        # Створюємо спіраль з кроком scanning_grid_size
        max_radius = 5.0  # максимальний радіус сканування (метри)
        angle_step = 0.5  # радіани

        radius = self.scanning_grid_size
        angle = 0

        while radius <= max_radius:
            scan_x = start_x + radius * math.cos(angle)
            scan_y = start_y + radius * math.sin(angle)

            scan_points.append((scan_x, scan_y))

            angle += angle_step
            if angle >= 2 * math.pi:
                angle = 0
                radius += self.scanning_grid_size

        return scan_points
```

```
    def _go_to_next_scan_point(self):
        if self.scanning_current_index >= len(self.scanning_path):
            print("СКАНУВАННЯ ЗАВЕРШЕНО!")
            self.stop_scanning_mode()
            return

        target_x, target_y = self.scanning_path[self.scanning_current_index]
        print(f"Сканування точки {self.scanning_current_index +
1}/{len(self.scanning_path)}: ({target_x:.2f}, {target_y:.2f})")

        success = self.navigate_to_goal(target_x, target_y)
        if not success:
            print(f"Не вдалося доїхати до точки сканування, пропускаємо")
            self.scanning_current_index += 1
            self._go_to_next_scan_point()
```

```
    def run_state_machine(self):
        # Обробляємо команди API
        self.process_api_commands()

        # Стандартна логіка
        if self.state == "scanning":
            self._handle_scanning_state()
        else:
            super().run_state_machine()
```

```
    def _handle_scanning_state(self):
        if not self.scanning_mode:
            self.state = "exploring"
            return

        sensors_data = self._get_sensors_data()

        # ОНОВЛЕНО: Перевіряємо перешкоди зі збільшеним порогом
```

```
        critical_threshold = 0.8  # Використовуємо новий поріг
        if self.detect_stuck() or any(v < critical_threshold for v in
sensors_data.values()):
            # Обходимо перешкоду стандартним способом
            super().run_state_machine()
            return

        # Перевіряємо, чи досягли поточної точки сканування
        if self.goal_position:
            dist_to_goal = math.sqrt((self.x - self.goal_position[0])**2 +
                                     (self.y - self.goal_position[1])**2)
            if dist_to_goal < 0.3:  # Досягли точки
                print(f"Точка сканування {self.scanning_current_index + 1}
досягнута")

                # Поворот на 360° для повного сканування
                self._perform_360_scan()

                # Перехід до наступної точки
                self.scanning_current_index += 1
                self._go_to_next_scan_point()

        # Звичайна навігація до точки
        follow_result = self.path_follower.update()
        if follow_result == 'completed':
            self.scanning_current_index += 1
            self._go_to_next_scan_point()
```

```
    def _perform_360_scan(self):
        print("Виконується 360° сканування...")

        start_angle = self.theta
        total_rotation = 0

        # Повільний поворот для якісного сканування
        while total_rotation < 2 * math.pi:
            self.set_motor_speeds(1.0, -1.0)  # Повільний поворот

            # Симулюємо один крок
            if self.robot.step(self.timestep) == -1:
                break

            self.update_odometry()
            self.update_map()  #Оновлюємо карту під час повороту

            # Підрахунок повороту
            angle_diff = abs(self.theta - start_angle)
            if angle_diff > math.pi:
                angle_diff = 2 * math.pi - angle_diff
            total_rotation = angle_diff

        self.stop()
        print("360° сканування завершено")
```

```
    def update_map(self):
        super().update_map()

        # Додаємо поточну позицію в шлях робота
        self.detailed_map['robot_path'].append([self.x, self.y, self.theta,
time.time()])

        # Обмежуємо розмір шляху (зберігаємо останні 1000 точок)
```

```
        if len(self.detailed_map['robot_path']) > 1000:
            self.detailed_map['robot_path'] = self.detailed_map['robot_path'][-1000:]

        #Оновлюємо дані датчиків
        current_time = time.time()
        self.detailed_map['sensor_data']['current_readings'] =
self.current_sensor_readings.copy()
        self.detailed_map['sensor_data']['last_update'] = current_time

        #Аналізуємо зони небезпеки
        danger_zones = []
        safe_zones = []

        for sensor_name, distance in self.current_sensor_readings.items():
            sensor_angle = self.sensor_angles.get(sensor_name, 0)
            global_angle = self.theta + sensor_angle

            # Координати точки виявлення
            detection_x = self.x + distance * math.cos(global_angle)
            detection_y = self.y + distance * math.sin(global_angle)

            zone_info = {
                'sensor': sensor_name,
                'distance': distance,
                'angle': math.degrees(global_angle),
                'world_x': detection_x,
                'world_y': detection_y,
                'timestamp': current_time
            }

            if distance < 1.0:
                danger_zones.append(zone_info)
            elif distance > 3.0:
                safe_zones.append(zone_info)

        self.detailed_map['sensor_data']['danger_zones'] = danger_zones
        self.detailed_map['sensor_data']['safe_zones'] = safe_zones

        if any(d < 0.8 for d in self.current_sensor_readings.values()):
            obstacle_detection = {
                'timestamp': current_time,
                'robot_position': [self.x, self.y, self.theta],
                'state': self.state,
                'closest_obstacles': [
                    {'sensor': name, 'distance': dist}
                    for name, dist in self.current_sensor_readings.items()
                    if dist < 1.5
                ],
                'action_taken': 'obstacle_avoidance' if self.state ==
'avoiding_obstacle' else 'normal_navigation'
            }

            self.detailed_map['obstacle_detections'].append(obstacle_detection)

            # Обмежуємо історію виявлень
            if len(self.detailed_map['obstacle_detections']) > 100:
                self.detailed_map['obstacle_detections'] =
self.detailed_map['obstacle_detections'][-100:]

        # Оновлюємо детальну карту
        self.detailed_map['obstacles'] = {f"{k[0]},{k[1]}": v for k, v in
self.obstacle_map.items()}
```

```
        self.detailed_map['free_space'] = {f"{k[0]},{k[1]}": v for k, v in
self.free_map.items()}

        # Метадані
        self.detailed_map['metadata'].update({
            'last_update': current_time,
            'scan_coverage': len(self.obstacle_map) + len(self.free_map),
            'current_position': [self.x, self.y, self.theta],
            'current_state': self.state,
            'scanning_progress':
f"{self.scanning_current_index}/{len(self.scanning_path)}" if self.scanning_mode else
"N/A",
            'total_sensors': len(self.sensors),
            'active_sensors': len([s for s in self.current_sensor_readings.values()
if not math.isnan(s)])
        })
```

```
    def export_detailed_map(self):
        try:
            # Додаємо scan_points в режимі сканування
            if self.scanning_mode:
                self.detailed_map['scan_points'] = [
                    {'x': point[0], 'y': point[1], 'completed': i <
self.scanning_current_index}
                    for i, point in enumerate(self.scanning_path)
                ]

            self._atomic_write_json('robot_map.json', self.detailed_map)

        except Exception as e:
            print(f"Помилка експорту карти: {e}")
```

```
    def export_sensor_data(self):
        try:
            sensor_export = {
                'timestamp': time.time(),
                'robot_position': [self.x, self.y, self.theta],
                'sensors': {},
                'analysis': {
                    'closest_obstacle': None,
                    'furthest_detection': None,
                    'danger_level': 'safe',
                    'recommended_action': 'continue'
                }
            }

            # Детальні дані кожного датчика
            min_distance = float('inf')
            max_distance = 0
            closest_sensor = None

            for sensor_name, distance in self.current_sensor_readings.items():
                if sensor_name in self.sensor_angles:
                    sensor_angle = self.sensor_angles[sensor_name]
                    global_angle = self.theta + sensor_angle

                    # Світові координати точки виявлення
                    world_x = self.x + distance * math.cos(global_angle)
                    world_y = self.y + distance * math.sin(global_angle)

                    sensor_export['sensors'][sensor_name] = {
                        'distance': distance,
```

```
                        'angle_local': math.degrees(sensor_angle),
                        'angle_global': math.degrees(global_angle),
                        'world_position': [world_x, world_y],
                        'status': 'danger' if distance < 1.0 else 'warning' if
distance < 2.0 else 'safe'
                    }

                    # Аналіз для загальних метрик
                    if distance < min_distance:
                        min_distance = distance
                        closest_sensor = sensor_name

                    max_distance = max(max_distance, distance)

            # Загальний аналіз ситуації
            if min_distance < 0.5:
                sensor_export['analysis']['danger_level'] = 'critical'
                sensor_export['analysis']['recommended_action'] = 'emergency_stop'
            elif min_distance < 1.0:
                sensor_export['analysis']['danger_level'] = 'high'
                sensor_export['analysis']['recommended_action'] = 'avoid_obstacle'
            elif min_distance < 2.0:
                sensor_export['analysis']['danger_level'] = 'medium'
                sensor_export['analysis']['recommended_action'] = 'slow_down'

            sensor_export['analysis']['closest_obstacle'] = {
                'sensor': closest_sensor,
                'distance': min_distance
            }
            sensor_export['analysis']['furthest_detection'] = max_distance

            # Експорт у файл
            self._atomic_write_json('robot_sensors.json', sensor_export)

        except Exception as e:
            print(f"Помилка експорту даних датчиків: {e}")
```

```
    def export_obstacle_data(self):
        """НОВИЙ МЕТОД: Експортує карту перешкод в окремий файл"""
        try:
            obstacle_export = {
                'timestamp': time.time(),
                'robot_position': [self.x, self.y, self.theta],
                'grid_size': self.grid_cell_size,
                'obstacles': {},
                'free_cells': {},
                'statistics': {
                    'total_obstacles': len(self.obstacle_map),
                    'total_free_cells': len(self.free_map),
                    'map_coverage': len(self.obstacle_map) + len(self.free_map),
                    'obstacle_density': len(self.obstacle_map) / max(1,
len(self.obstacle_map) + len(self.free_map))
                }
            }

            # Конвертуємо координати сітки в світові координати
            for grid_pos, probability in self.obstacle_map.items():
                world_x = grid_pos[0] * self.grid_cell_size + self.grid_cell_size / 2
                world_y = grid_pos[1] * self.grid_cell_size + self.grid_cell_size / 2

                obstacle_export['obstacles'][f"{grid_pos[0]},{grid_pos[1]}"] = {
                    'probability': probability,
```

```
                    'world_position': [world_x, world_y],
                    'grid_position': [grid_pos[0], grid_pos[1]]
                }

            for grid_pos, probability in self.free_map.items():
                world_x = grid_pos[0] * self.grid_cell_size + self.grid_cell_size / 2
                world_y = grid_pos[1] * self.grid_cell_size + self.grid_cell_size / 2

                obstacle_export['free_cells'][f"{grid_pos[0]},{grid_pos[1]}"] = {
                    'probability': probability,
                    'world_position': [world_x, world_y],
                    'grid_position': [grid_pos[0], grid_pos[1]]
                }

            # Експорт у файл
            self._atomic_write_json('robot_obstacles.json', obstacle_export)

        except Exception as e:
            print(f"Помилка експорту перешкод: {e}")
```

```
    def export_status(self):
        """ОНОВЛЕНО: Експортує поточний статус робота з даними датчиків"""
        status = {
            'position': [self.x, self.y, self.theta],
            'state': self.state,
            'timestamp': time.time(),
            'goal': list(self.goal_position) if self.goal_position else None,
            'scanning_mode': self.scanning_mode,
            'scanning_progress': {
                'current': self.scanning_current_index,
                'total': len(self.scanning_path),
                'percentage': round((self.scanning_current_index /
len(self.scanning_path)) * 100, 1) if self.scanning_path else 0
            },
            'map_stats': {
                'obstacles': len(self.obstacle_map),
                'free_cells': len(self.free_map),
                'total_coverage': len(self.obstacle_map) + len(self.free_map)
            },
            # Розширена інформація про датчики
            'sensor_stats': {
                'total_sensors': len(self.sensors),
                'active_sensors': len([s for s in
self.current_sensor_readings.values() if not math.isnan(s)]),
                'current_readings': self.current_sensor_readings,
                'danger_sensors': [
                    name for name, dist in self.current_sensor_readings.items()
                    if dist < 1.0
                ],
                'warning_sensors': [
                    name for name, dist in self.current_sensor_readings.items()
                    if 1.0 <= dist < 2.0
                ],
                'closest_obstacle': {
                    'distance': min(self.current_sensor_readings.values()) if
self.current_sensor_readings else float('inf'),
                    'sensor': min(self.current_sensor_readings.items(), key=lambda x:
x[1])[0] if self.current_sensor_readings else None
                }
            },
            'navigation_stats': self.navigation_stats
        }
```

```
        try:
            self._atomic_write_json('robot_status.json', status)
        except Exception as e:
            print(f"Помилка експорту статусу: {e}")
```

```
    def _atomic_write_json(self, final_filename: str, data: dict):
        temp_filename = f"{final_filename}.tmp"
        import os
        try:
            #Записуємо дані у тимчасовий файл
            with open(temp_filename, 'w') as f:
                json.dump(data, f, indent=2)

            #Намагаємося замінити цільовий файл. os.replace - атомарна операція.
            os.replace(temp_filename, final_filename)
```

```
        except PermissionError:
            #Якщо заміна не вдалася через блокування пробуємо видалити і
перейменувати
            print(f"Блокування файлу {final_filename}. Повторна спроба...")
            try:
                os.remove(final_filename)
                os.rename(temp_filename, final_filename)
            except Exception as e_retry:
                print(f"Помилка при повторній спробі запису в {final_filename}:
{e_retry}")

        except Exception as e:
            print(f"Критична помилка запису в {final_filename}: {e}")
            if os.path.exists(temp_filename):
                os.remove(temp_filename)
```

```
    def run(self):
        """Основний цикл з експортом даних"""
        step_counter = 0
        export_interval = 25  # Експорт кожні 25 кроків (~0.5 сек)

        print("Розширений контролер запущений з підтримкою API")
        print(f"API доступний за адресою: http://localhost:{self.api_port}")
        print("Доступні команди:")
        print("   POST /api/set_goal    {'x': 1.0, 'y': 2.0}")
        print("   POST /api/set_mode    {'mode':
'scanning'|'navigation'|'exploring'|'stop'}")
        print("   GET  /api/status      - загальний статус")
        print("   GET  /api/sensors     - дані датчиків")
        print("   GET  /api/obstacles   - карта перешкод")

        while self.robot.step(self.timestep) != -1:
            self.update_odometry()
            self.update_map()
            self.run_state_machine()

            # РОЗШИРЕНИЙ періодичний експорт даних
            if step_counter % export_interval == 0:
                self.export_detailed_map()
                self.export_status()
                self.export_sensor_data()      # НОВЕ
                self.export_obstacle_data()    # НОВЕ

            # Відлагоджувальна інформація
            if step_counter % 50 == 0:
```

```
                self.print_status()

            step_counter += 1
```

```
    def print_status(self):
        """РОЗШИРЕНИЙ вивід статусу з інформацією про датчики"""
        super().print_status()
        if self.scanning_mode:
            print(f"РЕЖИМ СКАНУВАННЯ:
{self.scanning_current_index}/{len(self.scanning_path)} точок")
            print(f"Покриття карти: {len(self.obstacle_map) + len(self.free_map)}
комірок")
```

```
def run_robot_with_api():
    """Запуск робота з підтримкою API та експортом даних датчиків"""
    controller = EnhancedSLAMControllerWithAPI(api_port=8080)

    print("Ініціалізація з підтримкою API...")

    # Стандартна ініціалізація
    initialization_steps = 0
    max_init_steps = 100

    while initialization_steps < max_init_steps:
        if controller.robot.step(controller.timestep) == -1:
            break

        controller.update_odometry()

        if initialization_steps > 30 and controller.is_robot_stable():
            print(f"Системи стабілізовані за {initialization_steps} кроків")
            break

        initialization_steps += 1

    # Додаткова стабілізація
    for step in range(20):
        if controller.robot.step(controller.timestep) == -1:
            break
        controller.update_odometry()
        controller.update_map()

    print(f"\nРОБОТ З API ГОТОВИЙ")
    print(f"API: http://localhost:{controller.api_port}")
    print(f"Позиція: ({controller.x:.3f}, {controller.y:.3f})")
    print(f"Датчиків: {len(controller.sensors)}/8")

    # Експорт початкового стану
    controller.export_detailed_map()
    controller.export_status()
    controller.export_sensor_data()
    controller.export_obstacle_data()

    # Запуск основного циклу
    controller.run()
```

```
if __name__ == "__main__":
    run_robot_with_api()
```

ДОДАТОК Г

Програмний код веб додатку із телеметрією

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Переглядач робота Phoenix з 8 датчиками</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 20px;
            background: linear-gradient(135deg, #667eea 0%, #764ba2 100%);
            color: #333;
            min-height: 100vh;
        }

        .container {
            max-width: 1400px;
            margin: 0 auto;
            background: rgba(255, 255, 255, 0.95);
            padding: 20px;
            border-radius: 15px;
            box-shadow: 0 8px 32px rgba(0,0,0,0.1);
            backdrop-filter: blur(10px);
        }

        h1 {
            text-align: center;
            margin-bottom: 30px;
            color: #2c3e50;
            text-shadow: 2px 2px 4px rgba(0,0,0,0.1);
        }

        .dashboard {
            display: grid;
            grid-template-columns: 1fr 1fr;
            gap: 20px;
            margin-bottom: 20px;
        }

        @media (max-width: 1200px) {
            .dashboard {
                grid-template-columns: 1fr;
            }
        }

        .connection-status {
            display: inline-flex;
            align-items: center;
            padding: 8px 15px;
            border-radius: 25px;
            font-size: 0.9em;
            margin-bottom: 20px;
            font-weight: 500;
        }
```

```
        .connection-status.connected {
            background: linear-gradient(135deg, #4ade80, #22c55e);
            color: white;
            box-shadow: 0 4px 12px rgba(34, 197, 94, 0.3);
        }

        .connection-status.disconnected {
            background: linear-gradient(135deg, #f87171, #ef4444);
            color: white;
            box-shadow: 0 4px 12px rgba(239, 68, 68, 0.3);
        }

        .status-dot {
            width: 10px;
            height: 10px;
            border-radius: 50%;
            margin-right: 8px;
            animation: pulse 2s infinite;
        }

        @keyframes pulse {
            0%, 100% { opacity: 1; }
            50% { opacity: 0.7; }
        }

        .panel {
            background: white;
            border-radius: 12px;
            padding: 20px;
            box-shadow: 0 4px 20px rgba(0,0,0,0.08);
            border: 1px solid rgba(0,0,0,0.05);
        }

        .panel h3 {
            margin: 0 0 15px 0;
            color: #2c3e50;
            font-size: 1.2em;
            display: flex;
            align-items: center;
            gap: 10px;
        }

        .status-grid {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(250px, 1fr));
            gap: 15px;
        }

        .status-card {
            background: linear-gradient(135deg, #667eea, #764ba2);
            color: white;
            padding: 20px;
            border-radius: 10px;
            text-align: center;
            box-shadow: 0 4px 15px rgba(102, 126, 234, 0.3);
        }

        .status-card h4 {
            margin: 0 0 10px 0;
            font-size: 0.9em;
            opacity: 0.9;
        }
```

```
        .status-value {
            font-size: 1.8em;
            font-weight: bold;
            margin-bottom: 5px;
        }

        .sensor-grid {
            display: grid;
            grid-template-columns: repeat(auto-fit, minmax(200px, 1fr));
            gap: 10px;
        }

        .sensor-card {
            background: #f8fafc;
            border: 2px solid #e2e8f0;
            border-radius: 8px;
            padding: 15px;
            text-align: center;
            transition: all 0.3s ease;
        }

        .sensor-card.danger {
            border-color: #ef4444;
            background: #fef2f2;
            animation: glow-red 2s infinite;
        }

        .sensor-card.warning {
            border-color: #f59e0b;
            background: #fffbeb;
        }

        .sensor-card.safe {
            border-color: #10b981;
            background: #f0fdf4;
        }

        @keyframes glow-red {
            0%, 100% { box-shadow: 0 0 5px rgba(239, 68, 68, 0.5); }
            50% { box-shadow: 0 0 20px rgba(239, 68, 68, 0.8); }
        }

        .sensor-name {
            font-weight: bold;
            font-size: 0.9em;
            margin-bottom: 8px;
            color: #374151;
        }

        .sensor-distance {
            font-size: 1.4em;
            font-weight: bold;
            margin-bottom: 5px;
        }

        .sensor-status {
            font-size: 0.8em;
            padding: 4px 8px;
            border-radius: 12px;
            font-weight: 500;
        }
```

```
        .status-danger { background: #fecaca; color: #dc2626; }
        .status-warning { background: #fef3c7; color: #d97706; }
        .status-safe { background: #d1fae5; color: #059669; }

        .controls {
            background: #f8fafc;
            padding: 20px;
            border-radius: 10px;
            margin-bottom: 20px;
            border: 1px solid #e2e8f0;
        }

        .controls h3 {
            margin: 0 0 15px 0;
            color: #374151;
        }

        .btn {
            background: linear-gradient(135deg, #667eea, #764ba2);
            color: white;
            border: none;
            padding: 12px 24px;
            border-radius: 8px;
            cursor: pointer;
            margin-right: 10px;
            margin-bottom: 10px;
            font-weight: 500;
            transition: all 0.3s ease;
            box-shadow: 0 4px 12px rgba(102, 126, 234, 0.3);
        }

        .btn:hover {
            transform: translateY(-2px);
            box-shadow: 0 6px 20px rgba(102, 126, 234, 0.4);
        }

        .btn.danger {
            background: linear-gradient(135deg, #ef4444, #dc2626);
            box-shadow: 0 4px 12px rgba(239, 68, 68, 0.3);
        }

        .btn.success {
            background: linear-gradient(135deg, #10b981, #059669);
            box-shadow: 0 4px 12px rgba(16, 185, 129, 0.3);
        }

        .btn.warning {
            background: linear-gradient(135deg, #f59e0b, #d97706);
            box-shadow: 0 4px 12px rgba(245, 158, 11, 0.3);
        }

        input[type="number"] {
            padding: 10px;
            margin: 0 8px;
            width: 80px;
            border: 2px solid #e2e8f0;
            border-radius: 6px;
            font-size: 1em;
        }

        input[type="number"]:focus {
```

```
            border-color: #667eea;
            outline: none;
            box-shadow: 0 0 0 3px rgba(102, 126, 234, 0.1);
        }

        #map-container {
            border: 2px solid #e2e8f0;
            border-radius: 12px;
            overflow: hidden;
            margin-bottom: 20px;
            background: white;
            position: relative;
        }

        #map-canvas {
            display: block;
            width: 100%;
            height: 500px;
            background: linear-gradient(45deg, #f8fafc 25%, transparent 25%),
                        linear-gradient(-45deg, #f8fafc 25%, transparent 25%),
                        linear-gradient(45deg, transparent 75%, #f8fafc 75%),
                        linear-gradient(-45deg, transparent 75%, #f8fafc 75%);
            background-size: 20px 20px;
            background-position: 0 0, 0 10px, 10px -10px, -10px 0px;
            cursor: crosshair;
        }

        .map-info {
            position: absolute;
            top: 10px;
            left: 10px;
            background: rgba(0, 0, 0, 0.8);
            color: white;
            padding: 8px 12px;
            border-radius: 6px;
            font-size: 0.9em;
            font-family: 'Courier New', monospace;
        }

        .log {
            background: #1f2937;
            color: #f9fafb;
            border-radius: 10px;
            padding: 20px;
            height: 200px;
            overflow-y: auto;
            font-family: 'Courier New', monospace;
            font-size: 0.9em;
            line-height: 1.4;
        }

        .log-entry {
            margin-bottom: 5px;
            padding: 2px 0;
        }

        .log-timestamp {
            color: #9ca3af;
            margin-right: 8px;
        }

        .emergency-alert {
```

```
            background: linear-gradient(135deg, #ef4444, #dc2626);
            color: white;
            padding: 15px;
            border-radius: 10px;
            margin-bottom: 20px;
            text-align: center;
            font-weight: bold;
            animation: emergency-blink 1s infinite;
            box-shadow: 0 4px 20px rgba(239, 68, 68, 0.4);
        }

        @keyframes emergency-blink {
            0%, 50% { opacity: 1; }
            51%, 100% { opacity: 0.7; }
        }

        .click-mode-info {
            background: #eff6ff;
            border: 1px solid #bfdbfe;
            border-radius: 8px;
            padding: 12px;
            margin-bottom: 15px;
            color: #1e40af;
            font-size: 0.9em;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Робот Phoenix - Система моніторингу</h1>

        <div class="connection-status disconnected" id="connectionStatus">
            <div class="status-dot" id="statusDot"></div>
            <span id="statusText">Підключення до робота...</span>
        </div>

        <!-- Панель екстрених попереджень -->
        <div id="emergencyAlert" class="emergency-alert" style="display: none;">
            КРИТИЧНА ПЕРЕШКОДА ВИЯВЛЕНА!
        </div>

        <div class="dashboard">
            <!-- Ліва панель: Статус -->
            <div class="panel">
                <h3>Загальний статус</h3>
                <div class="status-grid">
                    <div class="status-card">
                        <h4>Позиція робота</h4>
                        <div class="status-value"
id="robotPosition">Завантаження...</div>
                    </div>
                    <div class="status-card">
                        <h4>Поточний стан</h4>
                        <div class="status-value"
id="robotState">Завантаження...</div>
                    </div>
                    <div class="status-card">
                        <h4>Найближча перешкода</h4>
                        <div class="status-value" id="closestObstacle">--</div>
                    </div>
                    <div class="status-card">
                        <h4>Активних датчиків</h4>
```

```
                    <div class="status-value" id="activeSensors">0/8</div>
                </div>
            </div>
        </div>

        <!-- Права панель: Датчики -->
        <div class="panel">
            <h3>Датчики відстані</h3>
            <div class="sensor-grid" id="sensorGrid">
            </div>
        </div>
    </div>

    <!-- Управління -->
    <div class="controls">
        <h3>Управління роботом</h3>

        <div style="margin-bottom: 15px;">
            <label>Ціль: X = </label>
            <input type="number" id="goalX" value="3.0" step="0.1">
            <label>Y = </label>
            <input type="number" id="goalY" value="0.0" step="0.1">
            <button class="btn" onclick="sendGoal()">Відправити ціль</button>
            <button class="btn warning"
onclick="clearPreviewGoal()">Очистити</button>
        </div>
        <div>
            <button class="btn success" onclick="setMode('scanning')">Режим
сканування</button>
            <button class="btn" onclick="setMode('navigation')">Режим
навігації</button>
            <button class="btn" onclick="setMode('exploring')">Режим
дослідження</button>
            <button class="btn danger"
onclick="setMode('stop')">Зупинити</button>
        </div>
    </div>

    <!-- Карта -->
    <div id="map-container">
        <canvas id="map-canvas" width="1000" height="500"></canvas>
        <div class="map-info" id="mapInfo">
            Координати: (0.0, 0.0) | Масштаб: 50px/м
        </div>
    </div>

    <!-- Лог подій -->
    <div class="panel">
        <h3>Лог подій</h3>
        <div class="log" id="eventLog">
            <div class="log-entry">
                <span class="log-timestamp">[СИСТЕМА]</span>
                Очікування підключення до робота...
            </div>
        </div>
    </div>
</div>

<script>
    let robotStatus = null;
    let sensorData = null;
    let obstacleData = null;
```

```
        let isConnected = false;
        let previewGoal = null;
        const API_URL = 'http://localhost:8080';

        // Назви датчиків у зрозумілому форматі
        const sensorNames = {
            'ds_front_left': 'Передній лівий',
            'ds_front': 'Передній центр',
            'ds_front_right': 'Передній правий',
            'ds_left': 'Лівий боковий',
            'ds_right': 'Правий боковий',
            'ds_back_left': 'Задній лівий',
            'ds_back_right': 'Задній правий',
            'ds_back': 'Задній центр'
        };

        function log(message, type = 'info') {
            const logEl = document.getElementById('eventLog');
            const timestamp = new Date().toLocaleTimeString();
            const icon = type === 'error' ? 'ПОМИЛКА' : type === 'warning' ?
'УВАГА' : type === 'success' ? 'УСПІХ' : 'ІНФО';

            const entry = document.createElement('div');
            entry.className = 'log-entry';
            entry.innerHTML = `<span class="log-timestamp">[${timestamp}]</span>
${icon}: ${message}`;

            logEl.appendChild(entry);
            logEl.scrollTop = logEl.scrollHeight;

            // Обмежуємо кількість записів
            if (logEl.children.length > 50) {
                logEl.removeChild(logEl.firstChild);
            }
        }

        function updateConnectionStatus(connected) {
            isConnected = connected;
            const statusEl = document.getElementById('connectionStatus');
            const dotEl = document.getElementById('statusDot');
            const textEl = document.getElementById('statusText');

            if (connected) {
                statusEl.className = 'connection-status connected';
                textEl.textContent = 'Робот підключений';
            } else {
                statusEl.className = 'connection-status disconnected';
                textEl.textContent = 'Робот відключений';
            }
        }

        async function fetchRobotStatus() {
            try {
                const response = await fetch(`${API_URL}/api/status`);

                if (response.ok) {
                    robotStatus = await response.json();
                    updateConnectionStatus(true);
                    updateStatusDisplay();
                    return true;
                } else {
                    updateConnectionStatus(false);
```

```
                    return false;
                }
            } catch (error) {
                updateConnectionStatus(false);
                return false;
            }
        }

        async function fetchSensorData() {
            try {
                const response = await fetch(`${API_URL}/api/sensors`);
                if (response.ok) {
                    sensorData = await response.json();
                    updateSensorDisplay();
                }
            } catch (error) {
                console.log('Дані датчиків недоступні');
            }
        }

        async function fetchObstacleData() {
            try {
                const response = await fetch(`${API_URL}/api/obstacles`);
                if (response.ok) {
                    obstacleData = await response.json();
                }
            } catch (error) {
                console.log('Дані перешкод недоступні');
            }
        }

        function updateStatusDisplay() {
            if (!robotStatus) return;

            // Позиція
            const pos = robotStatus.position;
            document.getElementById('robotPosition').textContent =
                `(${pos[0].toFixed(2)}, ${pos[1].toFixed(2)})`;

            // Стан
            document.getElementById('robotState').textContent = robotStatus.state;

            // Інформація про датчики
            if (robotStatus.sensor_stats) {
                const stats = robotStatus.sensor_stats;
                document.getElementById('activeSensors').textContent =
                    `${stats.active_sensors}/${stats.total_sensors}`;

                if (stats.closest_obstacle && stats.closest_obstacle.distance < 10) {
                    document.getElementById('closestObstacle').textContent =
                        `${stats.closest_obstacle.distance.toFixed(2)}м`;
                } else {
                    document.getElementById('closestObstacle').textContent = 'Немає';
                }

                // Екстрене попередження
                const emergency = document.getElementById('emergencyAlert');
                if (stats.closest_obstacle && stats.closest_obstacle.distance < 0.5)
{
                    emergency.style.display = 'block';
                    log(`КРИТИЧНА ПЕРЕШКОДА на відстані
${stats.closest_obstacle.distance.toFixed(2)}м!`, 'error');
```

```
                } else {
                    emergency.style.display = 'none';
                }
            }

            drawMap();
        }

        function updateSensorDisplay() {
            if (!sensorData || !sensorData.sensors) return;

            const grid = document.getElementById('sensorGrid');
            grid.innerHTML = '';

            for (const [sensorId, data] of Object.entries(sensorData.sensors)) {
                const card = document.createElement('div');
                card.className = `sensor-card ${data.status}`;

                const statusClass = `status-${data.status}`;
                const statusText = data.status === 'danger' ? 'НЕБЕЗПЕЧНО' :
                                   data.status === 'warning' ? 'УВАГА' : 'БЕЗПЕЧНО';

                card.innerHTML = `
                    <div class="sensor-name">${sensorNames[sensorId] ||
sensorId}</div>
                    <div class="sensor-distance">${data.distance.toFixed(2)}м</div>
                    <div class="sensor-status ${statusClass}">${statusText}</div>
                `;

                grid.appendChild(card);
            }
        }

        function getMapCoordinates() {
            const canvas = document.getElementById('map-canvas');
            const centerX = canvas.width / 2;
            const centerY = canvas.height / 2;
            const scale = 50; // пікселів на метр

            return { centerX, centerY, scale };
        }

        function pixelsToWorldCoords(pixelX, pixelY) {
            const { centerX, centerY, scale } = getMapCoordinates();
            const worldX = (pixelX - centerX) / scale;
            const worldY = -(pixelY - centerY) / scale; // Y інвертований
            return { x: worldX, y: worldY };
        }

        function worldCoordsToPixels(worldX, worldY) {
            const { centerX, centerY, scale } = getMapCoordinates();
            const pixelX = centerX + worldX * scale;
            const pixelY = centerY - worldY * scale; // Y інвертований
            return { x: pixelX, y: pixelY };
        }

        function drawMap() {
            const canvas = document.getElementById('map-canvas');
            const ctx = canvas.getContext('2d');

            // Очищення
            ctx.clearRect(0, 0, canvas.width, canvas.height);
```

```
            const { centerX, centerY, scale } = getMapCoordinates();

            // Малюємо перешкоди якщо є дані
            if (obstacleData && obstacleData.obstacles) {
                for (const [key, data] of Object.entries(obstacleData.obstacles)) {
                    if (data.probability > 0.5) {
                        const x = centerX + data.world_position[0] * scale;
                        const y = centerY - data.world_position[1] * scale;

                        ctx.fillStyle = `rgba(239, 68, 68, ${data.probability})`;
                        ctx.fillRect(x - 5, y - 5, 10, 10);
                    }
                }
            }

            // Малюємо робота якщо є дані
            if (robotStatus && robotStatus.position) {
                const pos = robotStatus.position;
                const x = centerX + pos[0] * scale;
                const y = centerY - pos[1] * scale; // Y інвертований
                const angle = pos[2];

                // Тіло робота
                ctx.fillStyle = '#3b82f6';
                ctx.beginPath();
                ctx.arc(x, y, 15, 0, 2 * Math.PI);
                ctx.fill();

                // Напрямок робота
                ctx.strokeStyle = '#1e40af';
                ctx.lineWidth = 4;
                ctx.beginPath();
                ctx.moveTo(x, y);
                ctx.lineTo(x + Math.cos(angle) * 25, y - Math.sin(angle) * 25);
                ctx.stroke();

                // Малюємо промені датчиків
                if (sensorData && sensorData.sensors) {
                    const sensorAngles = {
                        'ds_front_left': 28.6,
                        'ds_front': 0,
                        'ds_front_right': -28.6,
                        'ds_left': 90,
                        'ds_right': -90,
                        'ds_back_left': 150,
                        'ds_back_right': -150,
                        'ds_back': 180
                    };

                    for (const [sensorId, data] of
Object.entries(sensorData.sensors)) {
                        const sensorAngle = angle + (sensorAngles[sensorId] || 0) *
Math.PI / 180;
                        const endX = x + Math.cos(sensorAngle) * data.distance *
scale;
                        const endY = y - Math.sin(sensorAngle) * data.distance *
scale;

                        ctx.strokeStyle = data.status === 'danger' ? '#ef4444' :
                                          data.status === 'warning' ? '#f59e0b' :
'#10b981';
```

```
                    ctx.lineWidth = 2;
                    ctx.setLineDash([5, 5]);
                    ctx.beginPath();
                    ctx.moveTo(x, y);
                    ctx.lineTo(endX, endY);
                    ctx.stroke();
                    ctx.setLineDash([]);

                    // Точка виявлення
                    ctx.fillStyle = ctx.strokeStyle;
                    ctx.beginPath();
                    ctx.arc(endX, endY, 3, 0, 2 * Math.PI);
                    ctx.fill();
                }
            }


            ctx.fillStyle = '#1f2937';
            ctx.font = '14px Arial';
            ctx.textAlign = 'center';
            ctx.fillText('Phoenix', x, y + 35);
        }


        if (robotStatus && robotStatus.goal) {
            const goal = robotStatus.goal;
            const x = centerX + goal[0] * scale;
            const y = centerY - goal[1] * scale;

            ctx.fillStyle = '#10b981';
            ctx.beginPath();
            ctx.arc(x, y, 10, 0, 2 * Math.PI);
            ctx.fill();

            // Хрест
            ctx.strokeStyle = '#fff';
            ctx.lineWidth = 3;
            ctx.beginPath();
            ctx.moveTo(x - 6, y);
            ctx.lineTo(x + 6, y);
            ctx.moveTo(x, y - 6);
            ctx.lineTo(x, y + 6);
            ctx.stroke();

            ctx.fillStyle = '#1f2937';
            ctx.font = '12px Arial';
            ctx.textAlign = 'center';
            ctx.fillText('Ціль', x, y + 25);
        }

        // Малюємо прев'ю цілі (якщо є)
        if (previewGoal) {
            const { x: pixelX, y: pixelY } = worldCoordsToPixels(previewGoal.x,
previewGoal.y);

            ctx.fillStyle = 'rgba(245, 158, 11, 0.6)';
            ctx.strokeStyle = '#f59e0b';
            ctx.lineWidth = 2;
            ctx.beginPath();
            ctx.arc(pixelX, pixelY, 12, 0, 2 * Math.PI);
            ctx.fill();
            ctx.stroke();
```

```
            ctx.strokeStyle = '#fff';
            ctx.lineWidth = 2;
            ctx.beginPath();
            ctx.moveTo(pixelX - 8, pixelY);
            ctx.lineTo(pixelX + 8, pixelY);
            ctx.moveTo(pixelX, pixelY - 8);
            ctx.lineTo(pixelX, pixelY + 8);
            ctx.stroke();

            ctx.fillStyle = '#1f2937';
            ctx.font = '11px Arial';
            ctx.textAlign = 'center';
            ctx.fillText('Прев\'ю', pixelX, pixelY + 28);
        }


        ctx.strokeStyle = '#e5e7eb';
        ctx.lineWidth = 1;

        // Вертикальні лінії
        for (let i = -10; i <= 10; i++) {
            const x = centerX + i * scale;
            ctx.beginPath();
            ctx.moveTo(x, 0);
            ctx.lineTo(x, canvas.height);
            ctx.stroke();
        }

        // Горизонтальні лінії
        for (let i = -5; i <= 5; i++) {
            const y = centerY + i * scale;
            ctx.beginPath();
            ctx.moveTo(0, y);
            ctx.lineTo(canvas.width, y);
            ctx.stroke();
        }

        // Осі
        ctx.strokeStyle = '#6b7280';
        ctx.lineWidth = 2;
        ctx.beginPath();
        ctx.moveTo(centerX, 0);
        ctx.lineTo(centerX, canvas.height);
        ctx.moveTo(0, centerY);
        ctx.lineTo(canvas.width, centerY);
        ctx.stroke();
    }


    function getCanvasMousePos(canvas, event) {
        const rect = canvas.getBoundingClientRect();
        const scaleX = canvas.width / rect.width;
        const scaleY = canvas.height / rect.height;

        return {
            x: (event.clientX - rect.left) * scaleX,
            y: (event.clientY - rect.top) * scaleY
        };
    }
```

```
        // Обробник кліку по карті
        function setupMapClickHandler() {
            const canvas = document.getElementById('map-canvas');
            const mapInfo = document.getElementById('mapInfo');

            // Оновлення інформації про координати при русі миші
            canvas.addEventListener('mousemove', (event) => {
                const mousePos = getCanvasMousePos(canvas, event);
                const worldCoords = pixelsToWorldCoords(mousePos.x, mousePos.y);

                mapInfo.textContent = `Координати: (${worldCoords.x.toFixed(2)},
${worldCoords.y.toFixed(2)}) | Масштаб: 50px/м`;
            });

            // Клік по карті для встановлення цілі
            canvas.addEventListener('click', (event) => {
                const mousePos = getCanvasMousePos(canvas, event);
                const worldCoords = pixelsToWorldCoords(mousePos.x, mousePos.y);

                // Встановлюємо прев'ю цілі
                previewGoal = { x: worldCoords.x, y: worldCoords.y };

                // Оновлюємо поля вводу
                document.getElementById('goalX').value = worldCoords.x.toFixed(2);
                document.getElementById('goalY').value = worldCoords.y.toFixed(2);

                // Перемальовуємо карту
                drawMap();

                log(`Встановлено прев'ю цілі: (${worldCoords.x.toFixed(2)},
${worldCoords.y.toFixed(2)})`, 'info');
            });
        }

        function clearPreviewGoal() {
            previewGoal = null;
            drawMap();
            log('Прев\'ю цілі очищено', 'info');
        }

        async function sendGoal() {
            const x = parseFloat(document.getElementById('goalX').value);
            const y = parseFloat(document.getElementById('goalY').value);

            if (isNaN(x) || isNaN(y)) {
                log('Помилка: неправильні координати цілі', 'error');
                return;
            }

            try {
                const response = await fetch(`${API_URL}/api/set_goal`, {
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify({ x, y })
                });

                const result = await response.json();

                if (response.ok) {
                    log(`Ціль (${x}, ${y}) відправлена`, 'success');
                    previewGoal = null; // Очищаємо прев'ю після відправки
                } else {
```

```
                    log(`Помилка відправки цілі: ${result.message}`, 'error');
                }
            } catch (error) {
                log(`Помилка з'єднання: ${error.message}`, 'error');
            }
        }

        async function setMode(mode) {
            try {
                const response = await fetch(`${API_URL}/api/set_mode`, {
                    method: 'POST',
                    headers: { 'Content-Type': 'application/json' },
                    body: JSON.stringify({ mode })
                });

                const result = await response.json();

                if (response.ok) {
                    log(`Режим '${mode}' встановлено`, 'success');
                } else {
                    log(`Помилка встановлення режиму: ${result.message}`, 'error');
                }
            } catch (error) {
                log(`Помилка з'єднання: ${error.message}`, 'error');
            }
        }

        // Автооновлення кожні 1 секунду
        setInterval(async () => {
            await fetchRobotStatus();
            await fetchSensorData();
            await fetchObstacleData();
        }, 1000);

        // Первинне завантаження
        fetchRobotStatus();
        fetchSensorData();
        fetchObstacleData();

        // Ініціалізація
        setupMapClickHandler();
        drawMap();

        log('Система моніторингу запущена', 'success');
    </script>
</body>
</html>
```

ДОДАТОК Д
Демонстраційний матеріал

№ Документа	Позначення	Найменування	Дод. відомості
		Текстові документи	
1	ГЮИК.ХХХХХХ.019.ПЗ	Пояснювальна записка	
		Додаткові матеріали	
2		Аробація результатів	
		роботи	
3		Програмний код	
		основного контролера	
		webots	
3		Програмний код	
		контролера з API для	
		роботи із командний	
		центром	
4		Програмний код веб	
		додатку із телеметрією	

					ГЮИК.ХХХХХХ.019.ВД			
Змін.	Арк	Номер докум.	Підпис	Дата				
Розроб.		Бєлік М.С			Розроблення конструкції рухомої платформи для робота- промоутера Відомість кваліфікаційної роботи	Літера	Аркуш	Аркушів
Перевір.		Сичова О.				Н	1	1
						Кафедра КІТАР ХНУРЕ		
Н. контр.		Демська Н.П.						
Затв.		Невлюдов І.Ш.						