ДОДАТОК А

Графічний матеріал кваліфікаційної роботи

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

КАФЕДРА ЕЛЕКТРОННИХ ОБЧИСЛЮВАЛЬНИХ МАШИН

КВАЛІФІКАЦІЙНА РОБОТА

«Модель управління паркуванням мобільного роботу на основі даних від датчика відстані»

Студент гр. СПм-23-3 Ілларіонов М.Ю.

Керівник проф. Каргін А.О.

Харків 2025

Мета та завдання кваліфікаційної роботи

Мета кваліфікаційної роботи: розробка, програмна реалізація та експериментальне дослідження інтелектуальної системи автономного паркування мобільного робота, що використовує дані від ультразвукових сенсорів для виявлення перешкод, оцінки придатності вільного простору та точного позиціонування в межах заданого паркувального місця.

Завдання:

- провести аналіз науково-технічних підходів до автоматизованого паркування роботизованих платформ, з акцентом на сенсорне забезпечення та алгоритми керування;
- розробити просторову модель мобільного робота із заданими конструктивними параметрами, включаючи компонування сенсорів, елементів живлення та виконавчих пристроїв;
- сформувати програмну архітектуру системи управління, яка забезпечує обробку даних ультразвукових сенсорів та генерує рішення щодо придатного паркувального місця;
- реалізувати алгоритм виявлення перешкод, розпізнавання вільного простору, планування траєкторії та контролю переміщення мобільного робота;
- виконати моделювання процесу автономного паркування в середовищі MATLAB/Simulink з урахуванням перешкод та обмеженого простору;
- провести візуалізацію руху, аналіз сенсорних даних та оцінку точності кінцевого позиціонування на основі отриманих результатів симуляції.

Об'єкт дослідження: процес автономного паркування мобільного колісного робота в умовах обмеженого простору з використанням ультразвукових сенсорів для виявлення перешкод і навігаційної орієнтації.

Застосування мобільних роботів з функцією паркування

Складська логістика

Автономні мобільні платформи здійснюють транспортування вантажів між зонами зберігання, сортуванням та відвантаженням. Точне позиціонування біля стелажів і док-станцій в умовах щільної забудови та перешкод.

Ключові вимоги:

- Точність позиціонування ±2-5 см
- Робота в структурованому середовищі
- Уникнення динамічних перешкод
- Інтеграція з WMS системами
- 24/7 режим роботи

Автомобільні системи

Автоматизовані паркувальні системи в автомобілях зменшують залежність від водія, знижують ризики зіткнень та оптимізують використання паркінгового простору в середовищах зі змінною геометрією.

Ключові вимоги:

- Оцінка доступного простору
- Динамічне маневрування
- Безпека пішоходів
- Адаптація до різних умов
- Зручність користувача

Громадські простори

Прибирання територій, доставка товарів, інформаційні послуги. Роботи паркуються у визначених зонах базування для підзарядки, заміни контейнерів або зміни режиму роботи у відкритому середовищі.

Ключові вимоги:

- Адаптація до різних покриттів
- Робота при різному освітленні
- Уникнення тимчасових перешкод
- Взаємодія з людьми
- Стійкість до погодних умов

Спеціальне призначення

Оборонна сфера, рятувальні операції, важкодоступні середовища (тунелі, шахти, колектори). Паркування для фіксації позиції, забезпечення стабільності при зміні режиму роботи або взаємодії з інфраструктурою.

Ключові вимоги:

- Критична точність і надійність
- Робота в екстремальних умовах
- Автономність без GPS
- Стійкість до вібрацій
- Швидке реагування

3

Проблеми енергоживлення в безпровідних сенсорних мережах сенсорних мережах

ОБМЕЖЕНА ЄМНІСТЬ БАТАРЕЇ

ВИСОКИЙ ВПЛИВ

Сенсорні вузли живляться від батарей з обмеженою ємністю. Заміна батарей у віддалених або недоступних місцях може бути складною або дорогою, що створює фундаментальне обмеження для довготривалої роботи мережі.

Основні фактори впливу:

- Типова ємність літієвих батарей: 1000-3000 мАг
- Час роботи без заміни: 6-24 місяці
- Вартість заміни в важкодоступних місцях
- Деградація ємності з часом та температурою

ЕНЕРГОСПОЖИВАННЯ ПРИ ПЕРЕДАЧІ

ВИСОКИЙ ВПЛИВ

Передача даних споживає найбільше енергії в сенсорних вузлах. Далекі відстані та перешкоди збільшують енергетичні витрати на зв'язок, що може швидко виснажити батарею вузла.

Енергетичні характеристики:

- Передача: 50-100 мВт потужності
- Прийом: 30-60 мВт потужності
- Залежність від квадрата відстані
- Втрати через багатопроменеве поширення

НЕЕФЕКТИВНЕ МАРШРУТИЗУВАННЯ

СЕРЕДНІЙ ВПЛИВ

Погано спроектовані алгоритми маршрутизування можуть призвести до швидкого виснаження енергії окремих вузлів через нерівномірний розподіл навантаження в мережі.

Проблеми традиційних підходів:

- Перевантаження вузлів біля базової станції
- Відсутність балансування навантаження
- Неврахування залишкової енергії
- Статичні маршрути без адаптації

РЕЖИМ ОЧІКУВАННЯ

СЕРЕДНІЙ ВПЛИВ

Навіть в режимі очікування вузли споживають енергію для підтримки готовності до прийому сигналів. Неефективне управління режимами роботи призводить до непотрібних витрат енергії.

Енергоспоживання режимів:

- Активний режим: 100% споживання
- Режим очікування: 10-30% споживання
- Глибокий сон: 1-5% споживання
- Час пробудження: 1-10 мс

ВПЛИВ НАВКОЛИШНЬОГО СЕРЕДОВИЩА

СЕРЕДНІЙ ВПЛИВ

Екстремальні температури можуть знизити ефективність батарей та збільшити енергоспоживання компонентів. Температурні коливання впливають на всі аспекти роботи сенсорних вузлів.

Температурні ефекти:

- Зниження ємності при низьких температурах
- Збільшення опору при нагріванні
- Температурна компенсація сенсорів
- Термічний шум у радіокомпонентах

ОБРОБКА ДАНИХ

НИЗЬКИЙ ВПЛИВ

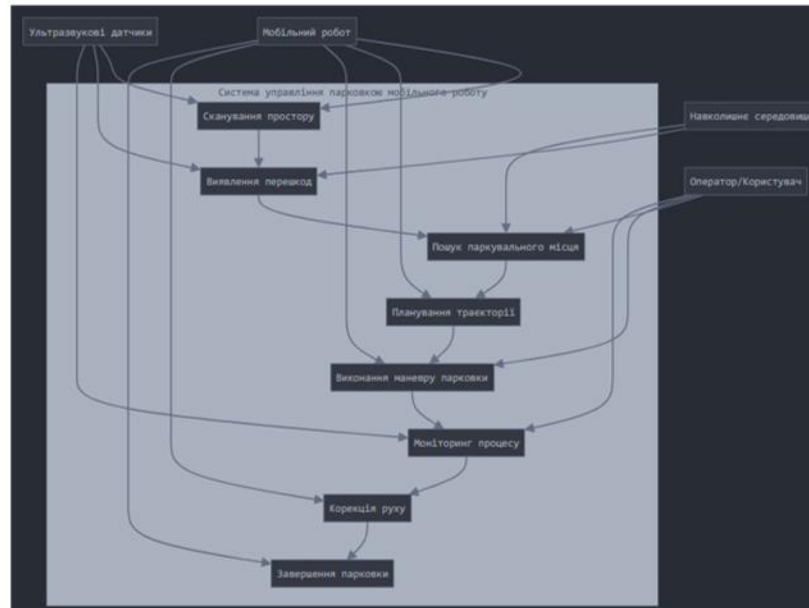
Складні обчислення на вузлах для обробки сенсорних даних також споживають значну кількість енергії, особливо при використанні алгоритмів машинного навчання або криптографії.

Обчислювальні витрати:

- Фільтрація та згладжування сигналів
- Стиснення даних перед передачею
- Криптографічне шифрування
- Локальна агрегація даних

4

Концептуальна модель системи

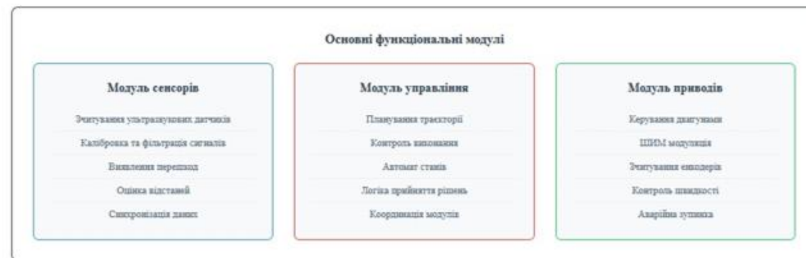


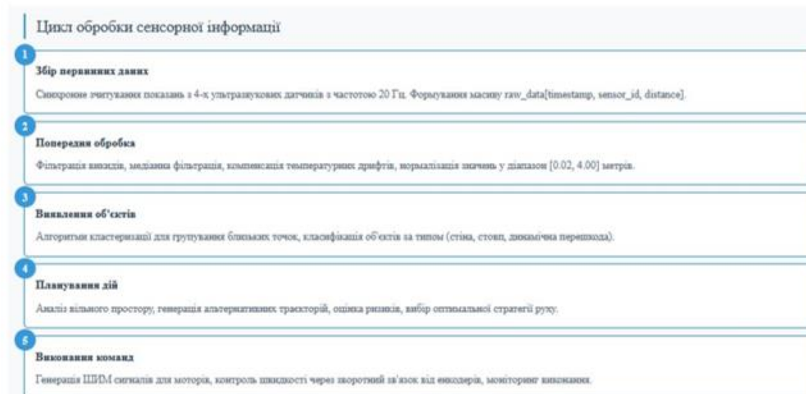
5

Багаторівнева архітектура системи

6

Функціональна декомпозиція системи. Потоки обробки даних





7

Математична модель системи. Схема зв'язків між компонентами. Технічні специфікації пристроїв

Основні кінематичні рівняння:

$$\dot{x} = v \times \cos(\theta)$$
$$\dot{y} = v \times \sin(\theta)$$
$$\dot{\theta} = \omega$$

де: $v = (v_L + v_R) / 2$

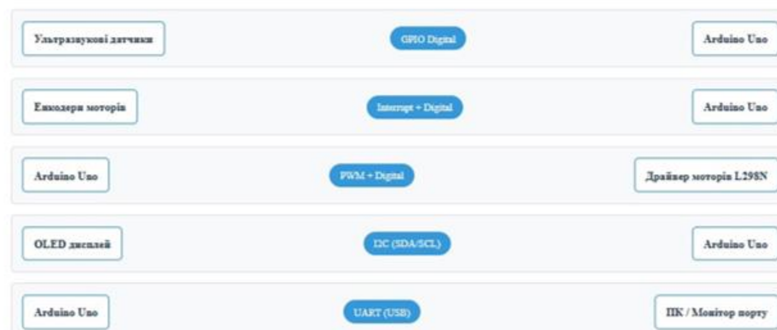
$\omega = (v_R - v_L) / L$

Вимірювання відстані:

$$d = (t_{echo} \times v_{sound}) / 2$$
$$v_{sound} = 331.3 + 0.606 \times T[°C]$$

Інтерфейс	Протокол	Швидкість	Призначення	Конфігурація
GPIO Digital	TTL 5V	До 8 МГц	Ультразвукові датчики	Trigger/Echo піни
PWM	Широтно-імпульсна модуляція	490 Гц	Управління швидкістю моторів	0-255 (8-біт)
I2C	Two-Wire Interface	100/400 кГц	OLED дисплей, сенсори	SDA/SCL + адреса пристрою
UART	Асинхронний серійний	9600-115200 бод	Дебаг, телеметрія	8N1, без контролю потоку
Interrupt	Апаратні переривання	Миттєво	Енкодери, аварійна зупинка	RISING/FALLING edge

Схема зв'язків між компонентами



8

Розподіл обчислювальних ресурсів. Багаторівнева система захисту

Процес	Пріоритет	Частота виконання	Час виконання	Ресурси CPU (%)
Зчитування сенсорів	Високий	50 Гц	2 мс	10%
Обробка даних	Високий	20 Гц	5 мс	15%
Планування траєкторії	Середній	10 Гц	8 мс	20%
Управління моторами	Високий	100 Гц	1 мс	25%
Інтерфейс користувача	Низький	5 Гц	3 мс	5%
Діагностика	Низький	1 Гц	10 мс	10%
Резерв системи	-	-	-	15%

Рівень	Умова спрацювання	Дія системи	Час реакції
Попередження	Відстань < 30 см	Звуковий сигнал, зменшення швидкості	< 100 мс
Критичний	Відстань < 15 см	Аварійна зупинка, блокування руху	< 50 мс
Аварійний	Відстань < 5 см або втрата сенсора	Миттєва зупинка, відключення моторів	< 20 мс
Системний	Втрата зв'язку, перегрів, низький заряд	Повна зупинка, режим відновлення	< 200 мс

9

Алгоритм виявлення паркувального місця

1. **Ініціалізація сканування**

   Встановлення початкових параметрів: швидкість сканування 0.2 м/с, мінімальна довжина місця 1.5×L_robot, пороговий зазор 0.6 м.

2. **Виявлення зазору**

   Моніторинг бічного датчика під час руху. Фіксація початку зазору при distance > threshold_gap протягом мінімум 3 послідовних вимірювань.

3. **Вимірювання довжини**

   Продовження руху з постійною швидкістю до виявлення кінця зазору (distance < threshold_gap). Обчислення gap_length = velocity × time_elapsed.

4. **Оцінка придатності**

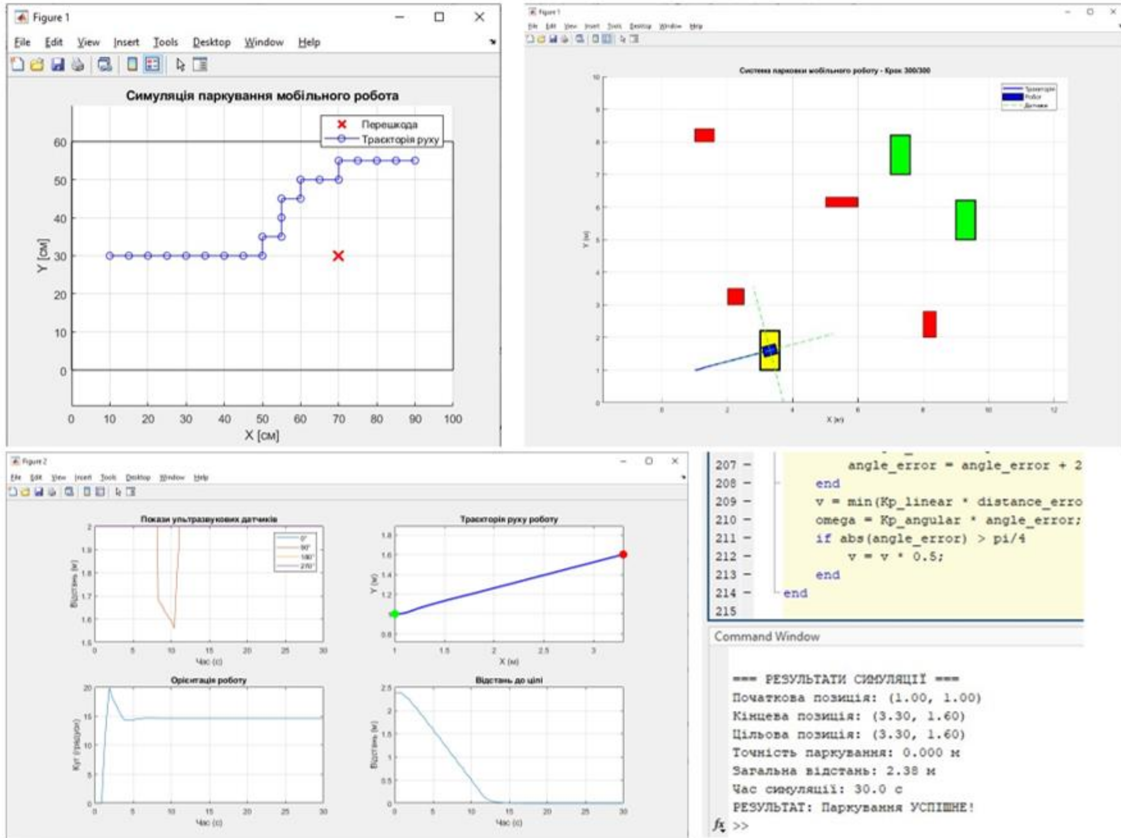
   Перевірка критеріїв: gap_length ≥ min_length, відсутність перешкод у глибині зазору, безпечна відстань від країв.

5. **Збереження координат**

   Фіксація глобальних координат початку та кінця придатного місця, розрахунок цільової точки для початку маневру паркування.

10

Симуляція в Matlab




```
207 -        angle_error = angle_error + 2
208 -      end
209 -      v = min(Kp_linear * distance_erro
210 -      omega = Kp_angular * angle_error;
211 -      if abs(angle_error) > pi/4
212 -          v = v * 0.5;
213 -      end
214 -  end
215
```

11

Просторова модель мобільного роботу

№	Компонент	Модель/Артикул	Кількість	Ціна за одиницю	Загальна вартість	Постачальник
1	Мікроконтролер	Arduino Uno R3	1	$25	$25	Arduino Store
2	Ультразвукові датчики	HC-SR04	4	$2	$8	AliExpress
3	Мотор-редуктори	TT Motor 3-6V	2	$3	$6	Local electronics
4	Драйвер моторів	L298N	1	$3	$3	AliExpress
5	Енкодери	Optical encoder 400p/r	2	$5	$10	Pololu
6	Акумулятор	Li-ion 18650 3.7V 2200mAh	3	$4	$12	Battery store
7	Контролер заряду	TP4056	1	$1	$1	AliExpress
8	Стабілізатор напруги	LM2596 DC-DC	2	$2	$4	AliExpress
9	OLED дисплей	SSD1306 128x64	1	$3	$3	Adafruit
10	LED індикатори	5mm LED RGB	3	$0.5	$1.5	Local electronics
11	Зумер	Passive buzzer 5V	1			

Параметр	Значення	Одиниці	Примітки
Габаритні розміри (Д×Ш×В)	300×200×120	мм	Без врахування виступаючих елементів
Маса платформи	1.2	кг	Повна маса з акумулятором
Колісна база	180	мм	Відстань між центрами коліс
Діаметр коліс	65	мм	Гумові колеса з протектором
Максимальна швидкість	0.5	м/с	При номінальній напрузі
Мінімальний радіус повороту	90	мм	Поворот на місці
Точність позиціонування	±5	мм	В ідеальних умовах
Час автономної роботи	4-6	год	Залежно від інтенсивності використання

12

Просторова модель мобільного роботу

№	Компонент	Модель/Артикул	Кількість	Ціна за одиницю	Загальна вартість	Постачальник
1	Мікроконтролер	Arduino Uno R3	1	$25	$25	Arduino Store
2	Ультразвукові датчики	HC-SR04	4	$2	$8	AliExpress
3	Мотор-редуктори	TT Motor 3-6V	2	$3	$6	Local electronics
4	Драйвер моторів	L298N	1	$3	$3	AliExpress
5	Енкодери	Optical encoder 400p/r	2	$5	$10	Pololu
6	Акумулятор	Li-ion 18650 3.7V 2200mAh	3	$4	$12	Battery store
7	Контролер заряду	TP4056	1	$1	$1	AliExpress
8	Стабілізатор напруги	LM2596 DC-DC	2	$2	$4	AliExpress
9	OLED дисплей	SSD1306 128x64	1	$3	$3	Adafruit
10	LED індикатори	5mm LED RGB	3	$0.5	$1.5	Local electronics
11	Зумер	Passive buzzer 5V	1			

Параметр	Значення	Одиниці	Примітки
Габаритні розміри (Д×Ш×В)	300×200×120	мм	Без врахування виступаючих елементів
Маса платформи	1.2	кг	Повна маса з акумулятором
Колісна база	180	мм	Відстань між центрами коліс
Діаметр коліс	65	мм	Гумові колеса з протектором
Максимальна швидкість	0.5	м/с	При номінальній напрузі
Мінімальний радіус повороту	90	мм	Поворот на місці
Точність позиціонування	±5	мм	В ідеальних умовах
Час автономної роботи	4-6	год	Залежно від інтенсивності використання

12

Діаграма взаємодії компонентів

14

Компоненти програмної частини

DataStructures.h - Основні структури даних

```cpp
// Структура для зберігання даних сенсора
struct SensorReading {
  float distance;            // Відстань в см
  float filteredDistance;  // Відфільтроване значення
  unsigned long timestamp; // Час вимірювання
  bool isValid;            // Валідність показання
  uint8_t sensorId;        // Ідентифікатор датчика
};

// Позиція та орієнтація робота
struct RobotPose {
  float x, y;              // Координати в мм
  float theta;             // Орієнтація в радіанах
  float confidence;        // Достовірність позиції
  unsigned long timestamp;
};

// Команда руху для моторів
struct MotionCommand {
  float leftSpeed;         // Швидкість лівого мотора (-255 до 255)
  float rightSpeed;        // Швидкість правого мотора
  unsigned int duration;   // Тривалість команди в мс
  bool isEmergency;        // Прапорець аварійної команди
};

// Опис паркувального місця
struct ParkingSpace {
  float startX, startY;    // Початок простору
  float endX, endY;        // Кінець простору
  float width, length;     // Розміри простору
  float approachAngle;     // Кут підходу
  bool isOccupied;         // Зайнятість місця
  uint8_t quality;         // Якість місця (0-100)
};

// Конфігурація системи
struct SystemConfig {
  float maxSpeed;          // Максимальна швидкість
  float minParkingGap;     // Мінімальний зазор для паркування
  float safetyDistance;    // Безпечна відстань
  float wheelBase;         // База коліс
  float robotWidth;        // Ширина робота
  float robotLength;       // Довжина робота
};
```

MotorController.h - Інтерфейс управління моторами

```cpp
class MotorController {
private:
  int leftMotorPWM, rightMotorPWM;
  int leftDir1, leftDir2, rightDir1, rightDir2;
  float currentLeftSpeed, currentRightSpeed;
  unsigned long lastEncoderTime;

public:
  MotorController(int lPWM, int rPWM, int lD1, int lD2, int rD1, int rD2);

  void initialize();
  void setSpeed(float leftSpeed, float rightSpeed);
  void moveForward(float speed);
  void moveBackward(float speed);
  void turnLeft(float speed);
  void turnRight(float speed);
  void stop();
  void emergencyStop();

  // PID регулятор швидкості
  void updatePID();
  float calculatePID(float setpoint, float measured, float& prevError, float& integral);
};
```

MathUtils.cpp - Математичні утиліти

```cpp
// Нормалізація кута в діапазон [-π, π]
float normalizeAngle(float angle) {
  while(angle > PI) angle -= 2 * PI;
  while(angle < -PI) angle += 2 * PI;
  return angle;
}

// Медіанний фільтр для сенсорних даних
float medianFilter(float values[], int size) {
  // Сортування масиву методом бульбашки
  for(int i = 0; i < size - 1; i++) {
    for(int j = 0; j < size - i - 1; j++) {
      if(values[j] > values[j + 1]) {
        float temp = values[j];
        values[j] = values[j + 1];
        values[j + 1] = temp;
      }
    }
  }
  return values[size / 2];
}

// Розрахунок відстані між двома точками
float calculateDistance(float x1, float y1, float x2, float y2) {
  float dx = x2 - x1;
  float dy = y2 - y1;
  return sqrt(dx * dx + dy * dy);
}
```

SensorArray.cpp - Реалізація керування датчиками

```cpp
#include "SensorArray.h"

SensorArray::SensorArray() {
  sensors[0] = UltrasonicSensor([illegible], [illegible]);  // Передній
  sensors[1] = UltrasonicSensor([illegible], [illegible]);  // Лівий
  sensors[2] = UltrasonicSensor([illegible], 7);  // Правий
  sensors[3] = UltrasonicSensor([illegible], [illegible]);  // Задній
}

void SensorArray::readAllSensors() {
  for(int i = 0; i < 4; i++) {
    sensorData[i].distance = sensors[i].measureDistance();
    sensorData[i].timestamp = millis();
    sensorData[i].isValid = validateReading(i);

    // Застосування медіанного фільтру
    sensorData[i].filteredDistance = medianFilter(i, sensorData[i].distance);
  }
}

bool SensorArray::validateReading(int sensorIndex) {
  float distance = sensorData[sensorIndex].distance;

  // Перевірка діапазону
  if(distance < [illegible] || distance > [illegible]) return false;

  // Перевірка швидкості зміни
  if(abs(distance - previousReadings[sensorIndex]) > [illegible]) {
    return false;
  }

  previousReadings[sensorIndex] = distance;
  return true;
}
```

ParkingStateMachine.h - Визначення станів та переходів

```cpp
enum ParkingState {
  IDLE,          // Очікування команд
  SCANNING,      // Пошук паркувального місця
  APPROACHING,   // Наближення до цільової позиції
  PARKING,       // Виконання маневру паркування
  PARKED,        // Успішно припарковано
  ERROR_STATE    // Стан помилки
};

class ParkingStateMachine {
private:
  ParkingState currentState;
  unsigned long stateStartTime;
  unsigned long stateTimeout;

public:
  ParkingStateMachine();
  void initialize();
  void update();
  bool transitionTo(ParkingState newState);
  ParkingState getCurrentState();
  void handleTimeout();
  const char* getStateString(ParkingState state);
};
```

15

Етапи тестування системи

Тести компонентів нижнього рівня

Тест 1: UltrasonicSensor.measureDistance() — ПРОЙДЕНО

Кількість вимірювань	Точність	Час відгуку	Успішність
1000	±3.2 мм	38.5 мс	99.8 %

Опис: Перевірка точності вимірювання відстані на еталонних дистанціях 10, 50, 100, 200 см. **Результат:** Всі вимірювання в межах допустимої похибки ±5 мм.

Тест 2: MotorController.setSpeed() — ПРОЙДЕНО

Діапазон швидкостей	Лінійність	Час встановлення	Точність ШІМ
0-255 PWM	$R^2 = 0.987$	150 мс	±1 Гц

Тест 3: PathPlanner.calculateTrajectory() — ПОПЕРЕДЖЕННЯ

Час планування	Успішні плани	Оптимальність	Пам'ять
245 мс	94.2 %	87.5 %	156 байт

Примітка: Час планування перевищує бажаний поріг 200 мс у 8% випадків при складних сценаріях.

Інтеграційний тест	Компоненти	Результат	Час виконання	Примітки
Sensor-Controller Integration	SensorArray + ParkingController	ПРОЙДЕНО	45 мс	Стабільна передача даних
Motor-Safety Integration	MotorController + SafetyManager	ПРОЙДЕНО	12 мс	Аварійна зупинка < 50 мс
Navigation-Motion Integration	NavigationService + MotionService	ПОПЕРЕДЖЕННЯ	89 мс	Невелика затримка при складних маневрах
Display-Controller Integration	DisplayManager + ParkingController	ПРОЙДЕНО	23 мс	Оновлення інформації в реальному часі

16

Тестування системи

Сценарій 1: Базове паркування в відкритому просторі — ПРОЙДЕНО

Кількість спроб	Успішність	Середній час	Точність
25	96% (24/25)	42.3 с	±4.2 см

Умови: Рівна поверхня, відсутність перешкод, паркувальне місце 80×60 см. Критерій успіху: Позиціонування в межах ±5 см від центру за час < 60 с.

Сценарій 2: Паркування з статичними перешкодами — ПРОЙДЕНО

Кількість спроб	Успішність	Середній час	Зіткнення
20	85% (17/20)	67.8 с	0 випадків

Умови: 3-4 статичні перешкоди, мінімальний прохід 45 см. Результат: Всі маневри виконані без зіткнень, 3 невдачі через таймаут.

Сценарій 3: Паркування в обмеженому просторі — ЧАСТКОВО ПРОЙДЕНО

Кількість спроб	Успішність	Середній час	Точність
15	73% (11/15)	89.5 с	±7.8 см

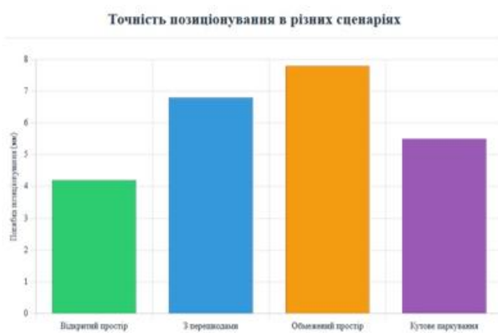
Проблема: Зниження точності при паркуванні в місцях розміром 1.2×довжина робота. Рекомендація: Покращення алгоритму точного позиціонування.

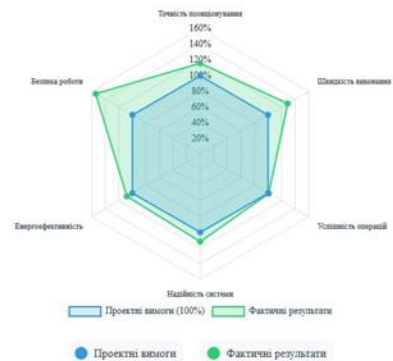
Сценарій 4: Стрес-тест тривалої роботи — ПРОЙДЕНО

Тривалість	Операцій паркування	Відмови	Деградація точності
4.5 год	127	2 випадки	< 5%

Метрика продуктивності	Виміряне значення	Цільове значення	Статус	Коментар
Час циклу основного loop()	47 мс	< 50 мс	✓ ДОБРЕ	Стабільний час виконання
Час зчитування 4 сенсорів	156 мс	< 200 мс	✓ ДОБРЕ	Послідовне опитування
Час планування простої траєкторії	89 мс	< 100 мс	✓ ДОБРЕ	Лінійна та дугова траєкторії
Час планування складної траєкторії	267 мс	< 300 мс	△ ЗАДОВІЛЬНО	З множинними перешкодами
Час аварійної зупинки	23 мс	< 50 мс	✓ ВІДМІННО	Критично важливо
Використання стеку	412 байт	< 500 байт	✓ ДОБРЕ	Резерв для розширення
Фрагментація heap	< 5%	< 10%	✓ ВІДМІННО	Мінімальне використання динамічної пам'яті

Результати





Апробація результатів дослідження

CONTROL MODELS FOR MOBILE ROBOT PARKING USING DISTANCE SENSOR DATA

Abstract. Relevance. The growing demand for autonomous mobile systems capable of independent navigation and parking is driven by several critical factors. Firstly, the rapid robotization in logistics, security, delivery, and service industries necessitates reliable mechanisms for precise positioning of mobile platforms in spatially constrained environments. Secondly, in the context of autonomous vehicle development, the issue of automatic parking becomes a priority for enhancing safety, reducing energy consumption, and minimizing human involvement in control processes. Currently, a significant number of studies focus on the implementation of automatic parking systems; however, most of them either rely on high-cost sensors (such as LiDARs or deep-learning-based cameras) or fail to ensure the required accuracy under dynamic or unfamiliar environmental conditions. Against this backdrop, the use of ultrasonic sensors represents an effective alternative, enabling a necessary level of adaptability and sensitivity while maintaining low system cost. The relevance of this research is further reinforced by the need to develop a universal control model that is scalable, adaptive, and easily integrable into various types of mobile platforms. This work focuses not only on the theoretical formulation of the control model but also on its experimental validation using data from ultrasonic sensors that reflect the physical environment in real time. Therefore, the development of a mobile robot parking control model based on ultrasonic sensors is a timely and important task that combines scientific novelty with practical significance for the advancement of autonomous systems. **The object of research**. A parking control system for a mobile robot that operates based on data obtained from ultrasonic distance sensors. This system comprises both hardware components, such as ultrasonic sensors, actuators, and controllers, and software that implements algorithms for environmental analysis, decision-making related to parking maneuvers, and motion control. **Purpose of the article.** This article presents a comprehensive review of contemporary models for mobile robot parking control based on distance sensor data. The objective is to identify and critically evaluate effective approaches to sensor integration, control algorithm design, and architectural implementation of such systems. Particular attention is given to analyzing their applicability in real-world environments, with the aim of outlining development prospects that enhance system accuracy, reliability, and adaptability under dynamic and constrained conditions. **Research results.** As a result of the conducted review, it has been established that modern mobile robot parking control systems encompass a wide range of modeling approaches, varying in both mathematical complexity and sensor configurations. The analysis reveals that the choice of control models is directly influenced by the availability of computational resources, the robot's chassis type, and the nature of the operational environment. Particular attention is given to the comparative assessment of sensors, with ultrasonic sensors remaining dominant in short-range positioning systems due to their low cost, ease of integration, and reliability in controlled conditions. Conversely, LiDAR sensors have demonstrated superior accuracy and spatial resolution, although they present higher implementation and maintenance complexity. Cameras and infrared sensors are regarded as supplementary data sources, functioning effectively only within well-defined conditions and with appropriate software support. The findings of the review confirm that an effective parking control system for mobile robots relies on a holistic approach that integrates sensor selection, control model design, algorithmic implementation, and system architecture. Such integration enables high accuracy and operational reliability even in complex, dynamic, or constrained environments. **Conclusions.** Effective mobile robot parking control is based on the integration of reliable sensor systems, particularly ultrasonic sensors and adaptive decision-making algorithms. Ultrasonic sensors remain the most suitable option for low-cost and simple systems, whereas hybrid approaches involving LiDAR or camera-based solutions offer higher precision. Among the control algorithms, finite state machines, fuzzy logic, and machine learning methods have demonstrated the greatest effectiveness. The most optimal system architecture is modular, with a clear separation between sensing, computation, and actuation layers, which ensures adaptability, accuracy, and operational stability under real-world conditions.

Keywords: mobile robot, autonomous parking, ultrasonic sensors, sensor system, decision-making algorithms, finite state machine, fuzzy logic, machine learning, motion control, navigation system.

Introduction

In the context of rapid advancements in autonomous mobility, the issue of efficient parking control for mobile robots has become increasingly relevant. Across multiple industries – from warehouse logistics to autonomous ... sors offer a simple, low-cost, and reliable means of detecting obstacles and measuring distances to them, enabling the implementation of adaptive maneuvering algorithms. The primary challenge lies in developing an appropriate mathematical model and software-hardware implementation that can effectively interpret sensor

control, and the evaluation of autonomous navigation system accuracy. The first two references are particularly foundational, as they systematize key methods and examine the potential of ultrasonic systems in real-world applications, especially in environments with spatial or resource constraints.

In [1], the authors analyze modern sensor systems used for obstacle detection in mobile robotics. The study focuses on comparing ultrasonic sensors, LiDAR, infrared sensors, and cameras. Ultrasonic sensors are identified as the most effective for short-range navigation, especially when computational resources or budget are limited. The authors conclude that ultrasonic sensors offer the lowest energy consumption and cost, and high indoor reliability, but can be affected by surfaces that strongly absorb sound.

The review in [2] explores a broad range of applications for ultrasonic sensors, including robotics, automotive ADAS systems, security systems, and smart cities. In the context of robotics and autonomous parking, ultrasonic sensors are highlighted as the best solution for short-distance measurement and object detection. The study also notes that combining them with other sensors (infrared, magnetic, cameras) significantly improves system stability and adaptability in complex environments.

Studies [3] and [4] focus on emerging technologies that enhance traditional applications of ultrasonic sensors in mobile robotics, particularly in the domain of autonomous parking. These works highlight the transition from simple two-dimensional systems to volumetric sensing approaches, which open new possibilities for achieving full autonomy in real-world-like environments.

Study [3] addresses a new generation of ultrasonic sensors designed to provide three-dimensional coverage of the space surrounding a mobile robot. Specifically, it explores Acoustic Detection and Ranging (ADAR) systems, which enable 360-degree spatial perception without the need for rotating mechanisms or complex optical setups. One of the key advantages of these sensors lies in their significantly lower cost compared to traditional LiDAR solutions: a single ADAR module is priced at approximately $1,000, whereas a conventional 2D LiDAR system may cost $4,000 or more. In addition to affordability, the authors emphasize safety advancements – ADAR sensors are certified under IEC 61508 (SIL2), making them suitable for use in industrial and

bot developers. It presents the advantages of a new acoustic radar architecture that relies not on the classic conical emission diagram, but on analytical reconstruction of the acoustic wave in space. By minimizing mechanical components and increasing the data refresh rate (up to 100 Hz in scanning mode), the system significantly improves obstacle detection stability during rapid maneuvers. The authors also emphasize the economic impact: a configuration using four ADAR modules effectively replaces two full-scale LiDAR units, resulting in an overall cost reduction of 60–80%. Practical examples include configurations for logistics-class robots, where the parking system operates based on a 3D ultrasonic map of the environment. Unlike traditional systems that require prior map alignment, the new system performs local spatial reconstruction and does not rely on external references. Consequently, this publication not only confirms the potential of 3D ultrasonic sensors but also serves as a technical foundation for their integration into next-generation mobile robot parking control solutions.

In [5], the authors propose a fundamentally novel approach to sensor integration: instead of mounting ultrasonic sensors on the robot body or masts, the sensing elements are embedded directly into the wheels of the mobile platform. This configuration creates an acoustic canalization system that combines contact-based and contactless sensitivity. The paper presents a technical description of the sensor architecture: a piezoelectric element embedded in the wheel rim generates acoustic waves that propagate through the tire and reflect from the surface with which the wheel makes contact. By analyzing the acoustic response, the system can determine the type of surface, detect micro-obstacles, and even localize the initial point of contact with a wall or other object. Experimental results demonstrate that the robot was able to accurately detect changes in ground material without the assistance of cameras or conventional sensors. This opens new prospects for parking systems operating under low-light, dusty, or confined conditions, where ultrasonic reflection from walls offers advantages over optical systems. The authors emphasize that such tactile ultrasonic sensing could serve as a backup system for registering real contact with obstacles during parking maneuvers, significantly reducing the risk of mechanical damage to the platform.

In [6], the authors investigate the deployment of multiple ultrasonic localization systems in a single envi-

A. Huk, V. Diachenko, M. Illarionov, Y. Titova Control models for mobile robot parking using distance sensor data. Системи управління, навігації та зв'язку, вип.4. Полтава, 2025.

Висновки

У результаті проведеної кваліфікаційної роботи було досягнуто поставлену мету – розроблено, програмно реалізовано та експериментально досліджено інтелектуальну систему автономного паркування мобільного робота, яка забезпечує точне та безпечне позиціонування в межах обмеженого простору. Запропонований підхід базується на використанні ультразвукових сенсорів для отримання інформації про навколишнє середовище, модулів аналізу перешкод та оцінки паркувальних місць, а також алгоритмів пропорційного керування рухом.

Побудована багаторівнева архітектура програмного забезпечення забезпечує модульність, розширюваність та надійність системи. Було реалізовано окремі компоненти керування сенсорами, привідною частиною, навігацією, обробкою траєкторій, виведенням інформації та забезпеченням безпеки. Моделювання в середовищі MATLAB дозволило протестувати логіку прийняття рішень, поведінку віртуального робота, а також провести аналіз траєкторії, змін орієнтації, сенсорних відгуків і точності паркування. Результати підтверджують високу точність досягнення цільової позиції та стійкість системи до типових похибок.

Апартно-програмна платформа на базі Arduino була доповнена компонентами для сенсорного сприйняття, обробки сигналів і приводу, що забезпечило реальну можливість апробації в лабораторних умовах. Проведені модульні, інтеграційні, системні та приймальні тести підтвердили коректну взаємодію всіх частин системи, виявили окремі слабкі місця, які можуть бути покращені в майбутньому, зокрема щодо оптимізації навігаційних розрахунків у складних сценаріях.

Загалом, результати роботи свідчать про доцільність застосування сенсорного аналізу, програмного моделювання та пропорційного регулювання для задач автоматизованого паркування мобільних роботів. Запропонована система може бути використана як прототип для подальших розробок автономних роботизованих платформ, здатних адаптуватися до змін середовища та приймати рішення на основі локальних даних у режимі реального часу.

ДОДАТОК Б

Програмний код

Б.1 Код ParkingSystemSimulation.m

```
% ParkingSystemSimulation.m
% Моделювання системи паркування мобільного робота з
ультразвуковими сенсорами

clear;
clc;

% Параметри паркувального середовища
parkingLength = 100;  % Довжина паркувального простору (см)
parkingWidth = 60;    % Ширина простору (см)
obstaclePos = [70, 30]; % Координати перешкоди [x, y]

% Початкове положення робота
robotPos = [10, 30];  % [x, y]
robotAngle = 0;       % Кут орієнтації (градуси)

% Параметри сенсора
ultrasonicRange = 30;  % Радіус дії сенсора (см)
sensorAccuracy = 0.9;

% Алгоритм паркування
path = robotPos;
for t = 1:50
    % Імітація сенсорних даних
    d = checkUltrasonic(robotPos, obstaclePos, ultrasonicRange);

    % Проста логіка ухилення від перешкоди
    if d > 0 && d < 20
        robotPos(2) = robotPos(2) + 5; % зміна траєкторії вбік
    else
        robotPos(1) = robotPos(1) + 5; % рух вперед
    end

    % Запис шляху
    path = [path; robotPos];

    % Умова завершення
    if robotPos(1) >= parkingLength - 10
        break;
    end
end
```

```
% Візуалізація
figure;
hold on;
rectangle('Position',[0, 0, parkingLength,
parkingWidth],'EdgeColor','k');
plot(obstaclePos(1), obstaclePos(2), 'rx', 'MarkerSize', 12,
'LineWidth', 2);
plot(path(:,1), path(:,2), 'b-o');
title('Симуляція паркування мобільного робота');
xlabel('X [см]');
ylabel('Y [см]');
legend('Перешкода', 'Траєкторія руху');
grid on;
axis equal;

% === Локальна функція в кінці скрипта ===
function distance = checkUltrasonic(pos, obstaclePos, range)
    dx = obstaclePos(1) - pos(1);
    dy = obstaclePos(2) - pos(2);
    d = sqrt(dx^2 + dy^2);
    if d <= range
        distance = d + randn * 0.5; % додати шум
    else
        distance = -1;  % нічого не виявлено
    end
end
```

Б.2 PSS_v1.m

```
function PSS_v1.m
    % Моделювання системи управління парковкою мобільного роботу
з ультразвуковими датчиками

    clear all; close all; clc;

    %% Параметри системи
    robot_width = 0.3;
    robot_length = 0.4;
    max_speed = 0.5;
    wheel_base = 0.25;

    num_sensors = 4;
    sensor_range = 2.0;
    sensor_accuracy = 0.01;
    sensor_angles = [0, 90, 180, 270];

    field_size = [10, 10];
    parking_spot_size = [0.6, 1.2];

    %% Ініціалізація середовища
    obstacles = [2, 3, 0.5, 0.5; 5, 6, 1.0, 0.3; 8, 2, 0.4, 0.8;
1, 8, 0.6, 0.4];
```

```
    parking_spots = [7, 7, parking_spot_size(1),
parking_spot_size(2);
                     3, 1, parking_spot_size(1),
parking_spot_size(2);
                     9, 5, parking_spot_size(1),
parking_spot_size(2)];
    robot_pos = [1, 1, 0];

    %% Симуляція
    dt = 0.1;
    simulation_time = 30;
    steps = simulation_time / dt;

    robot_trajectory = zeros(steps, 3);
    sensor_readings = zeros(steps, num_sensors);
    time_vector = 0:dt:(steps-1)*dt;

    target_spot = find_parking_spot(robot_pos, parking_spots,
obstacles, [robot_width, robot_length]);
    if isempty(target_spot)
        error('Не знайдено підходящого паркувального місця');
    end

    target_pos = [target_spot(1) + target_spot(3)/2,
target_spot(2) + target_spot(4)/2];
    planned_path = plan_path(robot_pos, target_pos, obstacles,
field_size);

    figure('Position', [100, 100, 1200, 800]);
    for step = 1:steps
        distances = simulate_ultrasonic_sensors(robot_pos,
obstacles, sensor_angles, sensor_range);
        sensor_readings(step, :) = distances;

        if step < steps * 0.8
            current_target = planned_path(min(ceil(step/10),
size(planned_path,1)), :);
            [v, omega] = motion_controller(robot_pos,
[current_target, 0], max_speed);
        else
            [v, omega] = motion_controller(robot_pos,
[target_pos, 0], max_speed * 0.3);
        end

        robot_pos(1) = robot_pos(1) + v * cos(robot_pos(3)) *
dt;
        robot_pos(2) = robot_pos(2) + v * sin(robot_pos(3)) *
dt;
        robot_pos(3) = robot_pos(3) + omega * dt;
        robot_trajectory(step, :) = robot_pos;

        if mod(step, 10) == 0
            clf;
```

```
            hold on;
            axis([0 field_size(1) 0 field_size(2)]);
            axis equal;
            grid on;

            for i = 1:size(obstacles, 1)
                rectangle('Position', obstacles(i,:),
'FaceColor', 'red', 'EdgeColor', 'black');
            end
            for i = 1:size(parking_spots, 1)
                rectangle('Position', parking_spots(i,:),
'FaceColor', 'green', 'EdgeColor', 'black', 'LineWidth', 2);
            end
            rectangle('Position', target_spot, 'FaceColor',
'yellow', 'EdgeColor', 'black', 'LineWidth', 3);
            plot(robot_trajectory(1:step, 1),
robot_trajectory(1:step, 2), 'b-', 'LineWidth', 2);

            robot_x = robot_pos(1) + [-robot_length/2,
robot_length/2, robot_length/2, -robot_length/2, -
robot_length/2] * cos(robot_pos(3)) - ...
                      [-robot_width/2, -robot_width/2,
robot_width/2, robot_width/2, -robot_width/2] *
sin(robot_pos(3));
            robot_y = robot_pos(2) + [-robot_length/2,
robot_length/2, robot_length/2, -robot_length/2, -
robot_length/2] * sin(robot_pos(3)) + ...
                      [-robot_width/2, -robot_width/2,
robot_width/2, robot_width/2, -robot_width/2] *
cos(robot_pos(3));
            fill(robot_x, robot_y, 'blue', 'EdgeColor', 'black',
'LineWidth', 2);

            for i = 1:num_sensors
                sensor_angle = deg2rad(sensor_angles(i)) +
robot_pos(3);
                sensor_end_x = robot_pos(1) + distances(i) *
cos(sensor_angle);
                sensor_end_y = robot_pos(2) + distances(i) *
sin(sensor_angle);
                plot([robot_pos(1), sensor_end_x],
[robot_pos(2), sensor_end_y], 'g--', 'LineWidth', 1);
            end
            title(sprintf('Система парковки мобільного роботу -
Крок %d/%d', step, steps));
            xlabel('X (м)'); ylabel('Y (м)');
            legend('Траєкторія', 'Робот', 'Датчики', 'Location',
'best');
            drawnow;
            pause(0.05);
        end
    end
```

```
    %% Аналіз
    figure('Position', [100, 100, 1200, 600]);
    subplot(2, 2, 1);
    plot(time_vector, sensor_readings);
    title('Покази ультразвукових датчиків');
    xlabel('Час (с)'); ylabel('Відстань (м)');
    legend('0°', '90°', '180°', '270°'); grid on;

    subplot(2, 2, 2);
    plot(robot_trajectory(:, 1), robot_trajectory(:, 2), 'b-',
'LineWidth', 2); hold on;
    plot(robot_trajectory(1, 1), robot_trajectory(1, 2), 'go',
'MarkerSize', 10, 'MarkerFaceColor', 'green');
    plot(robot_trajectory(end, 1), robot_trajectory(end, 2),
'ro', 'MarkerSize', 10, 'MarkerFaceColor', 'red');
    title('Траєкторія руху роботу'); xlabel('X (м)'); ylabel('Y
(м)'); grid on; axis equal;

    subplot(2, 2, 3);
    plot(time_vector, rad2deg(robot_trajectory(:, 3)));
    title('Орієнтація роботу'); xlabel('Час (с)'); ylabel('Кут
(градуси)'); grid on;

    subplot(2, 2, 4);
    distances_to_target = sqrt((robot_trajectory(:, 1) -
target_pos(1)).^2 + ...
                               (robot_trajectory(:, 2) -
target_pos(2)).^2);
    plot(time_vector, distances_to_target);
    title('Відстань до цілі'); xlabel('Час (с)');
ylabel('Відстань (м)'); grid on;

    fprintf('\n=== РЕЗУЛЬТАТИ СИМУЛЯЦІЇ ===\n');
    fprintf('Початкова позиція: (%.2f, %.2f)\n',
robot_trajectory(1, 1), robot_trajectory(1, 2));
    fprintf('Кінцева позиція: (%.2f, %.2f)\n',
robot_trajectory(end, 1), robot_trajectory(end, 2));
    fprintf('Цільова позиція: (%.2f, %.2f)\n', target_pos(1),
target_pos(2));
    fprintf('Точність паркування: %.3f м\n',
distances_to_target(end));
    fprintf('Загальна відстань: %.2f м\n',
sum(sqrt(diff(robot_trajectory(:,1)).^2 +
diff(robot_trajectory(:,2)).^2)));
    fprintf('Час симуляції: %.1f с\n', simulation_time);
    if distances_to_target(end) < 0.1
        fprintf('РЕЗУЛЬТАТ: Паркування УСПІШНЕ!\n');
    else
        fprintf('РЕЗУЛЬТАТ: Паркування потребує корекції\n');
    end
end
```

```
function distances = simulate_ultrasonic_sensors(robot_pos,
obstacles, sensor_angles, sensor_range)
    distances = zeros(1, length(sensor_angles));
    for i = 1:length(sensor_angles)
        sensor_angle = deg2rad(sensor_angles(i)) + robot_pos(3);
        ray_step = 0.01;
        max_steps = sensor_range / ray_step;
        for step = 1:max_steps
            x = robot_pos(1) + step * ray_step *
cos(sensor_angle);
            y = robot_pos(2) + step * ray_step *
sin(sensor_angle);
            collision = false;
            for j = 1:size(obstacles, 1)
                if x >= obstacles(j,1) && x <= obstacles(j,1) +
obstacles(j,3) && ...
                   y >= obstacles(j,2) && y <= obstacles(j,2) +
obstacles(j,4)
                    collision = true;
                    break;
                end
            end
            if collision
                distances(i) = step * ray_step;
                break;
            end
        end
        if distances(i) == 0
            distances(i) = sensor_range;
        end
    end
end

function target_spot = find_parking_spot(robot_pos,
parking_spots, obstacles, robot_size)
    target_spot = [];
    min_distance = inf;
    for i = 1:size(parking_spots, 1)
        spot_center = [parking_spots(i,1) +
parking_spots(i,3)/2, ...
                      parking_spots(i,2) +
parking_spots(i,4)/2];
        if parking_spots(i,3) >= robot_size(1) &&
parking_spots(i,4) >= robot_size(2)
            distance = norm(robot_pos(1:2) - spot_center);
            if distance < min_distance
                min_distance = distance;
                target_spot = parking_spots(i,:);
            end
        end
    end
end
```

```
function path = plan_path(start_pos, target_pos, obstacles, 
grid_size)
    grid_resolution = 0.1;
    x_grid = 0:grid_resolution:grid_size(1);
    y_grid = 0:grid_resolution:grid_size(2);
    path = [start_pos(1:2); target_pos(1:2)];
    num_waypoints = 10;
    waypoints = [];
    for i = 1:num_waypoints
        t = i / (num_waypoints + 1);
        waypoint = start_pos(1:2) * (1-t) + target_pos(1:2) * t;
        waypoints = [waypoints; waypoint];
    end
    path = [start_pos(1:2); waypoints; target_pos(1:2)];
end

function [v, omega] = motion_controller(current_pos, target_pos, 
max_speed)
    Kp_linear = 1.0;
    Kp_angular = 2.0;
    error_pos = target_pos(1:2) - current_pos(1:2);
    distance_error = norm(error_pos);
    target_angle = atan2(error_pos(2), error_pos(1));
    angle_error = target_angle - current_pos(3);
    while angle_error > pi
        angle_error = angle_error - 2*pi;
    end
    while angle_error < -pi
        angle_error = angle_error + 2*pi;
    end
    v = min(Kp_linear * distance_error, max_speed);
    omega = Kp_angular * angle_error;
    if abs(angle_error) > pi/4
        v = v * 0.5;
    end
end
```

Б.3 ParkingStateMachine.h

```
num ParkingState {
  IDLE,            // Очікування команд
  SCANNING,        // Пошук паркувального місця
  APPROACHING,     // Наближення до цільової позиції
  PARKING,         // Виконання маневру паркування
  PARKED,          // Успішно припарковано
  ERROR_STATE      // Стан помилки
};

class ParkingStateMachine {
private:
  ParkingState currentState;
```

```
  unsigned long stateStartTime;
  unsigned long stateTimeout;

public:
  ParkingStateMachine();
  void initialize();
  void update();
  bool transitionTo(ParkingState newState);
  ParkingState getCurrentState();
  void handleTimeout();
  const char* getStateString(ParkingState state);
};
```

Б.4 SensorArray.cpp

```
#include "SensorArray.h"

SensorArray::SensorArray() {
  sensors[0] = UltrasonicSensor(2, 3);   // Передній
  sensors[1] = UltrasonicSensor(4, 5);   // Лівий
  sensors[2] = UltrasonicSensor(6, 7);   // Правий
  sensors[3] = UltrasonicSensor(8, 9);   // Задній
}

void SensorArray::readAllSensors() {
  for(int i = 0; i < 4; i++) {
    sensorData[i].distance = sensors[i].measureDistance();
    sensorData[i].timestamp = millis();
    sensorData[i].isValid = validateReading(i);

    // Застосування медіанного фільтру
    sensorData[i].filteredDistance = medianFilter(i,
sensorData[i].distance);
  }
}

bool SensorArray::validateReading(int sensorIndex) {
  float distance = sensorData[sensorIndex].distance;

  // Перевірка діапазону
  if(distance < 2.0 || distance > 400.0) return false;

  // Перевірка швидкості зміни
  if(abs(distance - previousReadings[sensorIndex]) > 50.0) {
    return false;
  }

  previousReadings[sensorIndex] = distance;
  return true;
}
```

Б.5 MotorController.h

```
class MotorController {
private:
  int leftMotorPWM, rightMotorPWM;
  int leftDir1, leftDir2, rightDir1, rightDir2;
  float currentLeftSpeed, currentRightSpeed;
  unsigned long lastEncoderTime;

public:
  MotorController(int lPWM, int rPWM, int lD1, int lD2, int rD1,
int rD2);

  void initialize();
  void setSpeed(float leftSpeed, float rightSpeed);
  void moveForward(float speed);
  void moveBackward(float speed);
  void turnLeft(float speed);
  void turnRight(float speed);
  void stop();
  void emergencyStop();

  // PID регулятор швидкості
  void updatePID();
  float calculatePID(float setpoint, float measured, float&
prevError, float& integral);
};
```

Б.6 DataStructures.h

```
// Структура для зберігання даних сенсора
struct SensorReading {
  float distance;            // Відстань в см
  float filteredDistance;  // Відфільтроване значення
  unsigned long timestamp; // Час вимірювання
  bool isValid;            // Валідність показання
  uint8_t sensorId;        // Ідентифікатор датчика
};

// Позиція та орієнтація робота
struct RobotPose {
  float x, y;              // Координати в мм
  float theta;            // Орієнтація в радіанах
  float confidence;       // Достовірність позиції
  unsigned long timestamp;
};

// Команда руху для моторів
struct MotionCommand {
  float leftSpeed;        // Швидкість лівого мотора (-255 до
```

```
255)
  float rightSpeed;        // Швидкість правого мотора
  unsigned int duration;  // Тривалість команди в мс
  bool isEmergency;        // Прапорець аварійної команди
};

// Опис паркувального місця
struct ParkingSpace {
  float startX, startY;   // Початок простору
  float endX, endY;        // Кінець простору
  float width, length;    // Розміри простору
  float approachAngle;    // Кут підходу
  bool isOccupied;        // Зайнятість місця
  uint8_t quality;        // Якість місця (0-100)
};

// Конфігурація системи
struct SystemConfig {
  float maxSpeed;           // Максимальна швидкість
  float minParkingGap;      // Мінімальний зазор для паркування
  float safetyDistance;     // Безпечна відстань
  float wheelBase;          // База коліс
  float robotWidth;         // Ширина робота
  float robotLength;        // Довжина робота
};
```

Б.7 MathUtils.cpp

```
// Нормалізація кута в діапазон [-п, п]
float normalizeAngle(float angle) {
  while(angle > PI) angle -= 2 * PI;
  while(angle < -PI) angle += 2 * PI;
  return angle;
}

// Медіанний фільтр для сенсорних даних
float medianFilter(float values[], int size) {
  // Сортування масиву методом бульбашки
  for(int i = 0; i < size - 1; i++) {
    for(int j = 0; j < size - i - 1; j++) {
      if(values[j] > values[j + 1]) {
        float temp = values[j];
        values[j] = values[j + 1];
        values[j + 1] = temp;
      }
    }
  }
  return values[size / 2];
}

// Розрахунок відстані між двома точками
float calculateDistance(float x1, float y1, float x2, float y2)
```

```
{
  float dx = x2 - x1;
  float dy = y2 - y1;
  return sqrt(dx * dx + dy * dy);
}
```

Б.8 PerformanceTest.cpp

```
// Вимірювання часу виконання критичних операцій
void performanceTest() {
  unsigned long startTime, endTime;

  // Тест 1: Час зчитування всіх сенсорів
  startTime = micros();
  sensorArray.readAllSensors();
  endTime = micros();
  Serial.print("Sensor read time: ");
  Serial.print(endTime - startTime);
  Serial.println(" us");

  // Тест 2: Час планування траєкторії
  startTime = micros();
  pathPlanner.calculateOptimalPath(currentPose, targetPose);
  endTime = micros();
  Serial.print("Path planning time: ");
  Serial.print(endTime - startTime);
  Serial.println(" us");

  // Тест 3: Використання стеку
  int stackSize = measureStackUsage();
  Serial.print("Stack usage: ");
  Serial.print(stackSize);
  Serial.println(" bytes");
}

// Функція вимірювання використання стеку
int measureStackUsage() {
  extern int __heap_start, *__brkval;
  int v;
  return (int) &v - (__brkval == 0 ? (int) &__heap_start : (int)
__brkval);
}
```

Б.9 Реалізація на Arduino

```
* Система управління парковкою мобільного роботу
* з ультразвуковими датчиками
#include <NewPing.h>
```

```
#include <Servo.h>

// Константи системи
const int MAX_DISTANCE = 200;  // Максимальна відстань датчика
(см)
const int MIN_PARKING_DISTANCE = 15;  // Мінімальна відстань для
паркування (см)
const int SAFE_DISTANCE = 20;  // Безпечна відстань від перешкод
(см)
const int ROBOT_WIDTH = 25;    // Ширина роботу (см)
const int PARKING_SPOT_WIDTH = 40;  // Необхідна ширина
паркувального місця (см)

// Піни для ультразвукових датчиків
const int FRONT_TRIG = 2, FRONT_ECHO = 3;
const int LEFT_TRIG = 4, LEFT_ECHO = 5;
const int RIGHT_TRIG = 6, RIGHT_ECHO = 7;
const int REAR_TRIG = 8, REAR_ECHO = 9;

// Піни для моторів
const int LEFT_MOTOR_PWM = 10;
const int LEFT_MOTOR_DIR1 = 11;
const int LEFT_MOTOR_DIR2 = 12;
const int RIGHT_MOTOR_PWM = 13;
const int RIGHT_MOTOR_DIR1 = A0;
const int RIGHT_MOTOR_DIR2 = A1;

// Піни для світлодіодів статусу
const int LED_SEARCHING = A2;
const int LED_PARKING = A3;
const int LED_PARKED = A4;

// Ініціалізація датчиків
NewPing frontSensor(FRONT_TRIG, FRONT_ECHO, MAX_DISTANCE);
NewPing leftSensor(LEFT_TRIG, LEFT_ECHO, MAX_DISTANCE);
NewPing rightSensor(RIGHT_TRIG, RIGHT_ECHO, MAX_DISTANCE);
NewPing rearSensor(REAR_TRIG, REAR_ECHO, MAX_DISTANCE);

// Структури даних
struct SensorData {
  int front, left, right, rear;
  unsigned long timestamp;
};

struct Position {
  float x, y, theta;
};

struct ParkingSpot {
  float x, y, width, length;
  bool isValid;
};
```

```
// Стани системи
enum ParkingState {
  IDLE,
  SEARCHING,
  APPROACHING,
  PARKING,
  PARKED,
  ERROR_STATE
};

// Глобальні змінні
ParkingState currentState = IDLE;
SensorData sensors;
Position robotPos = {0, 0, 0};
ParkingSpot targetSpot;
unsigned long lastSensorRead = 0;
unsigned long stateStartTime = 0;

void setup() {
  Serial.begin(9600);

  // Налаштування пінів моторів
  pinMode(LEFT_MOTOR_PWM, OUTPUT);
  pinMode(LEFT_MOTOR_DIR1, OUTPUT);
  pinMode(LEFT_MOTOR_DIR2, OUTPUT);
  pinMode(RIGHT_MOTOR_PWM, OUTPUT);
  pinMode(RIGHT_MOTOR_DIR1, OUTPUT);
  pinMode(RIGHT_MOTOR_DIR2, OUTPUT);

  // Налаштування світлодіодів
  pinMode(LED_SEARCHING, OUTPUT);
  pinMode(LED_PARKING, OUTPUT);
  pinMode(LED_PARKED, OUTPUT);

  // Початковий стан
  stopMotors();
  updateStatusLEDs();

  Serial.println("Система парковки ініціалізована");
  Serial.println("Команди: START, STOP, STATUS");
}

void loop() {
  // Читання команд з серійного порту
  if (Serial.available()) {
    String command = Serial.readString();
    command.trim();
    processCommand(command);
  }

  // Читання датчиків кожні 100мс
  if (millis() - lastSensorRead > 100) {
    readSensors();
```

```
    lastSensorRead = millis();
  }

  // Основна логіка системи
  switch (currentState) {
    case IDLE:
      handleIdleState();
      break;
    case SEARCHING:
      handleSearchingState();
      break;
    case APPROACHING:
      handleApproachingState();
      break;
    case PARKING:
      handleParkingState();
      break;
    case PARKED:
      handleParkedState();
      break;
    case ERROR_STATE:
      handleErrorState();
      break;
  }

  updateStatusLEDs();
  delay(50);
}

void processCommand(String command) {
  if (command == "START") {
    if (currentState == IDLE) {
      changeState(SEARCHING);
      Serial.println("Початок пошуку паркувального місця");
    }
  } else if (command == "STOP") {
    changeState(IDLE);
    stopMotors();
    Serial.println("Зупинка системи");
  } else if (command == "STATUS") {
    printStatus();
  } else {
    Serial.println("Невідома команда: " + command);
  }
}

void readSensors() {
  sensors.front = frontSensor.ping_cm();
  sensors.left = leftSensor.ping_cm();
  sensors.right = rightSensor.ping_cm();
  sensors.rear = rearSensor.ping_cm();
  sensors.timestamp = millis();
```

```
  // Якщо датчик повертає 0, встановлюємо максимальне значення
  if (sensors.front == 0) sensors.front = MAX_DISTANCE;
  if (sensors.left == 0) sensors.left = MAX_DISTANCE;
  if (sensors.right == 0) sensors.right = MAX_DISTANCE;
  if (sensors.rear == 0) sensors.rear = MAX_DISTANCE;
}

void handleIdleState() {
  // Очікування команди START
  stopMotors();
}

void handleSearchingState() {
  // Пошук паркувального місця
  if (searchForParkingSpot()) {
    changeState(APPROACHING);
    Serial.println("Паркувальне місце знайдено!");
  } else {
    // Рух вперед для пошуку
    if (sensors.front > SAFE_DISTANCE) {
      moveForward(100);  // Повільний рух
    } else {
      // Поворот при перешкоді
      turnRight(500);  // Поворот на 500мс
      delay(100);
    }
  }

  // Таймаут пошуку (30 секунд)
  if (millis() - stateStartTime > 30000) {
    changeState(ERROR_STATE);
    Serial.println("Таймаут пошуку паркувального місця");
  }
}

void handleApproachingState() {
  // Наближення до паркувального місця
  if (approachParkingSpot()) {
    changeState(PARKING);
    Serial.println("Початок маневру паркування");
  }
}

void handleParkingState() {
  // Виконання маневру паркування
  if (performParkingManeuver()) {
    changeState(PARKED);
    Serial.println("Паркування завершено успішно!");
  }

  // Таймаут паркування (20 секунд)
  if (millis() - stateStartTime > 20000) {
    changeState(ERROR_STATE);
```

```
    Serial.println("Помилка під час паркування");
  }
}

void handleParkedState() {
  // Стан "припарковано"
  stopMotors();
  // Періодичний моніторинг датчиків
  if (sensors.front < MIN_PARKING_DISTANCE ||
      sensors.rear < MIN_PARKING_DISTANCE) {
    Serial.println("Попередження: занадто близько до
перешкоди!");
  }
}

void handleErrorState() {
  // Обробка помилок
  stopMotors();
  Serial.println("Система в стані помилки. Введіть STOP для
скидання.");
}

bool searchForParkingSpot() {
  // Спрощений алгоритм пошуку паркувального місця
  // Перевірка простору справа для паралельного паркування

  static int consecutiveGoodReadings = 0;
  static bool foundGap = false;

  if (sensors.right > PARKING_SPOT_WIDTH) {
    consecutiveGoodReadings++;
    if (consecutiveGoodReadings > 5) {  // 5 послідовних хороших
вимірювань
      foundGap = true;
    }
  } else {
    if (foundGap && consecutiveGoodReadings > 10) {
      // Знайдено підходяще місце
      targetSpot.x = robotPos.x;
      targetSp
```

ДОДАТОК В

Діаграма послідовності для процесу паркування

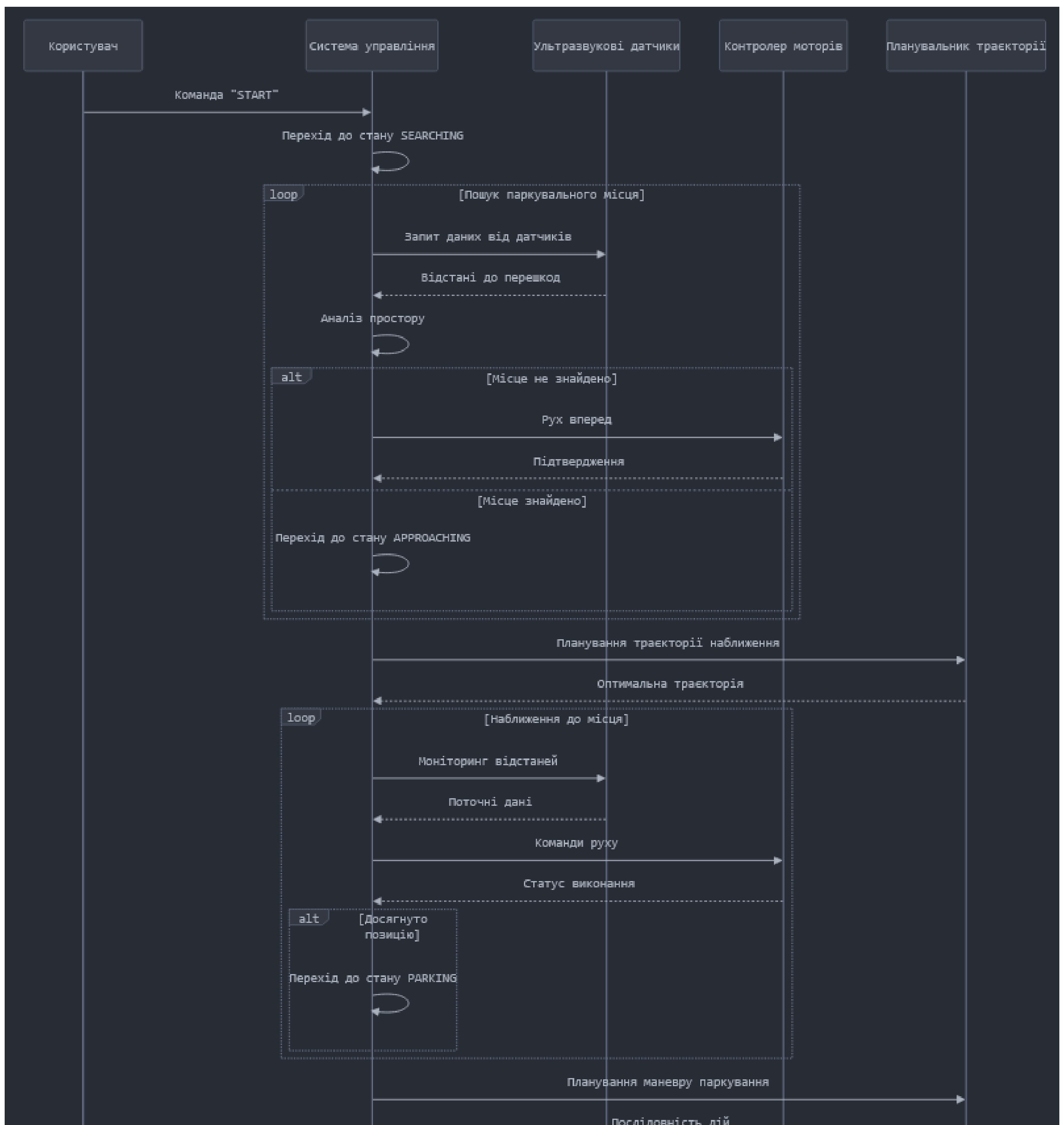


Рисунок В.1 – Діаграма послідовності для процесу паркування, частина 1

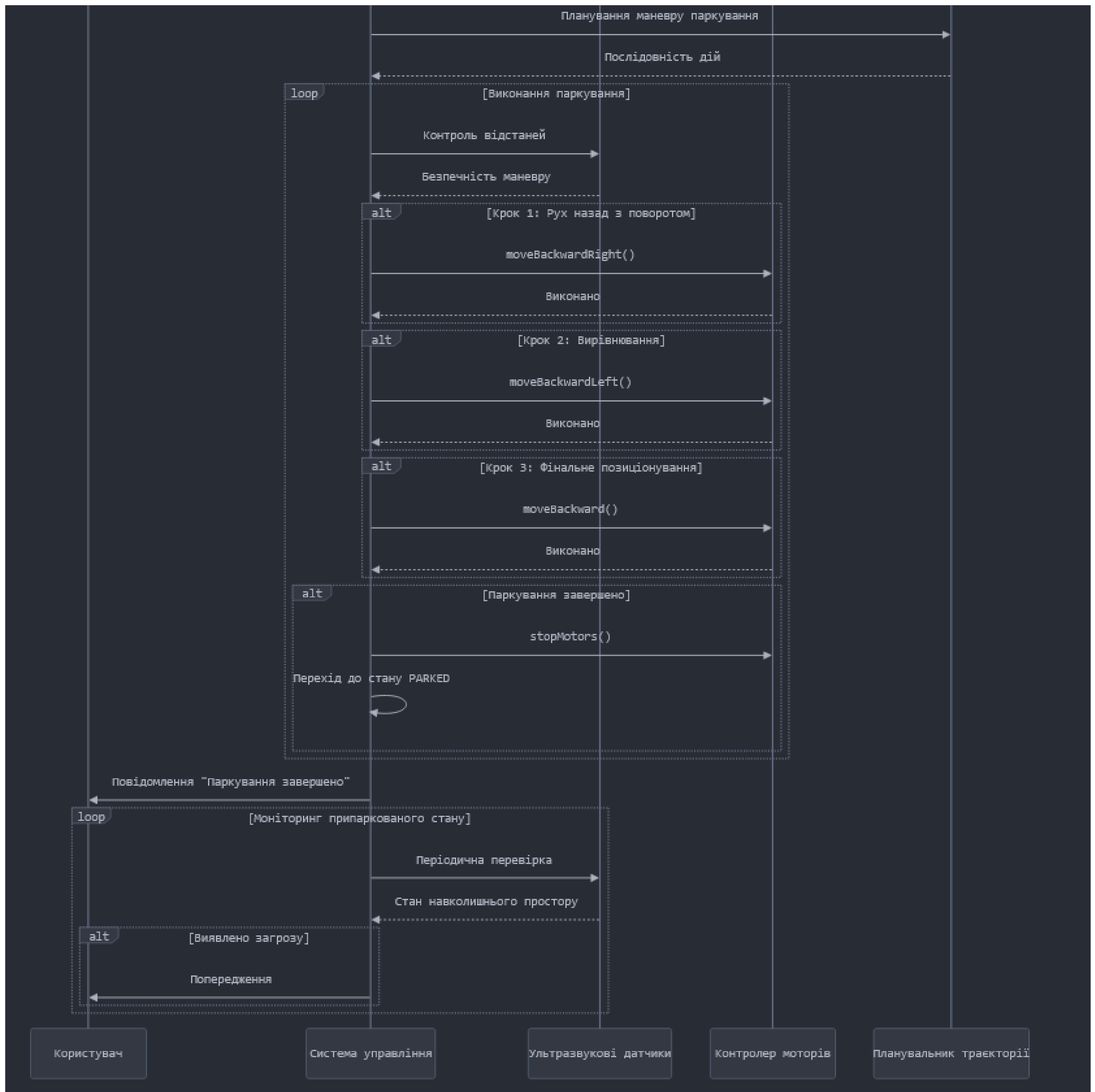


Рисунок В.2 – Діаграма послідовності для процесу паркування, частина 2